

# Working with Alternative Shells

## IN THIS CHAPTER

Understanding the dash shell

Programming in the dash shell

Introducing the zsh shell

Writing scripts for zsh

**A**lthough the bash shell is the most widely used shell in Linux distributions, it isn't the only one. Now that you've seen the standard Linux bash shell and what you can do with it, it's time to examine a few other shells available in the Linux world. This chapter describes two other shells that you may run into in your Linux journey and how they differ from the bash shell.

## What Is the dash Shell?

The Debian dash shell has had an interesting past. It's a direct descendant of the ash shell, a simple copy of the original Bourne shell available on Unix systems (see Chapter 1). Kenneth Almquist created a small-scale version of the Bourne shell for Unix systems and called it the Almquist shell, which was then shortened to *ash*. This original version of the ash shell was extremely small and fast but without many advanced features, such as command line editing or history features, making it difficult to use as an interactive shell.

The NetBSD Unix operating system adopted the ash shell and still uses it today as the default shell. The NetBSD developers customized the ash shell by adding several new features, making it closer to the Bourne shell. The new features include command line editing using both emacs and vi editor commands, as well as a history command to recall previously entered commands. This version of the ash shell is also used by the FreeBSD operating system as the default login shell.

The Debian Linux distribution created its own version of the ash shell (called Debian ash, or *dash*) for inclusion in its version of Linux. For the most part, dash copies the features of the NetBSD version of the ash shell, providing the advanced command line editing capabilities.

However, to add to the shell confusion, the dash shell is actually not the default shell in many Debian-based Linux distributions. Because of the popularity of the bash shell in Linux, most

Debian-based Linux distributions use the bash shell as the normal login shell and use the dash shell only as a quick-start shell for the installation script to install the distribution files.

The exception is the popular Ubuntu distribution. This often confuses shell script programmers and causes a great number of problems with running shell scripts in a Linux environment. The Ubuntu Linux distribution uses the bash shell as the default interactive shell, but uses the dash shell as the default `/bin/sh` shell. This “feature” really confuses shell script programmers.

As you saw in Chapter 11, every shell script must start with a line that declares the shell used for the script. In our bash shell scripts, we’ve been using this:

```
#!/bin/bash
```

This tells the shell to use the shell program located at `/bin/bash` to execute the script. In the Unix world, the default shell was always `/bin/sh`. Many shell script programmers familiar with the Unix environment copy this into their Linux shell scripts:

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

On most Linux distributions, the `/bin/sh` file is a symbolic link (see Chapter 3) to the `/bin/bash` shell program. This allows you to easily port shell scripts designed for the Unix Bourne shell to the Linux environment without having to modify them.

Unfortunately, the Ubuntu Linux distribution links the `/bin/sh` file to the `/bin/dash` shell program. Because the dash shell contains only a subset of the commands available in the original Bourne shell, this can — and often does — cause some shell scripts to not work properly.

The next section walks you through the basics of the dash shell and how it differs from the bash shell. This is especially important to know if you write bash shell scripts that may need to be run in an Ubuntu environment.

## The dash Shell Features

---

Although both the bash shell and the dash shell are modeled after the Bourne shell, they have some differences. This section walks you through the features found in the Debian dash shell to acquaint you with how the dash shell works before we dive into the shell scripting features.

### The dash command line parameters

The dash shell uses command line parameters to control its behavior. Table 23-1 lists the command line parameters and describes what each one does.

**TABLE 23-1 The dash Command Line Parameters**

Parameter	Description
-a	Exports all variables assigned to the shell
-c	Reads commands from a specified command string
-e	If not interactive, exits immediately if any untested command fails
-f	Displays pathname wildcard characters
-n	If not interactive, reads commands but doesn't execute them
-u	Writes an error message to <code>STDERR</code> when attempting to expand a variable that is not set
-v	Writes input to <code>STDERR</code> as it is read
-x	Writes each command to <code>STDERR</code> as it is executed
-I	Ignores EOF characters from the input when in interactive mode
-i	Forces the shell to operate in interactive mode
-m	Turns on job control (enabled by default in interactive mode)
-s	Reads commands from <code>STDIN</code> (the default behavior if no file arguments are present)
-E	Enables the emacs command line editor
-V	Enables the vi command line editor

Debian added a few additional command line parameters to the original ash shell command line parameter list. The `-E` and `-V` command line parameters enable the special command line editing features of the dash shell.

The `-E` command line parameter allows you to use the emacs editor commands for editing command line text (see Chapter 10). You can use all the emacs commands for manipulating text on a single line using the `Ctrl` and `Meta` key combinations.

The `-V` command line parameter allows you to use the vi editor commands for editing command line text (again, see Chapter 10). This feature allows you to switch between normal mode and vi editor mode on the command line by using the `Esc` key. When you're in vi editor mode, you can use all the standard vi editor commands (such as `x` to delete a character, and `i` to insert text). After you finish editing the command line, you must press the `Esc` key again to exit vi editor mode.

### The dash environment variables

The dash shell uses quite a few default environment variables to track information, and you can create your own environment variables as well. This section describes the environment variables and how dash handles them.

### Default environment variables

The dash environment variables are very similar to the environment variables used in bash (see Chapter 6). This is not by accident. Remember that both the dash and bash shells are extensions of the Bourne shell, so they both incorporate many of its features. However, because of its goal of simplicity, the dash shell contains significantly fewer environment variables than the bash shell. You need to take this into consideration when creating shell scripts in a dash shell environment.

The dash shell uses the `set` command to display environment variables:

```
$set
COLORTERM=' '
DESKTOP_SESSION='default'
DISPLAY=':0.0'
DM_CONTROL='/var/run/xdmctl'
GS_LIB='/home/atest/.fonts'
HOME='/home/atest'
IFS='
'
KDEROOTHOME='/root/.kde'
KDE_FULL_SESSION='true'
KDE_MULTHEAD='false'
KONSOLE_DCOP='DCOPRef(konsole-5293,konsole) '
KONSOLE_DCOP_SESSION='DCOPRef(konsole-5293,session-1) '
LANG='en_US'
LANGUAGE='en'
LC_ALL='en_US'
LOGNAME='atest'
OPTIND='1'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
PPID='5293'
PS1='$ '
PS2='> '
PS4='+ '
PWD='/home/atest'
SESSION_MANAGER='local/testbox:/tmp/.ICE-unix/5051'
SHELL='/bin/dash'
SHLVL='1'
TERM='xterm'
USER='atest'
XCURSOR_THEME='default'
_='ash'
$
```

Your default dash shell environment will most likely differ, because different Linux distributions assign different default environment variables at login.

### Positional parameters

In addition to the default environment variables, the dash shell also assigns special variables to any parameters defined in the command line. Here are the positional parameter variables available for use in the dash shell:

- `$0`: The name of the shell
- `$n`: The *n*th position parameter
- `$*`: A single value with the contents of all the parameters, separated by the first character in the IFS environment variable, or a space if IFS isn't defined
- `$@`: Expands to multiple arguments consisting of all the command line parameters
- `$#`: The number of positional parameters
- `$?`: The exit status of the most recent command
- `$-`: The current option flags
- `$$`: The process ID (PID) of the current shell
- `#!`: The process ID (PID) of the most recent background command

All the dash positional parameters mimic the same positional parameters available in the bash shell. You can use each of the positional parameters in your shell scripts just as you would in the bash shell.

### User-defined environment variables

The dash shell also allows you to set your own environment variables. As with bash, you can define a new environment variable on the command line by using the assignment statement:

```
$ testing=10 ; export testing
$ echo $testing
10
$
```

Without the `export` command, user-defined environment variables are visible only in the current shell or process.

### CAUTION

There's one huge difference between dash variables and bash variables. The dash shell doesn't support variable arrays. This small feature causes all sorts of problems for advanced shell script writers.

## The dash built-in commands

Just as with the bash shell, the dash shell contains a set of built-in commands that it recognizes. You can use these commands directly from the command line interface, or you can incorporate them in your shell scripts. Table 23-2 lists the dash shell built-in commands.

**TABLE 23-2 The dash Shell Built-In Commands**

Command	Description
<code>alias</code>	Creates an alias string to represent a text string
<code>bg</code>	Continues specified job in background mode
<code>cd</code>	Switches to the specified directory
<code>echo</code>	Displays a text string and environment variables
<code>eval</code>	Concatenates all arguments with a space
<code>exec</code>	Replaces the shell process with the specified command
<code>exit</code>	Terminates the shell process
<code>export</code>	Exports the specified environment variable for use in all child shells
<code>fg</code>	Continues specified job in foreground mode
<code>getopts</code>	Obtains options and arguments from a list of parameters
<code>hash</code>	Maintains and retrieves a hash table of recent commands and their locations
<code>pwd</code>	Displays the value of the current working directory
<code>read</code>	Reads a line from <code>STDIN</code> and assign the value to a variable
<code>readonly</code>	Reads a line from <code>STDIN</code> to a variable that can't be changed
<code>printf</code>	Displays text and variables using a formatted string
<code>set</code>	Lists or sets option flags and environment variables
<code>shift</code>	Shifts the positional parameters a specified number of times
<code>test</code>	Evaluates an expression and returns 0 if true or 1 if false
<code>times</code>	Displays the accumulated user and system times for the shell and all shell processes
<code>trap</code>	Parses and executes an action when the shell receives a specified signal
<code>type</code>	Interprets the specified name and displays the resolution (alias, built-in, command, keyword)
<code>ulimit</code>	Queries or sets limits on processes
<code>umask</code>	Sets the value of the default file and directory permissions
<code>unalias</code>	Removes the specified alias
<code>unset</code>	Removes the specified variable or option flag from the exported variables
<code>wait</code>	Waits for the specified job to complete and returns the exit status

You probably recognize all these built-in commands from the bash shell. The dash shell supports many of the same built-in commands as the bash shell. You'll notice that there are no commands for the command history file or for the directory stack. The dash shell doesn't support these features.

## Scripting in dash

Unfortunately, the dash shell doesn't recognize all the scripting features of the bash shell. Shell scripts written for the bash environment often fail when run in the dash shell, causing all sorts of grief for shell script programmers. This section describes the differences you'll need to be aware of to get your shell scripts to run properly in a dash shell environment.

### Creating dash scripts

You probably guessed by now that creating shell scripts for the dash shell is pretty similar to creating shell scripts for the bash shell. You should always specify which shell you want to use in your script to ensure that the script runs with the proper shell.

You do this on the first line of the shell:

```
#!/bin/dash
```

You can also specify a shell command line parameter on this line, as was documented earlier in "The dash command line parameters" section.

### Things that don't work

Unfortunately, because the dash shell is only a subset of the Bourne shell features, some things in bash shell scripts don't work in the dash shell. These are often called *bashisms*. This section is a quick summary of bash shell features you may be used to using in your bash shell scripts that don't work if you're in a dash shell environment.

#### Using arithmetic

Chapter 11 showed three ways to express a mathematical operation in the bash shell script:

- **Using the `expr` command:** `expr operation`
- **Using square brackets:** `$( operation )`
- **Using double parentheses:** `$(( operation ))`

The dash shell supports the `expr` command and the double parentheses method but doesn't support the square bracket method. This can be a problem if you have lots of mathematical operations that use the square brackets.

The proper format for performing mathematical operations in dash shell scripts is to use the double parentheses method:

```
$ cat test5b
#!/bin/dash
# testing mathematical operations

value1=10
value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
$ ./test5b
The answer is 150
$
```

Now the shell can perform the calculation properly.

### The test command

Although the dash shell supports the `test` command, you must be careful how you use it. The bash shell version of the `test` command is slightly different from the dash shell version.

The bash shell `test` command allows you to use the double equal sign (`==`) to test if two strings are equal. This is an add-on to accommodate programmers familiar with using this format in other programming languages.

However, the `test` command available in the dash shell doesn't recognize the `==` symbol for text comparisons. Instead, it only recognizes the `=` symbol. If you use the `==` symbol in your bash scripts, you need to change the text comparison symbol to just a single equal sign:

```
$ cat test7
#!/bin/dash
# testing the = comparison

test1=abcdef
test2=abcdef

if [ $test1 = $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test7
They're the same!
$
```



This little bashism is responsible for many hours of frustration for shell programmers!

### The function Command

Chapter 17 showed you how to define your own functions in your shell scripts. The bash shell supports two methods for defining functions:

- Using the `function()` statement
- Using the function name only

The dash shell doesn't support the `function` statement. Instead, in the dash shell you must define a function using the function name with parentheses.

If you're writing shell scripts that may be used in the dash environment, always define functions using the function name and not the `function()` statement:

```
$ cat test10
#!/bin/dash
# testing functions

func1() {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test10
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
This is the end of the script
$
```

Now the dash shell recognizes the function defined in the script just fine and uses it within the script.

# The zsh Shell

---

Another popular shell that you may run into is the Z shell (called zsh). The zsh shell is an open source Unix shell developed by Paul Falstad. It takes ideas from all the existing shells and adds many unique features to create a full-blown advanced shell designed for programmers.

The following are some of the features that make the zsh shell unique:

- Improved shell option handling
- Shell compatibility modes
- Loadable modules

Of all these features, a loadable module is the most advanced feature in shell design. As you've seen in the bash and dash shells, each shell contains a set of built-in commands that are available without the need for external utility programs. The benefit of built-in commands is execution speed. The shell doesn't have to load a utility program into memory before running it; the built-in commands are already in the shell memory, ready to go.

The zsh shell provides a core set of built-in commands, plus the capability to add more *command modules*. Each command module provides a set of additional built-in commands for specific circumstances, such as network support and advanced math functions. You can add only the modules you think you need for your specific situation.

This feature provides a great way to either limit the size of the zsh shell for situations that require a small shell size and few commands or expand the number of available built-in commands for situations that require faster execution speeds.

## Parts of the zsh Shell

---

This section walks you through the basics of the zsh shell, showing the built-in commands that are available (or can be added by installing modules), as well as the command line parameters and environment variables used by the zsh shell.

### Shell options

Most shells use command line parameters to define the behavior of the shell. The zsh shell uses a few command line parameters to define the operation of the shell, but mostly it uses *options* to customize the behavior of the shell. You can set shell options either on the command line or within the shell itself using the `set` command.

Table 23-3 lists the command line parameters available for the zsh shell.

**TABLE 23-3 The zsh Shell Command Line Parameters**

Parameter	Description
-c	Executes only the specified command and exits
-i	Starts as an interactive shell, providing a command line interface prompt
-s	Forces the shell to read commands from STDIN
-o	Specifies command line options

Although this may seem like a small set of command line parameters, the `-o` parameter is somewhat misleading. It allows you to set shell options that define features within the shell. By far, the zsh shell is the most customizable shell available. You can alter lots of features for your shell environment. The different options fit into several general categories:

- **Changing directories:** Options that control how the `cd` and `dirs` commands handle directory changes
- **Completion:** Options that control command completion features
- **Expansion and globbing:** Options that control file expansion in commands
- **History:** Options that control command history recall
- **Initialization:** Options that control how the shell handles variables and startup files when started
- **Input/output:** Options that control command handling
- **Job control:** Options that dictate how the shell handles and starts jobs
- **Prompting:** Options that define how the shell works with command line prompts
- **Scripts and functions:** Options that control how the shell processes shell scripts and defines shell functions
- **Shell emulation:** Options that allow you to set the behavior of the zsh shell to mimic the behavior of other shell types
- **Shell state:** Options that define what type of shell to start
- **zle:** Options for controlling the zsh line editor (zle) feature
- **Option aliases:** Special options that can be used as aliases for other option names

With this many different categories of shell options, you can imagine just how many actual options the zsh shell supports.

## Built-in commands

The zsh shell is unique in that it allows you to expand the built-in commands available in the shell. This provides for a wealth of speedy utilities at your fingertips for a host of different applications.

This section describes the core built-in commands, along with the various modules available at the time of this writing.

### Core built-in commands

The core of the zsh shell contains the basic built-in commands you're used to seeing in other shells. Table 23-4 describes the built-in commands available for you.

**TABLE 23-4 The zsh Core Built-In Commands**

Command	Description
alias	Defines an alternate name for a command and arguments
autoload	Preloads a shell function into memory for quicker access
bg	Executes a job in background mode
bindkey	Binds keyboard combinations to commands
builtin	Executes the specified built-in command instead of an executable file of the same name
bye	The same as <code>exit</code>
cd	Changes the current working directory
chdir	Changes the current working directory
command	Executes the specified command as an external file instead of a function or built-in command
declare	Sets the data type of a variable (same as <code>typeset</code> )
dirs	Displays the contents of the directory stack
disable	Temporarily disables the specified hash table elements
disown	Removes the specified job from the job table
echo	Displays variables and text
emulate	Sets zsh to emulate another shell, such as the Bourne, Korn, or C shells
enable	Enables the specified hash table elements
eval	Executes the specified command and arguments in the current shell process
exec	Executes the specified command and arguments replacing the current shell process
exit	Exits the shell with the specified exit status. If none specified, uses the exit status of the last command
export	Allows the specified environment variable names and values to be used in child shell processes
false	Returns an exit status of 1

<code>fc</code>	Selects a range of commands from the history list
<code>fg</code>	Executes the specified job in foreground mode
<code>float</code>	Sets the specified variable for use as a floating point variable
<code>functions</code>	Sets the specified name as a function
<code>getln</code>	Reads the next value in the buffer stack and places it in the specified variable
<code>getopts</code>	Retrieves the next valid option in the command line arguments and places it in the specified variable
<code>hash</code>	Directly modifies the contents of the command hash table
<code>history</code>	Lists the commands contained in the history file
<code>integer</code>	Sets the specified variable for use as an integer value
<code>jobs</code>	Lists information about the specified job or all jobs assigned to the shell process
<code>kill</code>	Sends a signal (Default <code>SIGTERM</code> ) to the specified process or job
<code>let</code>	Evaluates a mathematical operation and assigns the result to a variable
<code>limit</code>	Sets or displays resource limits
<code>local</code>	Sets the data features for the specified variable
<code>log</code>	Displays all users currently logged in who are affected by the watch parameter
<code>logout</code>	Same as <code>exit</code> , but works only when the shell is a login shell
<code>popd</code>	Removes the next entry from the directory stack
<code>print</code>	Displays variables and text
<code>printf</code>	Displays variables and text using C-style format strings
<code>pushd</code>	Changes the current working directory and puts the previous directory in the directory stack
<code>pushln</code>	Places the specified arguments into the editing buffer stack
<code>pwd</code>	Displays the full pathname of the current working directory
<code>read</code>	Reads a line and assigns data fields to the specified variables using the <code>IFS</code> characters
<code>readonly</code>	Assigns a value to a variable that can't be changed
<code>rehash</code>	Rebuilds the command hash table
<code>set</code>	Sets options or positional parameters for the shell
<code>setopt</code>	Sets the options for a shell
<code>shift</code>	Reads and deletes the first positional parameter and shifts the remaining ones down one position

*Continues*

**TABLE 23-4** (continued)

Command	Description
source	Finds the specified file and copies its contents into the current location
suspend	Suspends the execution of the shell until it receives a SIGCONT signal
test	Returns an exit status of 0 if the specified condition is TRUE
times	Displays the cumulative user and system times for the shell and processes that run in the shell
trap	Blocks the specified signals from being processed by the shell and executes the specified commands if the signals are received
true	Returns a zero exit status
ttctl	Locks and unlocks the display
type	Displays how the specified command would be interpreted by the shell
typeset	Sets or displays attributes of variables
ulimit	Sets or displays resource limits of the shell or processes running in the shell
umask	Sets or displays the default permissions for creating files and directories
unalias	Removes the specified command alias
unfunction	Removes the specified defined function
unhash	Removes the specified command from the hash table
unlimit	Removes the specified resource limit
unset	Removes the specified variable attribute.
unsetopt	Removes the specified shell option
wait	Waits for the specified job or process to complete
whence	Displays how the specified command would be interpreted by the shell
where	Displays the pathname of the specified command if found by the shell
Which	Displays the pathname of the specified command using csh-style output
zcompile	Compiles the specified function or script for faster autoloading
zmodload	Performs operations on loadable zsh modules

The zsh shell is no slouch when it comes to providing built-in commands! You should recognize most of these commands from their bash counterparts. The most important features of the zsh shell built-in commands are modules.

### Add-in modules

There's a long list of modules that provide additional built-in commands for the zsh shell, and the list continues to grow as resourceful programmers create new modules. Table 23-5 shows some of the more popular modules available.

**TABLE 23-5 The zsh Modules**

Module	Description
zsh/datetime	Additional date and time commands and variables
zsh/files	Commands for basic file handling
zsh/mapfile	Access to external files via associative arrays
zsh/mathfunc	Additional scientific functions
zsh/pcre	The extended regular expression library
zsh/net/socket	Unix domain socket support
zsh/stat	Access to the stat system call to provide system statistics
zsh/system	Interface for various low-level system features
zsh/net/tcp	Access to TCP sockets
zsh/zftp	A specialized FTP client command
zsh/zselect	Blocks and returns when file descriptors are ready
zsh/zutil	Various shell utilities

The zsh shell modules cover a wide range of topics, from providing simple command line editing features to advanced networking functions. The idea behind the zsh shell is to provide a basic minimum shell environment and let you add on the pieces you need to accomplish your programming job.

### Viewing, adding, and removing modules

The `zmodload` command is the interface to the zsh modules. You use this command to view, add, and remove modules from the zsh shell session.

Using the `zmodload` command without any command line parameters displays the currently installed modules in your zsh shell:

```
% zmodload
zsh/zutil
zsh/complete
zsh/main
zsh/terminfo
zsh/zle
zsh/parameter
%
```

Different zsh shell implementations include different modules by default. To add a new module, just specify the module name on the `zmodload` command line:

```
% zmodload zsh/zftp
%
```

Nothing indicates that the module loaded. You can perform another `zmodload` command, and the new module should appear in the list of installed modules.

After you load a module, the commands associated with the module are available as built-in commands:

```
% zftp open myhost.com rich testing1
Welcome to the myhost FTP server.
% zftp cd test
% zftp dir
01-21-11 11:21PM      120823 test1
01-21-11 11:23PM      118432 test2
% zftp get test1 > test1.txt
% zftp close
%
```

The `zftp` command allows you to conduct a complete FTP session directly from your zsh shell command line! You can incorporate these commands into your zsh shell scripts to perform file transfers directly from your scripts.

To remove an installed module, use the `-u` parameter, along with the module name:

```
% zmodload -u zsh/zftp
% zftp
zsh: command not found: zftp
%
```

### NOTE

It's a common practice to place `zmodload` commands in the `$HOME/.zshrc` startup file so your favorite functions load automatically when the zsh shell starts.

## Scripting with zsh

---

The main purpose of the zsh shell was to provide an advanced programming environment for shell programmers. With that in mind, it's no surprise that the zsh shell offers many features that make shell scripting easier.



## Mathematical operations

As you would expect, the zsh shell allows you to perform mathematical functions with ease. In the past, the Korn shell has led the way in supporting mathematical operations by providing support for floating-point numbers. The zsh shell has full support for floating-point numbers in all its mathematical operations!

### Performing calculations

The zsh shell supports two methods for performing mathematical operations:

- The `let` command
- Double parentheses

When you use the `let` command, you should enclose the operation in double quotation marks to allow for spaces:

```
% let value1=" 4 * 5.1 / 3.2 "
% echo $value1
6.3750000000
%
```

Be careful, using floating point numbers may introduce a precision problem. To solve this, it's always a good idea to use the `printf` command and to specify the decimal precision needed to correctly display the answer:

```
% printf "%6.3f\n" $value1
6.375
%
```

Now that's much better!

The second method is to use the double parentheses. This method incorporates two techniques for defining the mathematical operation:

```
% value1=$(( 4 * 5.1 ))
% (( value2 = 4 * 5.1 ))
% printf "%6.3f\n" $value1 $value2
20.400
20.400
%
```

Notice that you can place the double parentheses either around just the operation (preceded by a dollar sign) or around the entire assignment statement. Both methods produce the same results.

If you don't use the `typeset` command to declare the data type of a variable beforehand, the zsh shell attempts to automatically assign the data type. This can be dangerous when working with both integer and floating-point numbers. Look at this example:

```
% value1=10
% value2=$(( $value1 / 3 ))
% echo $value2
3
%
```

Now, that's probably not the answer you want to come out from the calculation. When you specify numbers without decimal places, the zsh shell interprets them as integer values and performs integer calculations. To ensure that the result is a floating-point number, you must specify the numbers with decimal places:

```
% value1=10.
% value2=$(( $value1 / 3. ))
% echo $value2
3.3333333333333335
%
```

Now the result is in the floating-point format.

### Mathematical functions

With the zsh shell, built-in mathematical functions are either feast or famine. The default zsh shell doesn't include any special mathematical function. However, if you install the zsh/mathfunc module, you have more math functions than you'll most likely ever need:

```
% value1=$(( sqrt(9) ))
zsh: unknown function: sqrt
% zmodload zsh/mathfunc
% value1=$(( sqrt(9) ))
% echo $value1
3.
%
```

That was simple! Now you have an entire math library of functions at your fingertips.

### NOTE

Lots of mathematical functions are supported in zsh. For a complete listing of all the math functions that the zsh/mathfunc module provides, look at the manual page for zsh modules.

## Structured commands

The zsh shell provides the usual set of structured commands for your shell scripts:

- if-then-else statements
- for loops (including the C-style)
- while loops

- until loops
- select statements
- case statements

The zsh shell uses the same syntax for each of these structured commands that you're used to from the bash shell. The zsh shell also includes a different structured command called `repeat`. The `repeat` command uses this format:

```
repeat param
do
    commands
done
```

The `param` parameter must be a number or a mathematical operation that evaluates to a number. The `repeat` command then performs the specified commands that number of times:

```
% cat test1
#!/bin/zsh
# using the repeat command

value1=$(( 10 / 2 ))
repeat $value1
do
    echo "This is a test"
done
$ ./test1
This is a test
This is a test
This is a test
This is a test
This is a test
%
```

This command allows you to repeat sections of code for a set number of times based on a calculation.

## Functions

The zsh shell supports the creation of your own functions either using the `function` command or by defining the function name with parentheses:

```
% function functest1 {
> echo "This is the test1 function"
}
% functest2() {
> echo "This is the test2 function"
}
```

```
% functest1
This is the test1 function
% functest2
This is the test2 function
%
```

As with bash shell functions (see Chapter 17), you can define functions within your shell script and then either use global variables or pass parameters to your functions.

## Summary

---

This chapter discussed two popular alternative Linux shells that you may run into. The dash shell was developed as part of the Debian Linux distribution and is mainly found in the Ubuntu Linux distribution. It's a smaller version of the Bourne shell, so it doesn't support as many features as the bash shell, which can cause problems for script writing.

The zsh shell is often found in programming environments, because it provides lots of cool features for shell script programmers. It uses loadable modules to load separate code libraries, which make using advanced functions as easy as running command line commands! There are loadable modules for lots of different functions, from complex mathematical algorithms to network applications such as FTP and HTTP.

The next section of this book dives into some specific scripting applications you might run into in the Linux environment. The next chapter shows how to write simple utilities to help with your day-to-day Linux administration functions. Those can greatly help simplify common tasks you perform on the system.