

PART

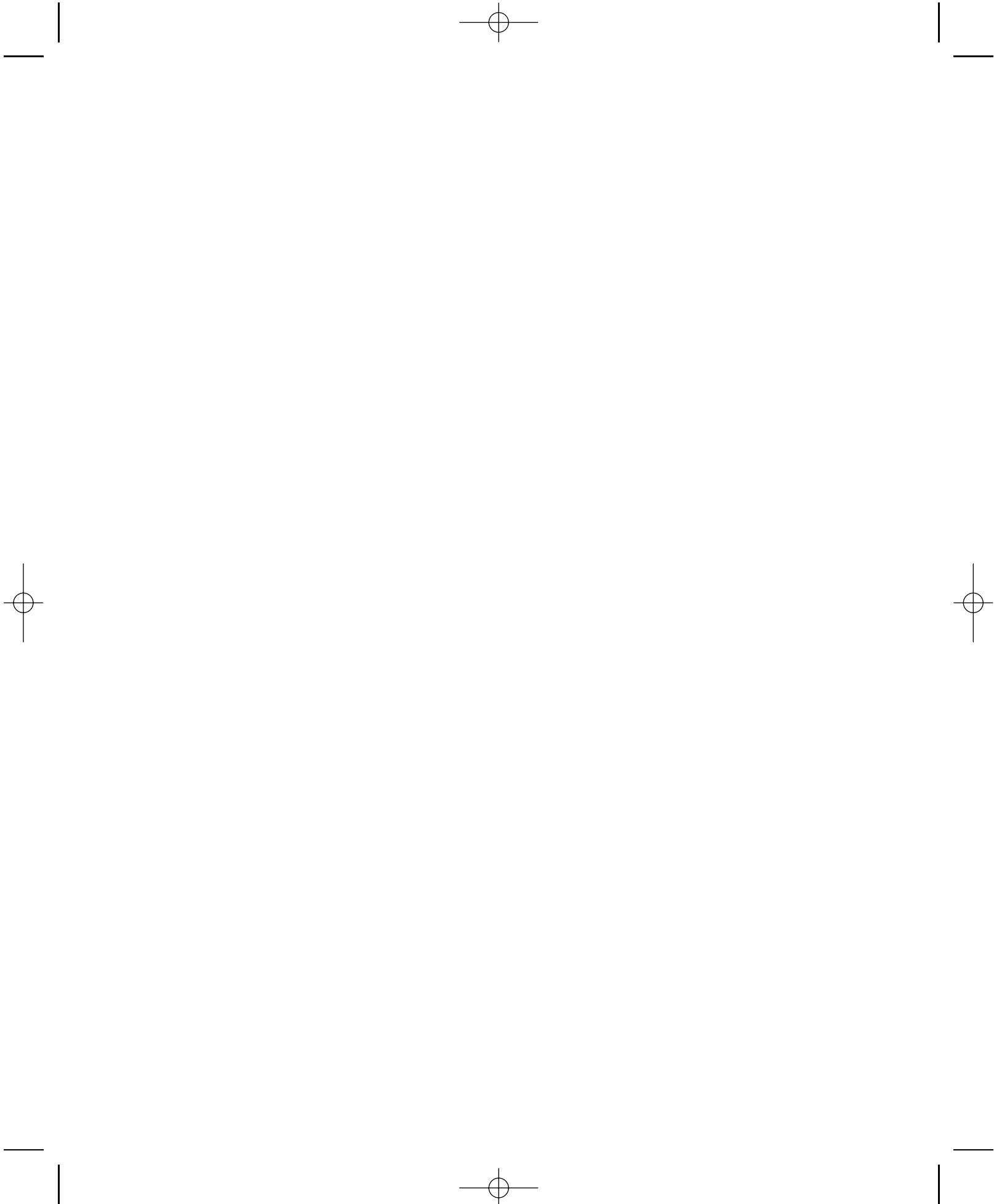
The Basics of Shell Scripting

Chapter 1: Scripting Quick Start and Review

Chapter 2: 24 Ways to Process a File Line-by-Line

Chapter 3: Automated Event Notification

Chapter 4: Progress Indicators Using a Series of Dots,
a Rotating Line, or Elapsed Time



CHAPTER**1**

Scripting Quick Start and Review

We are going to start out by giving a targeted refresher course. The topics that follow are short explanations of techniques that we always have to search the book to find; here they are all together in one place. The explanations range from showing the fastest way to process a file line-by-line to the simple matter of case sensitivity of UNIX and shell scripts. This should not be considered a full and complete list of scripting topics, but it is a very good starting point and it does point out a sample of the topics covered in the book. For each topic listed in this chapter there is a very detailed explanation later in the book.

We urge everyone to study this entire book. Every chapter hits a different topic using a different approach. The book is written this way to emphasize that there is never only one technique to solve a challenge in UNIX. All the shell scripts in this book are real-world examples of how to solve a problem. Thumb through the chapters, and you can see that we tried to hit most of the common (and some uncommon!) tasks in UNIX. All the shell scripts have a good explanation of the thinking process, and we always start out with the correct command syntax for the shell script targeting a specific goal. I hope you enjoy this book as much as I enjoyed writing it. Let's get started!

Case Sensitivity

UNIX is case sensitive. Because UNIX is case sensitive, our shell scripts are also case sensitive.

UNIX Special Characters

All of the following characters have a special meaning or function. If they are used in a way that their special meaning is not needed, they must be *escaped*. To escape,

4 **Part I ■ The Basics of Shell Scripting**

or remove its special function, the character must be immediately preceded with a backslash, `\`, or enclosed within `' '` forward tic marks (single quotes).

`\ / ; , . ~ # $ ? & * ( ) [ ] ' ' " + - ! ^ = | < >`

Shells

A *shell* is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions. This book works with the Bourne, Bash, and Korn shells. Shells are located in either the `/usr/bin/` directory or the `/bin/` directory, depending on the UNIX flavor and specific version.

Table 1-1

SHELL	DIRECTORY
Bourne	<code>/bin/sh</code> or <code>/usr/bin/sh</code>
Bash	<code>/bin/bash</code> or <code>/usr/bin/bash</code>
Korn	<code>/bin/ksh</code> or <code>/usr/bin/ksh</code>

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign or hash mark, `#`, describing the steps. There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, variables to read and store data, and the script may include *functions*.

We are going to write a lot of scripts in the next several hundred pages, and we should always start with a *clear goal* in mind. With a clear goal, we have a specific purpose for the script, and we have a set of expected results. We will also hit on some tips, tricks, and, of course, the gotchas in solving a challenge one way as opposed to another to get the same result. All techniques are not created equal.

Shell scripts and functions are both *interpreted*. This means they are not compiled. Both shell scripts and functions are ASCII text that is read by the shell command interpreter. When we execute a shell script, or function, a command interpreter goes through the ASCII text line-by-line, loop-by-loop, test-by-test, and executes each statement as each line is reached from the top to the bottom.

Functions

A function is written in much the same way as a shell script but is different in that it is defined, or written, within a shell script most of the time, and is called within the

Chapter 1 ■ Scripting Quick Start and Review 5

script. This way we can write a piece of code, which is used over and over, just once and use it without having to rewrite the code every time. We just call the function instead.

We can also define functions at the system level that is always available in our environment, but this is a topic for later discussion.

A function has the following form:

```
function function_name
{
    commands to execute
}
```

or

```
function_name ()
{
    commands to execute
}
```

When we write functions into our scripts we must remember to declare, or write, the function *before* we use it. The function must appear above the command statement calling the function. We can't use something that does not yet exist.

Running a Shell Script

A shell script can be executed in the following ways:

```
ksh shell_script_name
```

will create a Korn shell and execute the `shell_script_name` in the newly created Korn shell environment. The same is true for `sh` and `bash` shells.

```
shell_script_name
```

will execute `shell_script_name` if the *execution bit* is set on the file (see the manual page on the `chmod` command, **man chmod**). The script will execute in the shell that is *declared* on the first line of the shell script. If no shell is declared on the first line of the shell script, it will execute in the default shell, which is the user's system-defined shell. Executing in an unintended shell may result in a failure and give unpredictable results.

6 Part I ■ The Basics of Shell Scripting

Table 1-2 Different Types of Shells to Declare

COMMAND	DESCRIPTION
<code>#!/bin/sh</code> or <code>#!/usr/bin/sh</code>	Declares a Bourne shell
<code>#!/bin/ksh</code> or <code>#!/usr/bin/ksh</code>	Declares a Korn shell
<code>#!/bin/csh</code> or <code>#!/usr/bin/csh</code>	Declares a C shell
<code>#!/bin/bash</code> or <code>#!/usr/bin/bash</code>	Declares a Bourne-Again (Bash) shell

Declare the Shell in the Shell Script

Declare the shell! If we want to have complete control over how a shell script is going to run and in which shell it is to execute, we must *declare* the shell in *the first line of the script*. If no shell is declared, the script will execute in the default shell, defined by the system for the user executing the shell script. If the script was written, for example, to execute in Bash shell, `bash`, and the default shell for the user executing the shell script is the C shell, `csh`, the script will most likely have a failure during execution. To declare a shell, one of the declaration statements in Table 1-2 must appear on the *first line* of the shell script.

Comments and Style in Shell Scripts

Making good comments in our scripts is stressed throughout this book. What is intuitively obvious to us may be total Greek to others who follow in our footsteps. We have to write code that is readable and has an easy flow. This involves writing a script that is easy to read and easy to maintain, which means that it must have plenty of comments describing the steps. For the most part, the person who writes the shell script is not the one who has to maintain it. There is nothing worse than having to hack through someone else's code that has no comments to find out what each step is supposed to do. It can be tough enough to modify the script in the first place, but having to figure out the mindset of the author of the script will sometimes make us think about rewriting the entire shell script from scratch. We can avoid this by writing a clearly readable script and inserting plenty of comments describing what our philosophy is and how we are using the input, output, variables, and files.

For good style in our command statements, we need it to be readable. For this reason it is sometimes better, for instance, to separate a command statement onto three separate lines instead of stringing, or *piping*, everything together on the same line of code; it may be just too difficult to follow the pipe and understand what the expected result should be for a new script writer. However, in some cases it is more desirable to create a long pipe. But, again, it should have comments describing our thinking step by step. This way someone later will look at our code and say, "Hey, now that's a groovy way to do that."

Chapter 1 ■ Scripting Quick Start and Review 7

Command readability and step-by-step comments are just the very basics of a well-written script. Using a lot of comments will make our life much easier when we have to come back to the code after not looking at it for six months, and believe me; we will look at the code again. Comment everything! This includes, but is not limited to, describing what our variables and files are used for, describing what loops are doing, describing each test, maybe including expected results and how we are manipulating the data and the many data fields. A hash mark, #, precedes each line of a comment.

The *script stub* that follows is on this book's companion web site at www.wiley.com/go/michael2e. The name is `script.stub`. It has all the comments ready to get started writing a shell script. The `script.stub` file can be copied to a new filename. Edit the new filename, and start writing code. The `script.stub` file is shown in Listing 1-1.

```
#!/bin/bash
#
# SCRIPT: NAME_of_SCRIPT
# AUTHOR: AUTHORS_NAME
# DATE: DATE_of_CREATION
# REV: 1.1.A (Valid are A, B, D, T and P)
#
# (For Alpha, Beta, Dev, Test and Production)
#
# PLATFORM: (SPECIFY: AIX, HP-UX, Linux, OpenBSD, Solaris
#           or Not platform dependent)
#
# PURPOSE: Give a clear, and if necessary, long, description of the
#           purpose of the shell script. This will also help you stay
#           focused on the task at hand.
#
# REV LIST:
#           DATE: DATE_of_REVISION
#           BY: AUTHOR_of_MODIFICATION
#           MODIFICATION: Describe what was modified, new features, etc--
#
#
# set -n # Uncomment to check script syntax, without execution.
#       # NOTE: Do not forget to put the comment back in or
#       #       the shell script will not execute!
# set -x # Uncomment to debug this shell script
#
#####
#           DEFINE FILES AND VARIABLES HERE
#####
```

Listing 1-1 `script.stub` shell script starter listing

8 Part I ■ The Basics of Shell Scripting

```
#####
#               DEFINE FUNCTIONS HERE
#####

#####
#               BEGINNING OF MAIN
#####

# End of script
```

Listing 1-1 (*continued*)

The shell script starter shown in Listing 1-1 gives you the framework to start writing the shell script with sections to declare variables and files, create functions, and write the final section, BEGINNING OF MAIN, where the main body of the shell script is written.

Control Structures

The following control structures will be used extensively.

if ... then statement

```
if [ test_command ]
then

    commands

fi
```

if ... then ... else statement

```
if [ test_command ]
then

    commands

else

    commands

fi
```

Chapter 1 ■ Scripting Quick Start and Review 9

if ... then ... elif ... (else) statement

```
if [ test_command ]

then

    commands

elif [ test_command ]
then

    commands

elif [ test_command ]
then

    commands

.
.
.
else      (Optional)

    commands

fi
```

for ... in statement

```
for loop_variable in argument_list
do

    commands

done
```

while statement

```
while test_condition_is_true
do

    commands

done
```

until statement

```
until test_condition_is_true
do

    commands

done
```

10 Part I ■ The Basics of Shell Scripting

case statement

```

case $variable in

    match_1)

        commands_to_execute_for_1

        ;;

    match_2)

        commands_to_execute_for_2

        ;;

    match_3)

        commands_to_execute_for_3

        ;;

    .
    .
    .

    *)      (Optional - any other value)

        commands_to_execute_for_no_match

        ;;

esac

```

NOTE The last part of the case statement, shown here,

```

*)

    commands_to_execute_for_no_match

    ;;

```

is optional.

Using break, continue, exit, and return

It is sometimes necessary to *break* out of a `for` or `while` loop, *continue* in the next block of code, *exit* completely out of the script, or *return* a function's result back to the script that called the function.

Chapter 1 ■ Scripting Quick Start and Review 11

- The **break** command is used to terminate the execution of the entire loop, after completing the execution of all the lines of code up to the `break` statement. It then steps down to the code following the end of the loop.
- The **continue** command is used to transfer control to the next set of code, but it continues execution of the loop.
- The **exit** command will do just what one would expect: it exits the entire script. An integer may be added to an `exit` command (for example, `exit 0`), which will be sent as the return code.
- The **return** command is used in a function to send data back, or *return a result or return code*, to the calling script.

Here Document

A *here document* is used to redirect input *into* an interactive shell script or program. We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script. This is why it is called a here document: the required input is here, as opposed to somewhere else.

This is the syntax for a here document:

```
program_name <<LABEL
```

```
Program_Input_1
Program_Input_2
Program_Input_3
```

```
Program_Input_#
```

```
LABEL
```

Example:

```
/usr/local/bin/My_program << EOF
Randy
Robin
Rusty
Jim
EOF
```

Notice in the here documents that there are *no spaces* in the program input lines, between the two `EOF` labels. If a space is added to the input, the here document may fail. The input that is supplied must be the *exact* data that the program is expecting, and many programs will fail if spaces are added to the input.

12 Part I ■ The Basics of Shell Scripting

Shell Script Commands

The basis for the shell script is the automation of a series of commands. We can execute most any command in a shell script that we can execute from the command line. (One exception is trying to set an execution *suid* or *sgid*, *sticky bit*, within a shell script; it is not supported for security reasons.) For commands that are executed often, we reduce errors by putting the commands in a shell script. We will eliminate typos and missed device definitions, and we can do conditional tests that can ensure there are not any failures due to unexpected input or output. Commands and command structure will be covered extensively throughout this book.

Most of the commands shown in Table 1-3 are used at some point in this book, depending on the task we are working on in each chapter.

Table 1-3 UNIX Commands Review

COMMAND	DESCRIPTION
passwd	Changes user password
pwd	Prints current directory
cd	Changes directory
ls	Lists files in a directory
wildcards	* matches any number of characters; ? matches a single character
file	Prints the type of file
cat	Displays the contents of a file
pr	Displays the contents of a file
pg or page	Displays the contents of a file one page at a time
more	Displays the contents of a file one page at a time
clear	Clears the screen
cp or copy	Copies a file
chown	Changes the owner of a file
chgrp	Changes the group of a file
chmod	Changes file modes, permissions
rm	Removes a file from the system
mv	Renames a file
mkdir	Creates a directory

Chapter 1 ■ Scripting Quick Start and Review 13

Table 1-3 (continued)

COMMAND	DESCRIPTION
<code>rmdir</code>	Removes a directory
<code>grep</code>	Pattern matching
<code>egrep</code>	<code>grep</code> command for extended regular expressions
<code>find</code>	Locates files and directories
<code>>></code>	Appends to the end of a file
<code>></code>	Redirects, creates, or overwrites a file
<code> </code>	Strings commands together, known as a pipe
<code> </code>	Logical OR — <code>command1 command2</code> — execute <code>command2</code> if <code>command1</code> fails
<code>&</code>	Executes in background
<code>&&</code>	Logical AND — <code>command1 && command2</code> — execute <code>command2</code> if <code>command1</code> succeeds
<code>date</code>	Displays the system date and time
<code>echo</code>	Writes strings to standard output
<code>sleep</code>	Halts execution for the specified number of seconds
<code>wc</code>	Counts the number of words, lines, and characters in a file
<code>head</code>	Views the top of a file
<code>tail</code>	Views the end of a file
<code>diff</code>	Compares two files
<code>sdiff</code>	Compares two files side by side (requires 132-character display)
<code>spell</code>	Spell checker
<code>lp, lpr, enq, qprt</code>	Prints a file
<code>lpstat</code>	Status of system print queues
<code>enable</code>	Enables, or starts, a print queue
<code>disable</code>	Disables, or stops, a print queue
<code>cal</code>	Displays a calendar
<code>who</code>	Displays information about users on the system
<code>w</code>	Extended <code>who</code> command

(continued)

14 Part I ■ The Basics of Shell Scripting

Table 1-3 (continued)

COMMAND	DESCRIPTION
whoami	Displays \$LOGNAME or \$USER environment parameters
who am I	Displays login name, terminal, login date/time, and where logged in
f, finger	Displays information about logged-in users, including the users .plan and .project
talk	Enables two users to have a split-screen conversation
write	Displays a message on a user's screen
wall	Displays a message on all logged-in users' screens
rwall	Displays a message to all users on a remote host
rsh or remsh	Executes a command, or login, on a remote host
df	Displays filesystem statistics
ps	Displays information on currently running processes
netstat	Shows network status
vmstat	Shows virtual memory status
iostat	Shows input/output status
uname	Shows name of the current operating system, as well as machine information
sar	Reports system activity
basename	Displays base filename of a string parameter
man	Displays the online reference manual
su	Switches to another user, also known as super-user
cut	Writes out selected characters
awk	Programming language to parse characters
sed	Programming language for character substitution
vi	Starts the vi editor
emacs	Starts the emacs editor

Symbol Commands

The symbols shown in Table 1-4 are actually commands, and are used extensively in this book.

Chapter 1 ■ Scripting Quick Start and Review 15

Table 1-4 Symbol Commands

COMMAND	DESCRIPTION
()	Runs the enclosed command in a sub-shell
(())	Evaluates and assigns value to a variable and does math in a shell
\$(())	Evaluates the enclosed expression
[]	Same as the <code>test</code> command
< >	Used for string comparison
\$ ()	Command substitution
'command'	Command substitution

Variables

A *variable* is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. A variable is nothing more than a pointer to the actual data. We are going to use variables so much in our scripts that it will be unusual for us not to use them. In this book we are always going to specify a variable in uppercase — for example, `UPPERCASE`. Using uppercase variable names is not recommended in the real world of shell programming, though, because these uppercase variables may step on system environment variables, which are also in uppercase. Uppercase variables are used in this book to emphasize the variables and to make them stand out in the code. When you write your own shell scripts or modify the scripts in this book, make the variables lowercase text. To assign a variable to point to data, we use `UPPERCASE="value_to_assign"` as the assignment syntax. To access the data that the variable, `UPPERCASE`, is pointing to, we must add a dollar sign, `$`, as a prefix — for example, `$UPPERCASE`. To view the data assigned to the variable, we use `echo $UPPERCASE`, `print $UPPERCASE` for variables, or `cat $UPPERCASE`, if the variable is pointing to a file, as a command structure.

Command-Line Arguments

The command-line arguments `$1`, `$2`, `$3`, ... `$9` are positional parameters, with `$0` pointing to the actual command, program, shell script, or function and `$1`, `$2`, `$3`, ... `$9` as the arguments to the command.

The positional parameters, `$0`, `$2`, and so on in a function are for the function's use and may not be in the environment of the shell script that is calling the function. Where a variable is known in a function or shell script is called the *scope* of the variable.

16 Part I ■ The Basics of Shell Scripting

shift Command

The `shift` command is used to move positional parameters to the left; for example, `shift` causes `$2` to become `$1`. We can also add a number to the `shift` command to move the positions more than one position; for example, `shift 3` causes `$4` to move to the `$1` position.

Sometimes we encounter situations where we have an unknown or varying number of arguments passed to a shell script or function, `$1`, `$2`, `$3...` (also known as positional parameters). Using the `shift` command is a good way of processing each positional parameter in the order they are listed.

To further explain the `shift` command, we will show how to process an unknown number of arguments passed to the shell script shown in Listing 1-2. Try to follow through this example shell script structure. This script is using the `shift` command to process an unknown number of command-line arguments, or positional parameters. In this script we will refer to these as *tokens*.

```
#!/usr/bin/sh
#
# SCRIPT: shifting.sh
#
# AUTHOR: Randy Michael
#
# DATE: 12/30/2007
#
# REV: 1.1.A
#
# PLATFORM: Not platform dependent
#
# PURPOSE: This script is used to process all of the tokens which
# are pointed to by the command-line arguments, $1, $2, $3,etc...
#
# REV. LIST:
#

# Initialize all variables

TOTAL=0 # Initialize the TOTAL counter to zero

# Start a while loop

while true
do
    TOTAL=`expr $TOTAL + 1` # A little math in the
                           # shell script, a running
                           # total of tokens processed.
```

Listing 1-2 Example of using the `shift` command

Chapter 1 ■ Scripting Quick Start and Review 17

```

TOKEN=$1      # We always point to the $1 argument with a shift

                process each $TOKEN

shift         # Grab the next token, i.e. $2 becomes $1

done

echo "Total number of tokens processed: $TOTAL"

```

Listing 1-2 (*continued*)

We will go through similar examples of the `shift` command in great detail later in the book.

Special Parameters `$*` and `$@`

There are special parameters that allow accessing *all* the command-line arguments at once. `$*` and `$@` both will act the same unless they are enclosed in double quotes, " ".

Special Parameter Definitions

- The `$*` special parameter specifies *all* command-line arguments.
- The `$@` special parameter also specifies *all* command-line arguments.
- The "`$*`" special parameter takes the entire list as one argument with spaces between.
- The "`$@`" special parameter takes the entire list and separates it into separate arguments.

We can rewrite the shell script shown in Listing 1-2 to process an unknown number of command-line arguments with either the `$*` or `$@` special parameters, as shown in Listing 1-3.

```

#!/usr/bin/sh
#
# SCRIPT: shifting.sh
# AUTHOR: Randy Michael
# DATE:   12-31-2007
# REV:   1.1.A
# PLATFORM: Not platform dependent
#

```

Listing 1-3 Example using the special parameter `$*`

18 Part I ■ The Basics of Shell Scripting

```
# PURPOSE: This script is used to process all of the tokens which
# Are pointed to by the command-line arguments, $1, $2, $3, etc... -
#
# REV LIST:
#
#
# Start a for loop

for TOKEN in $*
do

    process each $TOKEN

done
```

Listing 1-3 (continued)

We could have also used the `$@` special parameter just as easily. As we see in the preceding code segment, the use of the `$@` or `$*` is an alternative solution to the same problem, and it was less code to write. Either technique accomplishes the same task.

Double Quotes, Forward Tics, and Back Tics

How do we know which one of these to use in our scripts, functions, and command statements? This decision causes the most confusion in writing scripts. We are going to set this straight now.

Depending on what the task is and the output desired, it is very important to use the correct enclosure. Failure to use these correctly will give unpredictable results.

We use `"`, double quotes, in a statement where we want to allow character or command substitution. Double quotes are required when defining a variable with data that contains white space, as shown here.

```
NAME="Randal K. Michael"
```

If the double quotes are missing we get the following error.

```
NAME=Randal K. Michael
-bash: K.: command not found
```

We use `'`, forward tics (single quotes), in a statement where we do *not* want character or command substitution. Enclosing in `'`, forward tics, is intended to use the *literal text* in the variable or command statement, without any substitution. All special meanings and functions are removed. It is also used when you want a variable reread each time it is used; for example, `'$PWD'` is used a lot in processing the `PS1` command-line prompt. Additionally, preceding the same string with a backslash, `\`, also removes the special meaning of a character, or string.

Chapter 1 ■ Scripting Quick Start and Review 19

We use ```, back ticks, in a statement where we want to execute a command, or script, and have its output substituted instead; this is *command substitution*. The ``` key is located to the left of the 1 key, and below the Escape key, Esc, on most keyboards. Command substitution is also accomplished by using the `$(command)` command syntax. We are going to see many different examples of these throughout this book.

Using awk on Solaris

We use `awk` a lot in this book to parse through lines of text. There is one special case where, on Solaris, we must to use **nawk** instead. If we need to specify a field separator other than a blank space, which is the default field delimiter, using `awk -F :`, for example, the `awk` statement will fail on a Solaris machine. To get around this problem, use `nawk` if we find the UNIX flavor is Solaris. Add the following code segment to the variable declaration section of all your shell scripts to eliminate the problem:

```
# Setup the correct awk usage. Solaris needs to
# use nawk instead of awk.

case $(uname) in
SunOS) alias awk=nawk
      ;;
esac
```

Using the echo Command Correctly

We use the **echo** command to display text. The `echo` command allows a lot of cursor control using backslash operators: `\n` for a new line, `\c` to continue on the same line, `\b` to backspace the cursor, `\t` for a tab, `\r` for a carriage return, and `\v` to move vertically one line. In Korn shell the `echo` command recognizes these command options by default. In Bash shell we must add the `-e` switch to the `echo` command, `echo -e "\n"` for one new line.

We can query the system for the executing shell by querying the `$SHELL` shell variable in the script. Many Linux distributions will execute in a Bash shell even though we specify Korn shell on the very first line of the script. Because Bash shell requires the use of the `echo -e` switch to enable the backslash operators, we can use a case statement to *alias* the `echo` command to `echo -e` if the executing shell is `*/bin/bash`. Now when we need to use the `echo` command, we are assured it will display text correctly.

Add the following code segment to all your Korn shell scripts in the variable declaration section, and this little problem is resolved:

```
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
```

20 Part I ■ The Basics of Shell Scripting

```
case $SHELL in
*/bin/bash) alias echo="echo -e"
            ;;
esac
```

Math in a Shell Script

We can do arithmetic in a shell script easily. The shell `let` command and the `((expr))` command expressions are the most commonly used methods to evaluate an integer expression. Later we will also cover the `bc` function to do floating-point arithmetic.

Operators

The shells use arithmetic operators from the C programming language (see Table 1-5), in decreasing order of precedence.

Table 1-5 Math Operators

OPERATOR	DESCRIPTION
<code>++ --</code> <code></editor query></code>	Auto-increment and auto-decrement, both prefix and postfix
<code>+</code>	Unary plus
<code>-</code>	Unary minus
<code>! ~</code>	Logical negation; binary inversion (one's complement)
<code>* / %</code>	Multiplication; division; modulus (remainder)
<code>+ -</code>	Addition; subtraction
<code><< >></code>	Bitwise left shift; bitwise right shift
<code><= >=</code>	Less than or equal to; greater than or equal to
<code>< ></code>	Less than; greater than
<code>== !=</code>	Equality; inequality (both evaluated left to right)
<code>&</code>	Bitwise AND
<code>^</code>	Bitwise exclusive OR
<code> </code>	Bitwise OR
<code>&&</code>	Logical AND
<code> </code>	Logical OR

Chapter 1 ■ Scripting Quick Start and Review 21

A lot of these math operators are used in the book, but not all. In this book we try to keep things very straightforward and not confuse you with obscure expressions.

Built-In Mathematical Functions

The shells provide access to the standard set of mathematical functions. They are called using C function call syntax. Table 1-6 shows a list of shell functions.

Table 1-6 Built-In Shell Functions

NAME	FUNCTION
<code>abs</code>	Absolute value
<code>log</code>	Natural logarithm
<code>acos</code>	Arc cosine
<code>sin</code>	Sine
<code>asin</code>	Arc sine
<code>sinh</code>	Hyperbolic sine
<code>cos</code>	Cosine
<code>sqrt</code>	Square root
<code>cosh</code>	Hyperbolic cosine
<code>tan</code>	Tangent
<code>exp</code>	Exponential function
<code>tanh</code>	Hyperbolic tangent
<code>int</code>	Integer part of floating-point number

We do not have any shell scripts in this book that use any of these built-in shell functions except for the `int` function to extract the integer portion of a floating-point number.

File Permissions, `suid` and `sgid` Programs

After writing a shell script we must remember to set the file permissions to make it *executable*. We use the `chmod` command to change the file's mode of operation. In addition to making the script executable, it is also possible to change the mode of the file to always execute as a particular user (`suid`) or to always execute as a member of a particular system group (`sgid`). This is called setting the *sticky bit*. If you try to `suid` or `sgid` a shell script, it is ignored for security reasons.

22 Part I ■ The Basics of Shell Scripting

Setting a program to always execute as a particular user, or member of a certain group, is often used to allow all users, or a set of users, to run a program in the proper environment. As an example, most system-check programs need to run as an administrative user, sometimes `root`. We do not want to pass out passwords, so we can just make the program always execute as `root` and it makes everyone's life easier. We can use the options shown in Table 1-7 in setting file permissions. Also, please review the `chmod` man page, **man chmod**.

By using combinations from the `chmod` command options, you can set the permissions on a file or directory to anything that you want. Remember that setting a shell script to `suid` or `sgid` is ignored by the system.

Table 1-7 `chmod` Permission Options

4000	Sets user ID on execution
2000	Sets group ID on execution
1000	Sets the link permission to directories or sets the save-text attribute for files
0400	Permits read by owner
0200	Permits write by owner
0100	Permits execute or search by owner
0040	Permits read by group
0020	Permits write by group
0010	Permits execute or search by group
0004	Permits read by others
0002	Permits write by others
0001	Permits execute or search by others

chmod Command Syntax for Each Purpose

The `chmod` command can be used with the octal file permission representation or by `r`, `w`, `x` notation. Both of these examples produce the same result.

To Make a Script Executable

```
chmod 754 my_script.sh
```

or

```
chmod u+rxw,g+rx,o+r my_script.ksh
```

The owner can read, write, and execute. The group can read and execute. The world can read.

Chapter 1 ■ Scripting Quick Start and Review 23

To Set a Program to Always Execute as the Owner

```
chmod 4755 my_program
```

The program will always execute as the owner of the file if it is not a shell script. The owner can read, write, and execute. The group can read and execute. The world can read and execute. So, no matter who executes this file, it will always execute as if the owner actually executed the program.

To Set a Program to Always Execute as a Member of the File Owner's Group

```
chmod 2755 my_program
```

The program will always execute as a member of the file's group, as long as the file is not a shell script. The owner of the file can read, write, and execute. The group can read and execute. The world can read and execute. So, no matter who executes this program, it will always execute as a member of the file's group.

To Set a Program to Always Execute as Both the File Owner and the File Owner's Group

```
chmod 6755 my_program
```

The program will always execute as the file's owner and as a member of the file owner's group, as long as the program is not a shell script. The owner of the file can read, write, and execute. The group can read and execute. The world can read and execute. No matter who executes this program, it will always execute as the file owner and as a member of the file owner's group.

Running Commands on a Remote Host

We sometimes want to execute a command on a remote host and have the result displayed locally. An example would be getting filesystem statistics from a group of machines. We can do this with the **rsh** command. The syntax is `rsh hostname command_to_execute`. This is a handy little tool but two system files will need to be set up on all of the hosts before the `rsh` command will work. The files are `.rhosts`, which would be created in the user's home directory and have the file permissions of 600 (permission to read and write by the owner only), and the `/etc/hosts.equiv` file.

For security reasons the `.rhosts` and `hosts.equiv` files, by default, are not set up to allow the execution of a remote shell. *Be careful!* The systems' security could be threatened. Refer to each operating system's documentation for details on setting up these files.

24 Part I ■ The Basics of Shell Scripting

Speaking of security, a better solution is to use Open Secure Shell (OpenSSH) instead of `rsh`. OpenSSH is a freeware encrypted replacement for `rsh`, `telnet`, and `ftp`, for the most part. To execute a command on another machine using OpenSSH, use the following syntax:

```
ssh user@hostname command_to_execute
```

or

```
ssh -l user hostname command_to_execute
```

This command prompts you for a password if the encryption key pairs have not been set up. Setting up the key pair relationships manually usually takes a few minutes, or you can use one of the `keyit` scripts shown in Listings 1-4 and 1-5 to set up the keys for you. The details of the procedure are shown in the `ssh` manual page (**man ssh**). You can download the OpenSSH code from <http://www.openssh.org>.

The `keyit.dsa` script in Listing 1-4 will set up DSA keys, if the DSA keys exist.

```
#!/bin/bash
#
# SCRIPT: keyit.dsa
# PURPOSE: This script is used to set up DSA SSH keys. This script must
#         be executed by the user who needs the keys setup.

REM_HOST=$1

cat $HOME/.ssh/id_dsa.pub | ssh $REM_HOST "cat >> ~/.ssh/authorized_keys"
```

Listing 1-4 `keyit.dsa` script used to set up DSA SSH keys

The `keyit.rsa` script in Listing 1-5 will set up the RSA keys, if the RSA keys exist.

```
#!/bin/bash
#
# SCRIPT: keyit.rsa
# PURPOSE: This script is used to set up RSA SSH keys.
#         This script must be executed by the user who needs the keys setup.

REM_HOST=$1

cat $HOME/.ssh/id_rsa.pub | ssh $REM_HOST "cat >> ~/.ssh/authorized_keys"
```

Listing 1-5 `keyit.rsa` script used to set up RSA SSH keys

If you need to set up the encryption keys for a new user, first `su` to that user ID, and then issue one of the following commands.

To set up DSA keys issue this command:

```
ssh-keygen -t dsa
```

Chapter 1 ■ Scripting Quick Start and Review 25

To set up RSA keys issue this one:

```
ssh-keygen -t rsa
```

Read the `ssh-keygen` man page for more details: `man ssh-keygen`.

Setting Traps

When a program is terminated before it would normally end, we can catch an exit signal. This is called a *trap*. Table 1-8 lists some of the exit signals.

Table 1-8 Exit Signals

0	—	Normal termination, end of script
1	SIGHUP	Hang up, line disconnected
2	SIGINT	Terminal interrupt, usually Ctrl + C
3	SIGQUIT	Quit key, child processes to die before terminating
9	SIGKILL	<code>kill -9</code> command, cannot trap this type of exit status
15	SIGTERM	<code>kill</code> command's default action
19	SIGSTOP	Stop, usually Ctrl + z

To see the entire list of supported signals for your operating system, enter the following command:

```
# kill -l [That's kill -(ell)]
```

This is a really nice tool to use in our shell scripts. On catching a trapped signal we can execute some cleanup commands before we actually exit the shell script. Commands can be executed when a signal is trapped. If the following command statement is added in a shell script, it will print to the screen “EXITING on a TRAPPED SIGNAL” and then make a clean exit on the signals 1, 2, 3, and 15. We cannot trap a `kill -9`.

```
trap 'echo "\nEXITING on a TRAPPED SIGNAL";exit' 1 2 3 15
```

We can add all sorts of commands that may be needed to clean up before exiting. As an example, we may need to delete a set of files that the shell script created before we exit.

User-Information Commands

Sometimes we need to query the system for some information about users on the system.

26 Part I ■ The Basics of Shell Scripting

who Command

The **who** command gives this output for each logged-in user: *username*, *tty*, *login time*, and *where* the user *logged in from*:

```
rmichael pts/0 Mar 13 10:24 192.168.1.104
root pts/1 Mar 15 10:43 (yogi)
```

w Command

The **w** command is really an extended **who**. The output looks like the following:

```
12:29PM up 27 days, 21:53, 2 users, load average 1.03, 1.17, 1.09
User tty login@ idle JCPU PCPU what
rmichael pts/0 Mon10AM 0 3:00 1 w
root pts/1 10:42AM 37 5:12 5:12 tar
```

Notice that the top line of the preceding output is the same as the output of the **uptime** command. The **w** command gives a more detailed output than the **who** command by listing job process time and total user process time, but it does not reveal *where* the users have logged in *from*. We often are interested in this for security purposes. One nice thing about the **w** command's output is that it also lists *what* the users are doing at the instant the command **w** is entered. This can be very useful.

last Command

The **last** command shows the history of who has logged in to the system since the **wtmp** file was created. This is a good tool when you need to do a little investigation of who logged in to the system and when. The following is example output:

```
root ftp booboo Aug 06 19:22 - 19:23 (00:01)
root pts/3 mrranger Aug 06 18:45 still logged in.
root pts/2 mrranger Aug 06 18:45 still logged in.
root pts/1 mrranger Aug 06 18:44 still logged in.
root pts/0 mrranger Aug 06 18:44 still logged in.
root pts/0 mrranger Aug 06 18:43 - 18:44 (00:01)
root ftp booboo Aug 06 18:19 - 18:20 (00:00)
root ftp booboo Aug 06 18:18 - 18:18 (00:00)
root tty0 Aug 06 18:06 still logged in.
root tty0 Aug 02 12:24 - 17:59 (4+05:34)
reboot ~ Aug 02 12:00
shutdown tty0 Jul 31 23:23
root ftp booboo Jul 31 21:19 - 21:19 (00:00)
root ftp bambam Jul 31 21:19 - 21:19 (00:00)
root ftp booboo Jul 31 20:42 - 20:42 (00:00)
root ftp bambam Jul 31 20:41 - 20:42 (00:00)
```

The output of the **last** command shows the username, the login port, where the user logged in from, the time of the login/logout, and the duration of the login session.

ps Command

The **ps** command will show information about current system processes. The **ps** command has many switches that will change what we look at. Table 1-9 lists some common command options.

Table 1-9 Common **ps** Command Options

<code>ps</code>	The user's currently running processes
<code>ps -f</code>	Full listing of the user's currently running processes
<code>ps -ef</code>	Full listing of all processes, except kernel processes
<code>ps -A</code>	All processes, including kernel processes
<code>ps -Kf</code>	Full listing of kernel processes
<code>ps auxw</code>	Wide listing sorted by percentage of CPU usage, %CPU

Communicating with Users

Communicate with the system's users and let them know what is going on! All Systems Administrators have the *maintenance window* where we can finally get control and handle some offline tasks. This is just one example of a need to communicate with the system users, if any are still logged in.

The most common way to get information to the system users is to use the `/etc/motd` file. This file is displayed each time the user logs in. If users stay logged in for days at a time they will not see any new messages of the day. This is one reason why real-time communication is needed. The commands shown in Table 1-10 allow communication to, or between, users who are currently logged into the system.

Table 1-10 Commands for Real-Time User Communication

<code>wall</code>	Writes a message on the screen of all logged-in users on the <i>local</i> host.
<code>rwall</code>	Writes a message on the screen of all logged-in users on a <i>remote</i> host.
<code>write</code>	Writes a message to an individual user. The user must currently be logged in.
<code>talk</code>	Starts an interactive program that allows two users to have a conversation. The screen is split in two, and both users can see what each person is typing.

28 Part I ■ The Basics of Shell Scripting

NOTE When using these commands, be aware that if a user is using a program — for example, an accounting software package — and has that program’s screen on the terminal, the user might not get the message or the user’s screen may become scrambled.

Uppercase or Lowercase Text for Easy Testing

We often need to test text strings like filenames, variables, file text, and so on, for comparison. It can sometimes vary so widely that it is easier to uppercase or lowercase the text for ease of comparison. The `tr` and `typeset` commands can be used to uppercase and lowercase text. This makes testing for things like variable input a breeze. Following are some examples of using the `tr` command:

Variable values:

Expected input	TRUE
Real input	TRUE
Possible input	true TRUE True True, and so on

Upcasing:

```
UPCASEVAR=$(echo $VARIABLE | tr '[a-z]' '[A-Z]')
```

Downcasing:

```
DOWNCASEVAR=$(echo $VARIABLE | tr '[A-Z]' '[a-z]')
```

In the preceding example of the `tr` command, we `echo` the string and use a pipe (`|`) to send the output of the `echo` statement to the `tr` command. As the preceding examples show, upcasing uses `'[a-z]'` `'[A-Z]'`.

NOTE The single quotes are required around the square brackets.

<code>'[a-z]'</code> <code>'[A-Z]'</code>	Used for lower to uppercase
<code>'[A-Z]'</code> <code>'[a-z]'</code>	Used for upper to lowercase

No matter what the user input is, we will always have the *stable* input of `TRUE`, if upcased, and `true`, if lowercased. This reduces our code testing and also helps the readability of the script.

We can also use `typeset` to control the attributes of a variable in the shell. In the previous example we are using the variable `VARIABLE`. We can set the attribute to always translate all of the characters to uppercase or lowercase. To set the case attribute of the variable `VARIABLE` to always translate characters assigned to it to uppercase, we use

```
typeset -u VARIABLE
```

Chapter 1 ■ Scripting Quick Start and Review 29

The `-u` switch to the `typeset` command is used for uppercase. After we set the attribute of the variable `VARIABLE`, using the `typeset` command, anytime we assign text characters to `VARIABLE` they are automatically translated to uppercase characters.

Example:

```
typeset -u VARIABLE
VARIABLE="True"
echo $VARIABLE
```

```
TRUE
```

To set the case attribute of the variable `VARIABLE` to always translate characters to lowercase, we use

```
typeset -l VARIABLE
```

Example:

```
typeset -l VARIABLE
VARIABLE="True"
echo $VARIABLE
true
```

Check the Return Code

Whenever we run a command there is a response back from the system about the last command that was executed, known as the *return code*. If the command was successful the return code will be 0, zero. If it was *not* successful the return will be something other than 0, zero. To check the return code we look at the value of the `$?` shell variable.

As an example, we want to check if the `/usr/local/bin` directory exists. Each of these blocks of code accomplishes the exact same thing:

```
test -d /usr/local/bin
if [ "$?" -eq 0 ] # Check the return code
then            # The return code is zero

    echo '/usr/local/bin does exist'

else            # The return code is NOT zero

    echo '/usr/local/bin does NOT exist'

fi
```

or

30 Part I ■ The Basics of Shell Scripting

```
if test -d /usr/local/bin
then
    # The return code is zero

    echo '/usr/local/bin does exist'

else
    # The return code is NOT zero

    echo '/usr/local/bin does NOT exist'
fi
```

or

```
if [ -d /usr/local/bin ]
then
    # The return code is zero

    echo '/usr/local/bin does exist'

else
    # The return code is NOT zero

    echo '/usr/local/bin does NOT exist'
fi
```

Notice that we checked the return code using `$?` once. The other examples use the control structure's built-in test. The built-in tests do the same thing of processing the return code, but the built-in tests hide this step in the process. All three of the previous examples give the exact same result. This is just a matter of personal choice and readability.

Time-Based Script Execution

We write a lot of shell scripts that we want to execute on a timed interval or run once at a specific time. This section addresses these needs with several examples.

Cron Tables

A *cron table* is a system file that is read every minute by the system and will execute any entry that is scheduled to execute in that minute. By default, any user can create a cron table with the `crontab -e` command, but the Systems Administrator can control which users are allowed to create and edit cron tables with the `cron.allow` and `cron.deny` files. When a user creates his or her own cron table, the commands, programs, or scripts will execute in that user's environment. It is the same thing as running the user's `$HOME/.profile` before executing the command.

The `crontab -e` command starts the default text editor, `vi` or `emacs`, on the user's cron table.

Chapter 1 ■ Scripting Quick Start and Review 31

NOTE When using the `crontab` command, the current user ID is the cron table that is acted on. To list the contents of the current user's cron table, issue the `crontab -l` command.

Cron Table Entry Syntax

It is important to know what each field in a cron table entry is used for. The following cron table entry executes the script, `/usr/local/bin/somescript.ksh`, at 3:15 a.m., January 8, on any day of the week that January 8 falls on. Notice that we used a *wildcard* for the weekday field.

_____	Minute of the Hour (0-59)
_____	Hour of the Day (0-23)
_____	Day of the Month (1-31)
_____	Month of the Year (1-12)
_____	Day of the Week (0 - 6 for Sunday through Saturday)
_____	Command to Execute (Full path is required)

```
15 3 8 1 * /usr/local/bin/somescript.bash 2>&1 >/dev/null
```

The following cron table entry is another example:

```
0 0 1 1 * /usr/bin/banner "Happy New Year" > /dev/console
```

At midnight on New Year's Eve, 00:00 hours on January 1, on any weekday, this cron table entry writes to the system's console (`/dev/console`) **Happy New Year** in large banner letters. Wildcard characters are defined in this table.

Wildcards:

*	Match any number of characters
?	Match a single character

at Command

Like a cron table, the **at** command executes commands based on time. Using the `at` command, we can schedule a job to run *once*, at a specific time. When the job is executed, the `at` command will send an email of the standard output and standard error to the user who scheduled the job to run, unless the output is redirected. As Systems Administrators we can control which users are allowed to schedule jobs with the `at.allow` and `at.deny` files. Refer to each operating system's man pages before modifying these files, and refer to them to learn the many ways to use the `at` command for timed controlled command execution. To execute a command in 10 minutes, use the following syntax:

```
echo '/usr/local/bin/somescript.bash' | at now + 10 minutes
```

32 Part I ■ The Basics of Shell Scripting

You can also use the `at` command interactively by typing the following:

```
at now + 10 minutes      Enter
```

Then type the full pathname of the command to execute, press `Enter`, and then press `Ctrl+D`. Here is an example:

```
at now + 10 minutes
/usr/local/bin/somescript.bash
Ctrl+D
```

For more `at` command options, see the `at` command manual page, **man at**.

Output Control

How is the script going to run? Where will the output go? These questions come under output control.

Silent Running

To execute a script in *silent mode* we can use the following syntax:

```
/PATH/script_name 2>&1 > /dev/null
```

In this command statement the `script_name` shell script will execute without any output to the screen. The reason for this is that the command is terminated with the following:

```
2>&1 > /dev/null
```

By terminating a command like this it redirects standard error (`stderr`), specified by file descriptor 2, to standard output (`stdout`), specified by file descriptor 1. Then we have another redirection to `/dev/null`, which sends all of the output to the bit bucket.

We can call this *silent running*. This means that there is absolutely no output from the script going to our screen. Inside the script there may be some output directed to files or devices, a particular terminal, or even the system's console, `/dev/console`, but none to the user's screen. This is especially useful when executing a script from one of the system's cron tables.

In the following example cron table entry, we want to execute a script named `/usr/local/bin/systemcheck.ksh`, which needs to run as the `root` user, every 15 minutes, 24 hours a day, 7 days a week, and not have any output to the screen. There will not be any screen output because we are going to end the cron table entry with the following:

```
2>&1 > /dev/null
```

Chapter 1 ■ Scripting Quick Start and Review 33

Inside the script it may do some kind of notification such as paging staff or sending output to the system's console, writing to a file or a tape device, but output such as `echo "Hello World"` would go to the *bit bucket*. But `echo "Hello World" > /dev/console` would go to the system's defined console if this command statement was within the shell script.

This cron table entry would need to be placed in the `root` cron table (must be logged in as the `root` user) with the following syntax:

```
5,20,35,50 * * * * /usr/local/bin/systemcheck.ksh 2>&1 >/dev/null
```

NOTE Most system check-type scripts need to be in the root cron table. Of course, a user must be logged in as root to edit root's cron table.

The preceding cron table entry would execute the `/usr/local/bin/systemcheck.ksh` every 15 minutes, at 5, 20, 35, and 50 minutes, each hour, 24 hours a day, 7 days a week. It would not produce any output to the screen due to the final `2>&1 > /dev/null`. Of course, the minutes selected to execute can be any. We sometimes want to spread out execution times in the cron tables so that we don't have a lot of CPU-intensive scripts and programs starting execution at the same time.

Using `getopts` to Parse Command-Line Arguments

The `getopts` command is built into the shell. It retrieves valid command-line options specified by a single character preceded by a `-` (minus sign) or a `+` (plus sign). To specify that a command switch requires an argument to the switch, it is followed by a `:` (colon). If the switch does not require any argument, the `:` should be omitted. All of the options put together are called the `OptionString`, and this is followed by some variable name. The argument for each switch is stored in a variable called `OPTARG`. If the entire `OptionString` is preceded by a `:` (colon), any unmatched switch option causes a `?` to be loaded into the `VARIABLE`. The form of the command follows:

```
getopts OptionString VARIABLE [ Argument ... ]
```

The easiest way to explain this is with an example. For a script we need seconds, minutes, hours, days, and a process to monitor. For each one of these we want to supply an argument — that is, `-s 5 -m10 -p my_backup`. In this we are specifying 5 seconds, 10 minutes, and the process is `my_backup`. Notice that there does not have to be a space between the switch and the argument, and they can be entered in any order. This is what makes `getopts` so great! The command line to set up our example looks like this:

```
SECS=0          # Initialize all to zero
MINUTES=0
HOURS=0
DAYS=0
PROCESS=        # Initialize to null
```

34 Part I ■ The Basics of Shell Scripting

```

while getopts :s:m:h:d:p: TIMED 2>/dev/null
do
    case $TIMED in
        s) SECS=$OPTARG
           ;;
        m) (( MINUTES = $OPTARG * 60 ))
           ;;
        h) (( HOURS = $OPTARG * 3600 ))
           ;;
        d) (( DAYS = $OPTARG * 86400 ))
           ;;
        p) PROCESS=$OPTARG
           ;;
        \?) usage
            exit 1
            ;;
    esac
done

(( TOTAL_SECONDS = SECONDS + MINUTES + HOURS + DAYS ))

```

There are a few things to note in the `getopts` command. The `getopts` command needs to be part of a `while` loop with a `case` statement within the loop for this example. On each option we specified, `s`, `m`, `h`, `d`, and `p`, we added a `:` (colon) after each switch. This tells `getopts` that an argument is required. The `:` (colon) *before* the `OptionString` list tells `getopts` that if an unspecified option is given, to set the `TIMED` variable to the `?` character. This allows us to call the `usage` function and `exit` with a return code of 1. The first thing to be careful of is that `getopts` does not care what arguments it receives, so we have to take action if we want to exit. The last thing to note is that the first line of the `while` loop has output redirection of standard error (file descriptor 2) to the bit bucket. Anytime an unexpected argument is encountered, `getopts` sends a message to standard error (file descriptor 2). Because we expect this to happen, we can just ignore the messages and discard them to `/dev/null`. We will study `getopts` a lot in this book.

Making a Co-Process with Background Function

We also need to cover setting up a co-process. A co-process is a communications link between a foreground and a background process. The most common question is *why* is this needed? In one of the scripts we are going to call a function that will handle all of the process monitoring for us while we do the timing control in the main script. The problem arises because *we need to run this function in the background and it has an infinite loop*. Within this background process-monitoring function there is an infinite loop. Without the ability to tell the loop to break out, it will continue to execute on its own after the main script, and function, is interrupted. We know what this causes — *one or more defunct processes*! From the main script we need a way to communicate with this loop, thus background function, to tell it to break out of the loop and exit the function cleanly when the countdown is complete and if the script is interrupted, `Ctrl+C`. To

Chapter 1 ■ Scripting Quick Start and Review 35

solve this little problem we kick off our `proc_watch` function as a *co-process* in the background. How do we do this, you ask? "*Pipe it to the background*" is the simplest way to put it, and that is what it looks like, too. Look at the next example code block in Listing 1-6.

```
#####
function trap_exit
{
    # Tell the co-process to break out of the loop
    BREAK_OUT='Y'
    print -p $BREAK_OUT # Use "print -p" to talk to the co-process
}
#####
function proc_watch
{
    # This function is started as a co-process!!!

    while :      # Loop forever
    do
        Some Code Here

        read $BREAK_OUT # Do NOT need a "-p" to read!
        if [[ $BREAK_OUT = 'Y' ]]
        then
            return 0
        fi
    done
}

#####
##### Start of Main #####
#####

### Set a Trap ###

trap 'trap_exit; exit 2' 1 2 3 15

TOTAL_SECONDS=300
BREAK_OUT='N'

proc_watch |&      # Start proc_watch as a co-process!!!!

PW_PID=$!          # Process ID of the last background job

until (( TOTAL_SECONDS == 0 ))
do
    (( TOTAL_SECONDS = TOTAL_SECONDS - 1 ))
done
```

Listing 1-6 Example code using a co-process

36 Part I ■ The Basics of Shell Scripting

```
    sleep 1
done

BREAK_OUT='Y'

# Use "print -p" to communicate with the co-process variable

print -p $BREAK_OUT

kill $PW_PID      # Kill the background co-process

exit 0
```

Listing 1-6 (continued)

In the code segment in Listing 1-6, we defined two functions. The `trap_exit` function will execute on `exit` signals 1, 2, 3, and 15. The other function is the `proc_watch` function, which is the function that we want to start as a background process. As you can see in `proc_watch`, it has an infinite loop. If the main script is interrupted without a means to exit the loop, within the function the loop *alone* will continue to execute! To solve this we start the `proc_watch` as a co-process by “piping it to the background” using *pipe ampersand*, `|&`, as a suffix. Then when we want to communicate to this co-process background function we use `print -p $VARIABLE_NAME`. Inside the co-process function we just use the standard `read $VARIABLE_NAME`. This is the mechanism that we are going to use to break out of the loop if the main script is interrupted on a trapped signal; of course, we *cannot* catch a `kill -9` with a trap.

Try setting up the scenario described previously with a background function that has an infinite loop. Then press the Ctrl+C key sequence to kill the main script, and do a `ps -ef | more`. You will see that the background *loop* is still executing! Get the PID, and do a `kill -9` on that PID to kill it. Of course, if the loop’s exit criteria is ever met, the loop will exit on its own.

Catching a Delayed Command Output

Have you ever had a hard time trying to catch the output of a command that has a delayed output? This can cause a lot of frustration when you *just miss it!* There is a little technique that allows you to catch these delayed responses. The trick is to use an `until` loop. Look at the code shown here:

```
OUTFILE="/tmp/outfile.out" # Define the output file
cat /dev/null > $OUTFILE   # Create a zero size output file

# Start an until loop to catch the delayed response
```

Chapter 1 ■ Scripting Quick Start and Review 37

```
until [ -s $OUTFILE ]
do
    delayed_output_command >> $OUTFILE
done

# Show the resulting output

more $OUTFILE
```

This code segment first defines an output file to store the delayed output data. We start with a zero-sized file and then enter an `until` loop that will continue until the `$OUTFILE` is no longer a zero-sized file, and the `until` loop exits. The last step is to show the user the data that was captured from the delayed output.

Fastest Ways to Process a File Line-by-Line

Most shell scripts work with files, and some use a file for data input. This next section shows the fastest methods studied in Chapter 2, “24 Ways to Process a File Line-by-Line.” The two fastest techniques for processing a file line-by-line are shown in Listings 1-7 and 1-8.

```
function while_read_LINE_bottom_FD_OUT
{
    # Zero out the $OUTFILE

    >$OUTFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE

    exec 4<&1
    exec 1> $OUTFILE

    while read LINE
    do
        echo "$LINE"
        :
    done < $INFILE

    # Restore standard output and close file
    # descriptor 4

    exec 1<&4
    exec 4>&-
}
```

Listing 1-7 Method 12 tied for first place

38 Part I ■ The Basics of Shell Scripting

The method in Listing 1-7 is tied for first place. This method uses my favorite input redirection for files by redirecting input at the bottom of the loop, after the `done` loop terminator. This method does use a file descriptor for redirecting standard output, `stdout`, to file descriptor 1.

The input redirection using `done < $INFILE` greatly speeds up any loop that requires file input. The nice thing about this method of input redirection is that it is intuitive to use for beginners to shell scripting. I was actually surprised that this method tied, actually won by 10 mS, using file descriptors for both input and output files, as shown in Listing 1-8.

```
function while_read_LINE_FD_IN_AND_OUT
{
    # Method 22

    # Zero out the $OUTFILE

    >$OUTFILE

    # Associate standard input with file descriptor 3
    # and redirect standard input to $INFILE

    exec 3<&0
    exec 0< $INFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE

    exec 4<&1
    exec 1> $OUTFILE

    while read LINE
    do
        echo "$LINE"
        :
    done

    # Restore standard output and close file
    # descriptor 4

    exec 1<&4
    exec 4>&-

    # Restore standard input and close file
    # descriptor 3

    exec 0<&3
    exec 3>&-
}
```

Listing 1-8 Method 22 tied for first place

Chapter 1 ■ Scripting Quick Start and Review 39

I tend not to use the method in Listing 1-8 when I write shell scripts, because it can be difficult to maintain through the code life cycle. If a user is not familiar with using file descriptors, a script using this method is extremely hard to understand. The method in Listing 1-7 produces the same timing results, and it is much easier to understand. Listing 1-9 shows the second-place loop method.

```
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
    # Method 16

    # Zero out the $OUTFILE

    >$OUTFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE

    exec 4<&1
    exec 1> $OUTFILE

    for LINE in $(cat $INFILE)
    do
        echo "$LINE"
        :
    done

    # Restore standard output and close file
    # descriptor 4

    exec 1<&4
    exec 4>&-
}
```

Listing 1-9 Method 16 made second place in timing tests

The method shown in Listing 1-10 is another surprise: a `for` loop using command substitution with file descriptor file output redirection.

```
for_LINE_cat_FILE_FD_OUT ()
{
    # Method 15

    # Zero out the $OUTFILE

    >$OUTFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE
```

Listing 1-10 Method 15 made third place in timing tests

40 Part I ■ The Basics of Shell Scripting

```
exec 4<&1
exec 1> $OUTFILE

for LINE in `cat $INFILE`
do
    echo "$LINE"
:
done

# Restore standard output and close file
# descriptor 4

exec 1<&4
exec 4>&-
}
```

Listing 1-10 (*continued*)

The method in Listing 1-11 is the fastest method to process a file line-by-line that does not use file descriptors.

```
function while_read_LINE_bottom
{
    # Method 2

    # Zero out the $OUTFILE

    >$OUTFILE

    while read LINE
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
}
```

Listing 1-11 The fastest method not using file descriptors

I use this technique in almost every shell script that does file parsing, simply because of the ease of maintaining the shell script throughout the life cycle.

Using Command Output in a Loop

The technique shown here is a nice little trick to execute a command and use the command's output in a loop without using a pipe:

Chapter 1 ■ Scripting Quick Start and Review 41

```
while read LINE
do
    echo "$LINE"

done < <(command)
```

I know this looks a bit odd. I got this trick from one of my co-workers, Brian Beers. What we are doing here is input redirection from the bottom of the loop, after the `done` loop terminator, specified by the `done <` notation. The `< (command)` notation executes the command and points the command's output into the bottom of the loop.

NOTE The space between `< <` in `< <(command)` is required!

Mail Notification Techniques

In a lot of the shell scripts in this book it is a good idea to send notifications to users when errors occur, when a task is finished, and for many other reasons. Some of the email techniques are shown in this section.

Using the mail and mailx Commands

The most common notification method uses the **mail** and **mailx** commands. The basic syntax of both these commands is shown in Listing 1-12.

```
mail -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE

or

cat $MAIL_FILE | mail -s "This is the subject" $MAILOUT_LIST

or

mailx -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE

or

cat $MAIL_FILE | mailx -s "This is the subject" $MAILOUT_LIST
```

Listing 1-12 Examples using the mail and mailx commands

Not all systems support the **mailx** command, but the systems that do have support use the same syntax as the **mail** command. To be safe when dealing with multiple UNIX platforms, always use the **mail** command.

Using the sendmail Command to Send Outbound Mail

In one shop I worked at I could not send outbound mail from the user named **root**. The *from* field had to be a valid email address that was recognized by the mail server,

42 Part I ■ The Basics of Shell Scripting

and root is not valid. To get around this little problem I changed the command that I used from mail to sendmail. The sendmail command allows us to add the -f switch to indicate a valid internal email address for the *from* field. The sendmail command is in /usr/sbin/sendmail on AIX, HP-UX, Linux, and OpenBSD, but on SunOS the location changed to /usr/lib/sendmail. Look at the send_notification function in Listing 1-13.

```
function send_notification
{
  if [ -s $MAIL_FILE -a "$MAILOUT" = "TRUE" ];
  then
    case $(uname) in
      AIX|HP-UX|Linux|OpenBSD) SENDMAIL="/usr/sbin/sendmail"
                              ;;
      SunOS)                  SENDMAIL="/usr/lib/sendmail"
                              ;;
      esac

    echo "\nSending email notification"
    $SENDMAIL -f randy@$THISHOST $MAIL_LIST < $MAIL_FILE

  fi
}
```

Listing 1-13 send_notification function

The mail and mailx commands also support specifying a from field by using the -r switch, as shown in Listing 1-14.

```
mail -r randy@$THISHOST -s "This is the subject" \
$MAILOUT_LIST < $MAIL_FILE

or

cat $MAIL_FILE | mail -r randy@$THISHOST -s
"This is the subject" $MAILOUT_LIST

or

mailx -r randy@$THISHOST -s "This is the subject"
$MAILOUT_LIST < $MAIL_FILE

or

cat $MAIL_FILE | mailx -r randy@$THISHOST -s
"This is the subject" $MAILOUT_LIST
```

Listing 1-14 Specifying a from field with mail and mailx

Both techniques should allow you to get the message out quickly.

Creating a Progress Indicator

Anytime that a user is forced to wait as a long process runs, it is an excellent idea to give the user some feedback. This section deals with progress indicators.

A Series of Dots

The `echo` command prints a single dot on the screen, and the backslash `c`, `\c`, specifies a continuation on the same line without a new line or carriage return. To make a series of dots we will put this single command in a loop, with some sleep time between each dot. We will use a `while` loop that loops forever with a 10-second sleep between printing each dot on the screen:

```
while true
do
    echo ".\c"
    sleep 10
done
```

A Rotating Line

The `rotate_line` function, shown in Listing 1-15, shows what appears to be a rotating line as the process runs.

```
function rotate_line
{
    INTERVAL=1      # Sleep time between "twirls"
    TCOUNT="0"      # For each TCOUNT the line twirls one increment

    while :         # Loop forever...until this function is killed
    do
        TCOUNT=`expr $TCOUNT + 1`    # Increment the TCOUNT

        case $TCOUNT in
            "1") echo '-'\b\c"
                sleep $INTERVAL
                ;;
            "2") echo '\'\b\c"
                sleep $INTERVAL
                ;;
            "3") echo '|'\b\c"
                sleep $INTERVAL
                ;;
        esac
    done
}
```

Listing 1-15 rotate_line function

44 Part I ■ The Basics of Shell Scripting

```

        "4") echo "\b\c"
            sleep $INTERVAL
            ;;
        *)    TCOUNT="0" ;; # Reset the TCOUNT to "0", zero.
    esac
done
}
```

Listing 1-15 (*continued*)

To use `rotate_line` in a shell script, use the technique shown in Listing 1-16 to start and stop the rotation.

```

#####
##### Beginning of Main #####
#####

rotate_line & # Run the function in the background

ROTATE_PID=$! # Capture the PID of the last background process

/usr/local/bin/my_time_consuming_task.ksh

# Stop the rotating line function

kill -9 $ROTATE_PID

# Cleanup...this removes the left over line.

echo "\b\b "
```

Listing 1-16 Using `rotate_line` in a shell script

Elapsed Time

The `elapsed_time` function requires as input the number of seconds, and produces as output the time represented in hours, minutes, and seconds. Listing 1-17 shows the `elapsed_time` function.

```

elapsed_time ()
{
    SEC=$1
    (( SEC < 60 )) && echo "[Elapsed time: $SEC seconds]\c"
```

Listing 1-17 `elapsed_time` function

Chapter 1 ■ Scripting Quick Start and Review 45

```
(( SEC >= 60  && SEC < 3600 )) && echo "[Elapsed time: $(( SEC / 60 )) \
min $(( SEC % 60 )) sec]\c"

(( SEC > 3600 )) && echo "[Elapsed time: $(( SEC / 3600 )) hr \
$(( (SEC % 3600) / 60 )) min $(( (SEC % 3600) % 60 )) sec]\c"
}
```

Listing 1-17 (continued)

An example using the `elapsed_time` function follows:

```
elapsed_time 6465
[Elapsed time: 1 hr 47 min 45 sec]
```

Note that the `elapsed_time` function continues on the same line of text without a new line. To add a new line, change the ending `\c` to `\n` on each of the three `echo` commands.

Working with Record Files

We often want to add a filename to each record in a record file. The code shown in Listing 1-18 shows a merge process where we loop through a list of record filenames and, as we append each record file to build the big batch-processing file, we append the record filename to each record. Check out Listing 1-18, and we will cover the details at the end.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst

while read RECORDFILENAME
do

    sed s/${$(basename $RECORDFILENAME)}/g \
$RECORDFILENAME >> $MERGERECORDFILE

done < $RECORDFILELIST
```

Listing 1-18 Code for a merge process for fixed-length record files

Listing 1-18 is a merge script for fixed-length record files. We first define a couple of files. The `MERGERECORDFILE` variable definition specifies the name of the resulting merged record data file. The `RECORDFILELIST` variable defines the file that contains a list of record files that must be merged. Notice the use of the `basename` command to strip the directory part of the `$RECORDFILENAME` on each loop iteration. As we feed the `while` loop from the bottom, after `done`, to process the `$RECORDFILELIST`

46 Part I ■ The Basics of Shell Scripting

file line-by-line, we assign a new record file to the `RECORDFILENAME` variable. We use this new value in our `sed` statement that appends the record filename to the end of each record in the file. As the last step, this record file is appended to the end of the `$MERGERECORDFILE` file. Listing 1-19 shows the same technique working with variable-length record files.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst
FD= :

while read RECORDFILENAME
do

    sed s/${FD}$(basename $RECORDFILENAME)/g \
    $RECORDFILENAME >> $MERGERECORDFILE

done < $RECORDFILELIST
```

Listing 1-19 Code for a merge process for variable-length record files

Listing 1-19 is a merge script for variable-length record files. The only difference between this script and Listing 1-18 is that we added the field delimiter variable, `FD`. We define the field delimiter and then add the field delimiter between the end of the record and the record filename.

Working with Strings

A useful thing to know when working with fixed-length record files is the string length. The length of a string assigned to a variable can be found using the following syntax:

```
echo ${#VAR}
```

For example, if I assign the `VAR` variable the string `1234567890`, the string length should be 10:

```
VAR=1234567890
echo ${#VAR}
10
```

We sometimes need the string length for fixed-length records to pad the extra space in a data field that the data does not fill up. As an example, the `TOTAL` field is defined as characters 46–70, which is 25 characters long. It is unlikely that anyone would ever owe that kind of cash, but that is the data field we have to work with. This field is right-justified and filled with leading zeros, as shown here:

```
0000000000000000000000009825
```

Chapter 1 ■ Scripting Quick Start and Review 47

If we change a record field and the data is not as long as the data field, we need to pad the rest of the data field with something — in this case, zeros. The hard way to pad the data with leading zeros is to check the original data for the string length, and then get the string length of the new data and start looping to add the correct number of leading zeros followed by the new data. It is so much easier to do this task with the `typeset` command. The `-Z` switch specifies right justification with leading zeros. Well, that's just what we need here. Our `TOTAL` variable is 25 characters long. The following `typeset` command makes this definition for us:

```
typeset -Z25 TOTAL
```

Now we can change the `TOTAL` data field to a different value and not worry about how many leading zeros to add. As an example, let's change the total to 0 — we paid this bill off!

```
TOTAL=0
echo $TOTAL
000000000000000000000000
```

The `typeset` command can do a lot of the work for us. Table 1-11 shows more `typeset` options you might find handy.

Table 1-11 Options for the `typeset` Command

SWITCH	RESULTING TYPESET
<code>-L</code>	Left-justify and remove blank spaces.
<code>-R</code>	Right-justify and fill with leading blanks.
<code>-Zn</code>	Right-justify to <i>n</i> length and fill with leading zeros if the first non-blank character is a digit.
<code>-i</code>	Variable is an integer.
<code>-l</code>	Convert uppercase characters to lowercase characters.
<code>-u</code>	Convert lowercase characters to uppercase characters.
<code>-x</code>	Automatically export this variable.

NOTE Using `+` instead of `-` turns off the `typeset` definition.

Creating a Pseudo-Random Number

There is a built-in shell variable that will create a pseudo-random number called `RANDOM`. The following code segment creates a pseudo-random number between 1 and an upper limit defined by the user:

48 Part I ■ The Basics of Shell Scripting

```
RANDOM=$((RANDOM % $UPPER_LIMIT + 1))
UPPER_LIMIT=$1

RANDOM_NUMBER=$((RANDOM % $UPPER_LIMIT + 1))

echo "$RANDOM_NUMBER"
```

If the user specified the `UPPER_LIMIT` to be 100, the result would be a pseudo-random number between 1 and 100.

Using `/dev/random` and `/dev/urandom`

We can also use the `/dev/random` and `/dev/urandom` character special files to produce pseudo-random numbers. Trying to read the `/dev/random` and `/dev/urandom` character special files directly with `dd` returns non-printable binary data. To get some usable random numbers, we need to pipe the `dd` command output to the `od`, octal dump, command. Specifically, we use `od` to dump the data to an unsigned integer. The code shown in Listing 1-20 assigns an unsigned random integer to the `RN` variable.

```
RN=$((dd if=/dev/random count=1 2>/dev/null \
| od -t u4 | awk '{print $2}' | head -n 1))
```

Listing 1-20 Using `/dev/random` to return a random number

Notice in Listing 1-20 that the `dd` command uses `/dev/random` as the input file. We set the `count` equal to 1 to return one byte of data. Then, *and this is important*, we send all the standard error output, specified by file descriptor 2, to the bit bucket. If we omit the `2 > /dev/null` redirection, we get unwanted data. The remaining standard output is piped to the `od` command to convert the binary data to an unsigned integer, specified by the `-t u4` command switch. By changing the value assigned to `u`, we change the length of the random number returned. To create a 64-bit, not 64-character, random number, we just change the `-t u4` to `-t u8`. More examples are in Chapter 11, “Pseudo-Random Number and Data Generation.”

Checking for Stale Disk Partitions in AIX

Ideally, we want the stale disk partition value to be zero, 0. If the value is greater than zero we have a problem. Specifically, the mirrored disks in this Logical Volume are not in sync, which translates to a worthless mirror. Take a look at the following command statement:

```
LV=apps_lv

NUM_STALE_PP=$(lslv -L $LV | grep "STALE PP" | awk '{print $3}')
```

Chapter 1 ■ Scripting Quick Start and Review 49

This statement saves the number of stale PPs into the `NUM_STALE_PP` variable. We accomplish this feat by command substitution, specified by the `VARIABLE=$( commands )` notation.

Automated Host Ping

Depending on the operating system that you are running, the `ping` command varies if you want to send three pings to each host to see if the machines are up. The `ping_host` function shown in Listing 1-21 can ping from AIX, HP-UX, Linux, OpenBSD, and SunOS machines.

```
function ping_host
{
  HOST=$1 # Grab the host to ping from ARG1.
  PING_COUNT=3
  PACKET_SIZE=54

  # This next case statement executes the correct ping
  # command based on the Unix flavor

  case $(uname) in

    AIX|OpenBSD|Linux)
      ping -c${PING_COUNT} $HOST 2>/dev/null
      ;;

    HP-UX)
      ping $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
      ;;

    SunOS)
      ping -s $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
      ;;

    *)
      echo "\nERROR: Unsupported Operating System - $(uname)"
      echo "\n\t...EXITING...\n"
      exit 1

  esac
}
```

Listing 1-21 `ping_host` function

The main body of the shell script must supply the hostname to ping. This is usually done with a `while` loop.

Highlighting Specific Text in a File

The technique shown here highlights specific text in a file with reverse video while displaying the entire file. To add in the reverse video piece, we have to do some

50 Part I ■ The Basics of Shell Scripting

command substitution within the `sed` statement using the `tput` commands. Where we specify the `new_string`, we will add in the control for reverse video using command substitution, one to turn highlighting on and one to turn it back off. When the command substitution is added, our `sed` statement will look like the following:

```
sed s/string/${tput smso}string${tput rmso}/g
```

We also want the string to be assigned to a variable, as in the next command:

```
sed s/"$STRING"/$(tput smso)"$STRING"$(tput rmso)/g
```

Notice the double quotes around the string variable, `"$STRING"`. Do not forget to add the double quotes around variables!

As an experiment using command substitution, try this next command statement to highlight the machine's hostname in the `/etc/hosts` file on any UNIX machine:

```
cat /etc/hosts | sed s/'hostname'/${tput smso}'hostname'$(tput rmso)/g
```

Keeping the Printers Printing

Keeping the printers enabled in a large shop can sometimes be overwhelming. There are two techniques to keep the printers printing. One technique is for the AIX "classic" printer subsystem, and the other is for System V and CUPS printing.

AIX "Classic" Printer Subsystem

To keep AIX "classic" printer subsystem print queues running, use either of the following commands:

```
enable $(enq -AW | tail +3 | grep DOWN | awk '{print $1}') 2>/dev/null
```

or

```
enable $(lpstat -W | tail +3 | grep DOWN | awk '{print $1}') 2>/dev/null
```

System V and CUPS Printing

To keep System V and CUPS printers printing any of the following commands:

```
lpc enable $(lpstat -a | grep 'not accepting' | awk '{print $1}')
```

```
lpc start $(lpstat -p | grep disabled | awk '{print $2}')
```

```
lpc up all # Enable all printing and queuing
```

It is a good idea to use the root cron table to execute the appropriate command every 15 minutes or so.

Automated FTP File Transfer

You can use a here document to script an FTP file transfer. The basic idea is shown here:

```
ftp -i -v -n wilma <<END_FTP

user randy mypassword
binary
lcd /scripts/download
cd /scripts
get auto_ftp_xfer.ksh
bye

END_FTP
```

Using rsync to Replicate Data

We use `rsync` much the same way that we use `rcp` and `scp`. All three methods require a source and destination file or directory, and there are command-line switches that allow us to save file permissions, links, ownership, and so on, as well as to copy directory structures recursively. A few examples are the best way to learn how to use `rsync`. Let's start with this simple `rsync` statement:

```
rsync myscript.bash yogi:/scripts/
```

This would transfer the file `myscript.bash` from the current directory on the local machine to the `/scripts` directory on the `yogi` server. Now, if the `myscript.bash` file already exists on the server `yogi`, in the `/scripts` directory, then the *rsync remote-update protocol* is used to update the file by sending only the differences between the source and destination files:

```
rsync -avz yogi:/scripts/data /scripts/tmp
```

This `rsync` statement will recursively transfer all the files and directories in the `/scripts/data` directory on the `yogi` machine to the `/scripts/tmp` directory on the local machine. The `-a` `rsync` switch specifies *archive mode*, which preserves the permissions, ownerships, symbolic links, attributes, and so on, and specifies a recursive copy in the transfer. The `-z` `rsync` switch specifies compression is to be used in the transfer to reduce the amount of data in the transfer. Note that this example will create a new directory on the local machine `/scripts/tmp/data`. The `-v` `rsync` switch specifies verbose mode.

```
rsync -avz yogi:/scripts/data/ /scripts/tmp
```

Notice the trailing slash on the source: `yogi:/scripts/data/`

52 Part I ■ The Basics of Shell Scripting

This trailing slash changes the behavior of `rsync` to eliminate creating the additional directory on the destination, as the previous example produced: `/scripts/tmp/data`. The trailing slash on the *source* side tells `rsync` to *copy the directory contents*, as opposed to *copy the directory name and all of its contents*.

```
rsync -avz /scripts/data /scripts/tmp
rsync -avz /scripts/data/ /scripts/tmp
```

As you can see by these two examples, we can copy files locally as well as remotely. Notice that local file copying does not have a hostname specified by the `hostname:` notation.

Simple Generic `rsync` Shell Script

A simple generic shell script for `rsync` consists only of defining some variables to point to the file/directory we want to copy, and a one-line `rsync` statement. Check out Listing 1-22 and we will cover the details at the end.

```
#!/bin/bash
#
# SCRIPT: generic_rsync.bash
# AUTHOR: Randy Michael
# DATE: 11/18/2007
# REV: 1.0
# PURPOSE: This is a generic shell script to copy files
#          using rsync.
#
# set -n # Uncomment to check script syntax without execution
# set -x # Uncomment to debug this script
#
# REV LIST:
#
#
#####
# DEFINE FILES AND VARIABLES HERE
#####

# Define the source and destination files/directories

SOURCE_FL="/scripts/"
DESTIN_FL="booboo:/scripts"

#####
# BEGINNING OF MAIN
#####

# Start the rsync copy
```

Listing 1-22 generic_rsync.bash script

Chapter 1 ■ Scripting Quick Start and Review 53

```
rsync -avz "$SOURCE_FL" "$DESTIN_FL"

# End of generic_rsync.bash
```

Listing 1-22 (continued)

As you can see, there is not much to this shell script. We define the source and destination files/directories to the `SOURCE_FL` and `DESTIN_FL` variables, respectively. Next we use these variables in our `rsync` statement:

```
rsync -avz "$SOURCE_FL" "$DESTIN_FL"
```

This `rsync` command will recursively transfer all the files and subdirectories in the local `/scripts/` directory (notice the trailing slash on the source) to the `/scripts` directory on the `booboo` server using compression to reduce the amount of data transferred. Notice that the trailing slash avoided creating a second `scripts` directory on the destination: `/scripts/scripts/`.

Capturing a List of Files Larger than \$MEG

Who filled up that filesystem? If you want to look quickly for large files, use the following syntax:

```
# Search for files > $MEG_BYTES starting at the $SEARCH_PATH
#
HOLD_FILE=/tmp/largefiles.list
MEG_BYTES=$1
SEARCH_PATH=$(pwd) # Use the current directory

find $SEARCH_PATH -type f -size +${MEG_BYTES}000000c -print > $HOLDFILE
```

Note that in the `find` command after the `-size` parameter there is a plus sign (+) preceding the file size, and there is a `c` added as a suffix. This combination specifies files larger than `$MEG_BYTES` measured in bytes, as opposed to blocks.

Capturing a User's Keystrokes

In most large shops there is a need, at least occasionally, to monitor a user's actions. You may even want to audit the keystrokes of anyone with `root` access to the system or other administration-type accounts, such as `oracle`. Contractors onsite can pose a particular security risk. Typically when a new application comes into the environment, one or two contractors are onsite for a period of time for installation, troubleshooting, and training personnel on the product.

The code shown in Listing 1-23 uses the `script` command to capture all the keystrokes.

54 Part I ■ The Basics of Shell Scripting

```

TS=$(date +%m%d%y%H%M%S)          # File time stamp
THISHOST=$(hostname|cut -f1-2 -d.) # Host name of this machine
LOGDIR=/usr/local/logs/script      # Directory to hold the logs
LOGFILE=${THISHOST}.${LOGNAME}.${TS} # Creates the name of the log file
touch $LOGDIR/$LOGFILE             # Creates the actual file

# Set the command prompt
export PS1="[$LOGNAME:$THISHOST]@''$PWD> "

##### RUN IT HERE #####

chown $LOGNAME ${LOGDIR}/${LOGFILE} # Let the user own the log file
                                     # while the script executes
chmod 600 ${LOGDIR}/${LOGFILE}      # Change permission to RW for the owner

script ${LOGDIR}/${LOGFILE}          # Start the script monitoring session

chown root ${LOGDIR}/${LOGFILE}     # Change the ownership to root
chmod 400 ${LOGDIR}/${LOGFILE}     # Set permission to read-only by root

```

Listing 1-23 Capturing a user's keystrokes

Using the bc Utility for Floating-Point Math

On UNIX machines there is a utility called `bc` that is an interpreter for arbitrary-precision arithmetic language. The `bc` command is an interactive program that provides arbitrary-precision arithmetic. You can start an interactive `bc` session by typing `bc` on the command line. Once in the session you can enter most complex arithmetic expressions as you would in a calculator.

The code segment shown in Listing 1-24 creates the mathematical expression for the `bc` utility and then uses a here document to load the expression into `bc`.

```

# Loop through each number and build a math statement that
# will add all of the numbers together.

for X in $NUM_LIST
do
    ADD="$ADD $PLUS $X"
    PLUS="+"
done

#####

# Do the math here by using a here document to supply

```

Listing 1-24 Example of using `bc` in a shell script

Chapter 1 ■ Scripting Quick Start and Review 55

```
# input to the bc command. The sum of the numbers is
# assigned to the SUM variable.

SUM=$(bc <<EOF
scale=$SCALE
({ADD})
EOF)
```

Listing 1-24 (continued)

This is about as simple as `bc` gets. This is just a taste. Look for more later in the book.

Number Base Conversions

There are a lot of occasions when we need to convert numbers between bases. The code that follows shows some examples of how to change the base.

Using the `typeset` Command

Using the `typeset` command is valid up to base 36.

```
Convert a base 10 number to base 16

# typeset -i16 BASE_16_NUM
# BASE_16_NUM=47295
# echo $BASE_16_NUM

16#b8bf

Convert a base 8 number to base 16

[root@yogi:/scripts]> typeset -i16 BASE_16_NUM
[root@yogi:/scripts]> BASE_16_NUM=8#472521
[root@yogi:/scripts]> echo $BASE_16_NUM

16#735c9
```

Using the `printf` Command

We can use the `printf` command to convert base-10 numbers to octal and hexadecimal notation, as shown here:

```
Convert a base 10 number to base 8

# printf %o 20398

47656
```

56 Part I ■ The Basics of Shell Scripting

Convert a base 10 number to base 16

```
# printf %x 20398
```

```
4fae
```

We can display a number in exponential notation with the `printf` command with the following syntax:

```
# printf %e 20398
```

```
2.039800e+04
```

Create a Menu with the `select` Command

There are many times when you just need to provide a menu for the end user to select from, and this is where a `select` statement comes in. The menu prompt is assigned to the PS3 system variable, and the `select` statement is used a lot like a `for` loop. A `case` statement is used to specify the action to take on each selection.

```
#!/bin/bash
#
# SCRIPT: select_system_info_menu.bash
# AUTHOR: Randy Michael
# DATE: 1/17/2008
# REV: 1.0
#
# PURPOSE: This shell script uses the shell's select
# command to create a menu to show system information

# Clear the screen
clear

# Display the menu title header
echo -e "\n\tSYSTEM INFORMATION MENU\n"

# Define the menu prompt

PS3="Select an option and press Enter: "

# The select command defines what the menu
# will look like

select i in OS Host Filesystems Date Users Quit
do
    case $i in
```

Chapter 1 ■ Scripting Quick Start and Review 57

```
OS)      echo
         uname
         ;;
Host)    echo
         hostname
         ;;
Filesystems)
         echo
         df -k | more
         ;;
Date)    echo
         date
         ;;
Users)   echo
         who
         ;;
Quit)    break
         ;;
esac

# Setting the select command's REPLY variable
# to NULL causes the menu to be redisplayed

REPLY=

# Pause before redisplaying the menu

echo -e "\nPress Enter to Continue...\c"
read

# Ready to redisplay the menu again

# clear the screen

clear

# Display the menu title header

echo -e "\n\tSYSTEM INFORMATION MENU\n"

done

# Clear the screen before exiting

clear
```

Notice in this code segment the use of the `select` statement. This looks just like a `for` loop with a list of possible values. Next is an embedded `case` statement that allows us to specify the action to take when each selection is made.

58 Part I ■ The Basics of Shell Scripting

The output of this simple menu is shown here:

```
SYSTEM INFORMATION MENU

1) OS              3) Filesystems  5) Users
2) Host            4) Date          6) Quit
Select an option and press Enter: 1

Linux

Press Enter to Continue...
```

Removing Repeated Lines in a File

The `uniq` command is used to report and remove repeated lines in a file. This is a valuable tool for a lot of scripting and testing.

If you have a file that has repeated lines named `my_list` and you want to save the list without the repeated lines in a file called `my_list_no_repeats`, use the following command:

```
# uniq my_list my_list_no_repeats
```

If you want to see a file's output without repeated lines, use the following command:

```
# cat repeat_file | uniq
```

Removing Blank Lines from a File

The easiest way to remove blank lines from a file is to use a `sed` statement. The following syntax removes the blank lines:

```
# cat my_file | sed /^$/d
```

or

```
# sed /^$/d my_file
```

Testing for a Null Variable

Variables that have nothing assigned to them are sometimes hard to deal with. The following test will ensure that a variable is either Null or has a value assigned to it. The double quotes are very important and must be used!

Chapter 1 ■ Scripting Quick Start and Review 59

```

VAL= # Creates a NULL variable

if [[ -z "$VAL" && "$VAL" = '' ]]
then
    echo "The VAL variable is NULL"
fi

```

or

```

VAL=25

if [[ ! -z "$VAL" && "$VAL" != '' ]]
then
    echo "The VAL variable is NOT NULL"
fi

```

Directly Access the Value of the Last Positional Parameter, \$#

To access the value of the `$#` positional parameter directly, use the following regular expression:

```
eval '$'$#
```

or

```
eval \$$#
```

There are a lot of uses for this technique, as you will see later in this book.

Remove the Column Headings in a Command Output

There are many instances when we want to get rid of the column headings in a command's output. A lot of people try to use `grep -v` to pattern-match on something unique in the heading. A much easier and more reliable method is to use the `tail` command. An example is shown here with the `df` command output:

```

[root:yogi]@/scripts# df -k

```

Filesystem	1024-blocks	Free	%Used	Iused	%Iused	Mounted on
/dev/hd4	32768	15796	52%	1927	12%	/
/dev/hd2	1466368	62568	96%	44801	13%	/usr
/dev/hd9var	53248	8112	85%	1027	8%	/var
/dev/hd3	106496	68996	36%	245	1%	/tmp
/dev/hd1	4096	3892	5%	55	6%	/home

60 Part I ■ The Basics of Shell Scripting

```

/proc          -          -          -          -          /proc
/dev/hd10opt    655360    16420    98%    16261    10% /opt
/dev/scripts_lv  102400    24012    77%      1137     5% /scripts
/dev/lv_temp    409600    147452   65%      29      1% /tmpfs

```

Now look at the same output with the column headings removed:

```

[root:yogi]@/scripts# df -k | tail +2
/dev/hd4          32768    15796    52%     1927    12% /
/dev/hd2         1466368    62568    96%    44801    13% /usr
/dev/hd9var       53248      8112    85%     1027     8% /var
/dev/hd3         106496    68996    36%      245     1% /tmp
/dev/hd1          4096      3892     5%       55     6% /home
/proc             -          -          -          -          /proc
/dev/hd10opt      655360    16420    98%    16261    10% /opt
/dev/scripts_lv   102400    24012    77%     1137     5% /scripts
/dev/lv_temp      409600    147452   65%      29      1% /tmpfs

```

Just remember to add one to the total number of lines that you want to remove.

Arrays

The shell supports one-dimensional arrays. The maximum number of array elements is 1,024. When an array is defined, it is automatically dimensioned to 1,024 elements. A one-dimensional array contains a sequence of *array elements*, which are like the boxcars connected together on a train track. An array element can be just about anything, except for another array. I know; you're thinking that you can use an array to access an array to create two- and three-dimensional arrays. This can be done, but it is beyond the scope of this book.

Loading an Array

An array can be loaded in two ways. You can define and load the array in one step with the `set -A` command, or you can load the array one element at a time. Both techniques are shown here:

```
set -A MY_ARRAY alpha beta gamma
```

or

```

X=0 # Initialize counter to zero.
# Load the array with the strings alpha, beta, and gamma
for ELEMENT in alpha gamma beta
do
    MY_ARRAY[$X]=$ELEMENT
    ((X = X + 1))
done

```

Chapter 1 ■ Scripting Quick Start and Review 61

The first array element is referenced by 0, not 1. To access array elements use the following syntax:

```
echo ${MY_ARRAY[2]} # Show the third array element
gamma

echo ${MY_ARRAY[*]} # Show all array elements
alpha beta gamma

echo ${MY_ARRAY[@]} # Show all array elements
alpha beta gamma

echo ${#MY_ARRAY[*]} # Show the total number of array elements
3

echo ${#MY_ARRAY[@]} # Show the total number of array elements
3

echo ${MY_ARRAY}      # Show array element 0 (the first element)
alpha
```

We will use arrays in shell scripts in several chapters in this book.

Testing a String

One of the hardest things to do in a shell script is to test the user's input from the command line. The shell script shown in Listing 1-25 will do the trick using regular expressions to define the string composition.

```
#!/bin/ksh
#
# SCRIPT: test_string.ksh
# AUTHOR: Randy Michael
# REV: 1.0.D - Used for development
# DATE: 10/15/2007
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This script is used to test a character
#          string, or variable, for its composition.
#          Examples include numeric, lowercase or uppercase
#          characters, alpha-numeric characters, and IP address.
#
# REV LIST:
#
# set -x # Uncomment to debug this script
```

Listing 1-25 test_string.ksh shell script

62 Part I ■ The Basics of Shell Scripting

```

# set -n # Uncomment to verify syntax without any execution.
#       # REMEMBER: Put the comment back or the script will
#       # NOT EXECUTE!
#
#####
##### DEFINE FUNCTIONS HERE #####
#####

test_string ()
{
# This function tests a character string

# Must have one argument ($1)

if (( $# != 1 ))
then
    # This error would be a programming error

    print "ERROR: $(basename $0) requires one argument"
    return 1
fi
# Assign arg1 to the variable --> STRING

STRING=$1

# This is where the string test begins

case $STRING in

+([0-9]).+([0-9]).+([0-9]).+([0-9]))
    # Testing for an IP address - valid and invalid
    INVALID=FALSE

    # Separate the integer portions of the "IP" address
    # and test to ensure that nothing is greater than 255
    # or it is an invalid IP address.

    for i in $(echo $STRING | awk -F . '{print $1, $2, $3, $4}')
    do
        if (( i > 255 ))
        then
            INVALID=TRUE
        fi
    done

    case $INVALID in
    TRUE) print 'INVALID_IP_ADDRESS'
        ;;

```

Listing 1-25 (*continued*)

Chapter 1 ■ Scripting Quick Start and Review 63

```

FALSE) print 'VALID_IP_ADDRESS'
    ;;
esac
;;
+([0-1])) # Testing for 0-1 only
print 'BINARY_OR_POSITIVE_INTEGER'
;;
+([0-7])) # Testing for 0-7 only
print 'OCTAL_OR_POSITIVE_INTEGER'
;;
+([0-9])) # Check for an integer
print 'INTEGER'
;;
+([-0-9])) # Check for a negative whole number
print 'NEGATIVE_WHOLE_NUMBER'
;;
+([0-9]|[.][0-9]))
# Check for a positive floating point number
print 'POSITIVE_FLOATING_POINT'
;;
+([+0-9]|[.][0-9]))
# Check for a positive floating point number
# with a + prefix
print 'POSITIVE_FLOATING_POINT'
;;
+([-0-9]|[.][0-9]))
# Check for a negative floating point number
print 'NEGATIVE_FLOATING_POINT'
;;
+([-0-9]))
# Check for a negative floating point number
print 'NEGATIVE_FLOATING_POINT'
;;
+([+0-9]))
# Check for a positive floating point number
print 'POSITIVE_FLOATING_POINT'
;;
+([a-f])) # Test for hexadecimal or all lowercase characters
print 'HEXIDECIMAL_OR_ALL_LOWERCASE'
;;
+([a-f]|[0-9])) # Test for hexadecimal or all lowercase characters
print 'HEXIDECIMAL_OR_ALL_LOWERCASE_ALPHANUMERIC'
;;
+([A-F])) # Test for hexadecimal or all uppercase characters
print 'HEXIDECIMAL_OR_ALL_UPPERCASE'
;;
+([A-F]|[0-9])) # Test for hexadecimal or all uppercase characters
print 'HEXIDECIMAL_OR_ALL_UPPERCASE_ALPHANUMERIC'

```

Listing 1-25 (continued)

64 Part I ■ The Basics of Shell Scripting

```

;;
+([a-f]|[A-F]))
# Testing for hexadecimal or mixed-case characters
print 'HEXIDECIMAL_OR_MIXED_CASE'
;;
+([a-f]|[A-F]|[0-9]))
# Testing for hexadecimal/alpha-numeric strings only
print 'HEXIDECIMAL_OR_MIXED_CASE_ALPHANUMERIC'
;;
+([a-z]|[A-Z]|[0-9]))
# Testing for any alpha-numeric string only
print 'ALPHA-NUMERIC'
;;
+([a-z])) # Testing for all lowercase characters only
print 'ALL_LOWERCASE'
;;
+([A-Z])) # Testing for all uppercase numbers only
print 'ALL_UPPERCASE'
;;
+([a-z]|[A-Z]))
# Testing for mixed case alpha strings only
print 'MIXED_CASE'
;;
*) # None of the tests matched the string composition
print 'INVALID_STRING_COMPOSITION'
;;

esac
}

#####

usage ()
{
echo "\nERROR: Please supply one character string or variable\n"
echo "USAGE: $THIS_SCRIPT {character string or variable}\n"
}

#####
##### BEGINNING OF MAIN #####
#####

# Query the system for the name of this shell script.
# This is used for the "usage" function.

THIS_SCRIPT=$(basename $0)

# Check for exactly one command-line argument

```

Listing 1-25 (continued)

Chapter 1 ■ Scripting Quick Start and Review 65

```
if (( $# != 1 ))
then
    usage
    exit 1
fi

# Everything looks okay if we got here. Assign the
# single command-line argument to the variable "STRING"

STRING=$1

# Call the "test_string" function to test the composition
# of the character string stored in the $STRING variable.

test_string $STRING

# End of script
```

Listing 1-25 *(continued)*

This is a good start, but this shell script does not cover everything. Play around with it to see if you can make some improvements.

NOTE Bash shell does not support the `+( [0-9] )`-type regular expressions.

Summary

This chapter is just a primer to get you started with a quick review and some little tricks and tips. In the next 27 chapters, we are going to write a lot of shell scripts to solve some real-world problems. Sit back and get ready to take on the UNIX world!

The first thing that we are going to study is 24 ways to process a file line-by-line. I have seen a lot of good and bad techniques for processing a file line-by-line over the past 15 years, and some have been rather inventive. The next chapter presents the 24 techniques that I have seen the most; at the end of the chapter there is a shell script that times each technique to find the fastest. Read on, and find out which one wins the race. See you in the next chapter!

