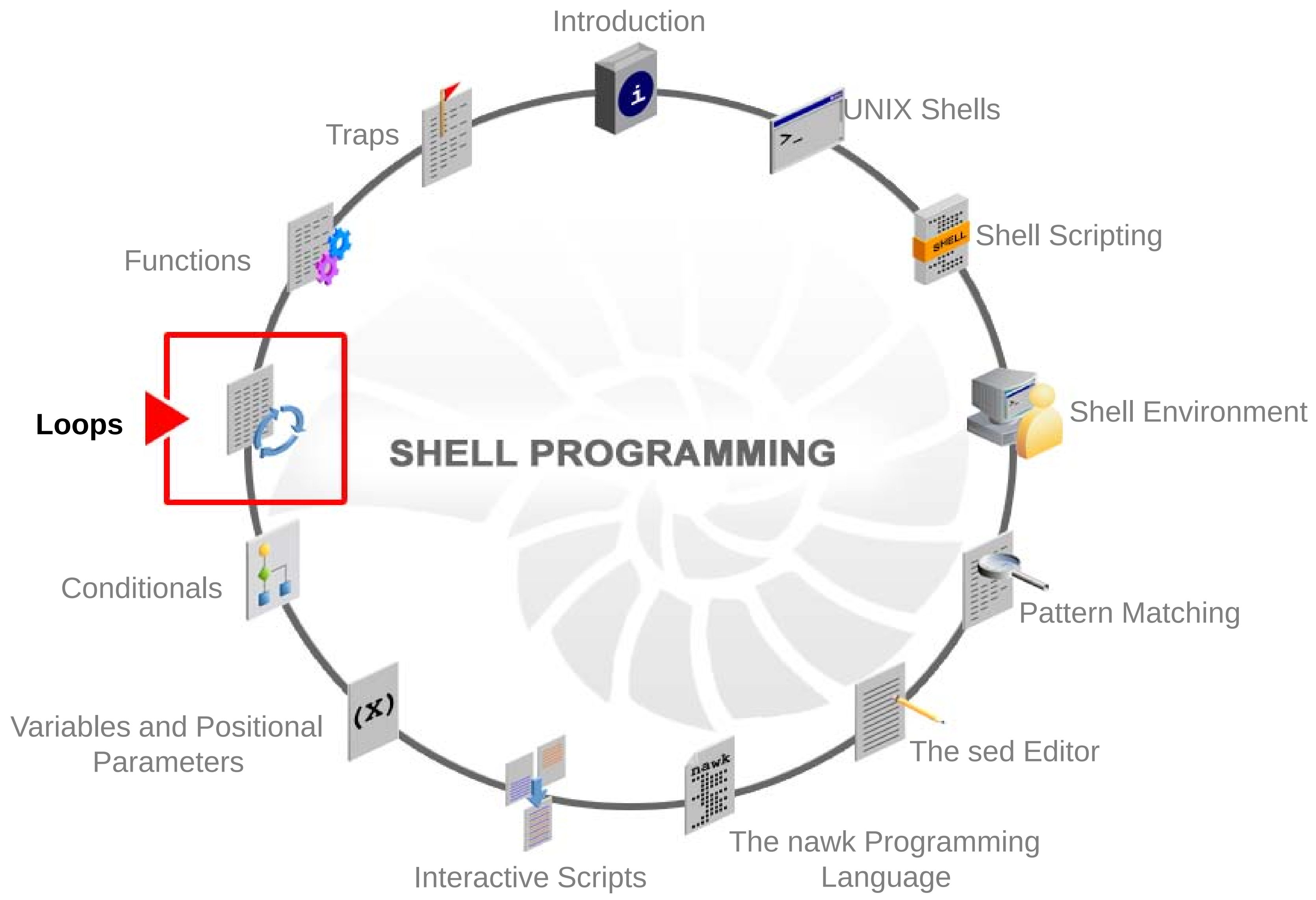
11

Loops


Introduction
UNIX Shells
Shell Scripting
Shell Environment
Pattern Matching
The sed Editor
The nawk Programming Language
Interactive Scripts
Variables and Positional Parameters
Conditionals
Loops
Functions
Traps
SHELL PROGRAMMING

Objectives

After completing this lesson, you should be able to:

- Describe the `for`, `while`, and `until` looping constructs
- Create menus by using the `select` looping statement
- Provide a variable number of arguments to the script by using the `shift` statement
- Parse script options by using the `getopts` statement

Agenda

- Describing the `for`, `while`, and `until` looping constructs
- Creating menus by using the `select` looping statement
- Providing a variable number of arguments to the script by using the `shift` statement
- Parsing script options by using the `getopts` statement

Shell Loops

- Shell loops are useful scripting tools that enable you to execute a set of commands or statements repeatedly either specified times or until some condition is met.
- The shell provides three looping constructs:
  – The `for` loop
  – The `while` loop
  – The `until` loop

The `for` Loop

- The `for` loop repeats a set of commands for every item in an argument list.
- The `for` loop in the shell takes a list of words (strings) as an argument.
- The number of words in the list determines the number of times the statements in the `for` loop are executed.
- Syntax:

```
for var in argument_list ...
do
       statement1
       ...
       statementN
done
```

The `for` Loop Argument List

- The argument list in a `for` loop can be any list of words, strings, or numbers.
- You can generate an argument list by using any of the following methods (or combination of methods):
  - Explicit list
  - Content of a variable
  - Command-line arguments
  - Command substitution
  - File names in command substitution
  - File-name substitution

Using an Explicit List to Specify Arguments

- When arguments are listed for a `for` loop, they are called an explicit list.

```
for var in arg1 arg2 arg3 arg4 ... argn
```

- Example:

```
for fruit in apple orange banana peach kiwi
do
print "Value of fruit is:  $fruit"
done
```

- The output for the example `for` loop is:

```
Value of fruit is:  apple
Value of fruit is:  orange
Value of fruit is:  banana
Value of fruit is:  peach
Value of fruit is:  kiwi
```

Using the Content of a Variable to Specify Arguments

```
$ cat ex1.sh
#!/bin/bash
# Script name: ex1.sh
echo "Enter some text: \c"
read INPUT
for var in $INPUT
do
        echo "var contains: $var"
done

$ ./ex1.sh
Enter some text: I like the Bash shell.
var contains: I
var contains: like
var contains: the
var contains: Bash
var contains: shell.
```

Using Command-Line Arguments to Specify Arguments

```
$ cat ex2.sh
#!/bin/sh
# Script name: ex2.sh
for var in $*
do
        echo "command line contains: $var"
Done

$ ./ex2.sh The Bourne shell is good too.
command line contains: The
command line contains: Bourne
command line contains: shell
command line contains: is
command line contains: good
command line contains: too.
```

ORACLE

Using Command Substitution to Specify Arguments

```
$ cat fruit1
apple
orange
banana
peach
Kiwi
$ cat ex3.sh
#!/bin/bash
# Script name: ex3.sh
for var in $(cat fruit1)
do
        print  "$var"
done
$ ./ex3.sh
apple
orange
banana
peach
kiwi
```

Using File Names in Command Substitution to Specify Arguments

- Some commands provide file names and directory names as their output.
- In the following example, the shell substitutes the output of the command, `ls /etc/p*` (`/etc/passwd /etc/profile` and so on), as the argument list for the `for` loop.

```
$ cat ex7.sh
#!/bin/bash

# Script name: ex.sh

for var in $(ls /etc/p*)
do
        print  "var contains: $var"
done
```

Using File-Name Substitution to Specify Arguments

- Additionally, there is a syntax for specifying just files and directories as the argument list for the `for` loop.

```
for var in file_list
```

- In the following example, the shell substitutes the file names that match `/etc/p*` as the argument list.

```
ls /etc/p*
/etc/passwd /etc/profile /etc/prtvtoc
...
for var in /etc/p*
```

Quiz

The loop, `for var in arg1 arg2 arg3 arg4 ... argn`, is an example of using which of the following methods for generating the argument list in a `for` loop?

a. Content of a variable
b. Command-line arguments
c. Explicit list
d. Command substitution
e. File-name substitution

The `while` Loop

- The `while` loop allows you to repeatedly execute a group of statements while a command executes successfully.
- Syntax:

```
while control_command
do
        statement1
        ...
        statementN
done
```

  – Where:
    – `control_command` can be any command that exits with a success or failure status.
    – The statements in the body of the `while` loop can be any utility commands, user programs, shell scripts, or shell statements.

The `while` Loop Syntaxes

- While the contents of `$var` are equal to “`value`”, the loop continues.

```
    while [ "$var" = "value" ]
    while [[ "$var" == "value" ]]
```

- While the value of `$num` is less than or equal to `10`, the loop continues.

```
    while [ $num -le 10 ]
    while (( num <= 10 ))
```

The while Loop: Example

```
$ cat whiletest1.sh
#!/bin/bash
# Script name: whiletest1.sh
num=5
while [ $num -le 10 ]
do
        echo $num
        num=‘expr $num + 1‘
done

$ cat whiletest1.sh
#!/bin/bash
# Script name: whiletest.sh
num=5
while (( num <= 10 ))
do
        echo $num
        (( num = num + 1 ))     # let num=num+1
done
```

The while Loop: Example

```
$ cat while.sh
#!/bin/bash
# Script name: while.sh
num=1
while (( num < 6 ))
do
        print "The value of num is: $num"
        (( num = num + 1 ))          # let num=num+1
done
print "Done."

$ ./while.sh
Value of n is: 1
Value of n is: 2
Value of n is: 3
Value of n is: 4
Value of n is: 5
Done.
$
```

The `while` Loop: Example

```
$ cat readinput.sh
#!/bin/bash

# Script name: readinput.sh

print -n "Enter a string: “

while read var
do
        print "Keyboard input is: $var"
        print -n "\nEnter a string: “
done

print "End of input.“
```

Redirecting Input for a `while` Loop

```
$ cat phonelist
Claude Rains:214-555-5107
Agnes Moorehead:710-555-6538
Rosalind Russel:710-555-0482
Loretta Young:409-555-9327
James Mason:212-555-2189
$

$ cat internal_redir.sh
#!/bin/bash
# Script name: internal_redir.sh
# set the Internal Field Separator to a colon
IFS=:
while read name number
do
        print "The phone number for $name is $number"
done < phonelist
```

Quiz

The `while` loop allows you to repeatedly execute a group of statements while a command executes successfully.

a. True

b. False

The `until` Loop

- The `until` loop executes as long as the command fails.
- After the command succeeds, the loop exits, and execution of the script continues with the statement following the `done` statement.
- Syntax:

```
until control_command
do
           statement1
           ...
           statementn
done
```

The until Loop: Example

```
$ cat until.sh
#!/bin/bash
# Script name: until.sh
num=1
until (( num == 6 ))
do
        print "The value of num is: $num"
        (( num = num + 1 ))
Done
print "Done.“

$ ./until.sh
The value of num is: 1
The value of num is: 2
The value of num is: 3
The value of num is: 4
The value of num is: 5
Done.
```

The `break` Statement

- The `break` statement allows you to exit the current loop.
- It is often used in an `if` statement that is contained within a `while` loop, with the condition in the `while` loop always evaluating to `true`.
- The `break` statement exits out of the *innermost loop* in which it is contained.

The break Statement: Example

```
$ cat break.sh
#!/bin/bash
# Script name: break.sh

typeset -i  num=0
while true
do
        print -n "Enter any number (0 to exit): "
        read num junk
        if (( num == 0 ))
        then
            break
        else
            print "Square of $num is $(( num * num )). \n"
        fi
print "script has ended"
```

The `continue` Statement

- The `continue` statement forces the shell to skip the statements in the loop below the `continue` statement and return to the top of the loop for the next iteration.
- When `continue` is used in a `for` loop, the variable `var` takes on the value of the next element in the list.
- When `continue` is used in a `while` or an `until` loop, execution resumes with the test of the `control_command` at the top of the loop.

The `continue` Statement: Example

```
$ cat continue.sh
#!/bin/bash
# Script name: continue.sh

typeset -l new
for file in *
do
        print "Working on file $file..."
        if [[ $file != *[A-Z]* ]]
        then
                continue
        fi
        orig=$file
        new=$file
        mv $orig $new
        print "New file name for $orig is $new."
done

print "Done."
```

The continue Statement: Example

```
$ ls test.dir
Als             a               sOrt.dAtA       slAlk
Data.File       recreate_names  scR1            teXtfile

$ ../continue.sh
Working on file Als...
New file name for Als is als.
Working on file Data.File...
New file name for Data.File is data.file.
Working on file a...
Working on file recreate_names...
Working on file sOrt.dAtA...
New file name for sOrt.dAtA is sort.data.
Working on file scR1...
New file name for scR1 is scr1.
Working on file slAlk...
New file name for slAlk is slalk.
Working on file teXtfile...
New file name for teXtfile is textfile.
Done.
```

Quiz

Which of the following statements allows you to exit the current loop?

a. `until`

b. `continue`

c. `exit`

d. `break`

ORACLE

Agenda

- Describing the `for`, `while`, and `until` looping constructs
- Creating menus by using the `select` looping statement
- Providing a variable number of arguments to the script by using the `shift` statement
- Parsing script options by using the `getopts` statement

The `select` Statement

- The `select` statement is used for creating menus.
- Syntax:

```
    select var in list
    do
              statement1
              ...
              statementN
    done
```

The PS3 Reserved Variable

- After displaying the list of menu choices for a user, the shell prints a prompt and waits for user input.
- The prompt value is the value of the PS3 variable.
- You can set this variable to any value.
- Default value is #?

The `select` Loop: Example

```
$ cat menu.sh
#!/bin/bash
# Script name: menu.sh

PS3="Enter the number for your fruit choice: "
select fruit in apple orange banana peach pear
do
        case $fruit in
        apple)
                print "An apple has 80 calories."
                ;;
        orange)
                print "An orange has 65 calories."
                ;;
       banana)
```

Continued…

The `select` Loop: Example

Continued…

```
                        print "A banana has 100 calories."
                         ;;
            peach)
                         print "A peach has 38 calories."
                         ;;
            pear)
                         print "A pear has 100 calories."
                         ;;
            *)
                         print "Please try again. Use ’1’-’5’"
                         ;;
esac

done
```

The `select` Loop: Example

```
$ ./menu.sh
1) apple
2) orange
3) banana
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 7
Please try again. Use ’1’-’5’
Enter the number for your fruit choice: apple
Please try again. Use ’1’-’5’
Enter the number for your fruit choice: 1
An apple has 80 calories.
Enter the number for your fruit choice: ^d
$
```

Exiting the `select` Loop: Example

```
$ cat menu1.sh
#!/bin/bash

# Script name: menu.sh

PS3="Enter the number for your fruit choice: "
select fruit in apple orange banana peach pear "Quit Menu"
do
        case $fruit in
                apple)
                       print "An apple has 80 calories."
                        ;;
                orange)
                      print "An orange has 65 calories."
                        ;;
                banana)
```

Exiting the `select` Loop: Example

```
                        print "A banana has 100 calories."
                        ;;
                peach)
                        print "A peach has 38 calories."
                        ;;
                pear)
                        print "A pear has 100 calories."
                        ;;
                "Quit Menu")
                        break
                        ;;
                *)
                        print "You did not enter a correct
    choice."
                        ;;
        esac
done
```

Exiting the `select` Loop: Example

```
$ ./menu1.sh
1) apple
2) orange
3) Banana
4) peach
5) pear
6) Quit Menu
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 6
$
```

Submenus

- You can include a `select` loop inside another `select` loop to create submenus.
- To exit the submenu loop, include the `break` statement in the action.
- When moving from one menu to another, reset the `PS3` variable to the prompt needed.

Submenus: Example

```
$ cat submenu.sh
#!/bin/bash
main_prompt="Main Menu: What would you like to order? "
dessert_menu="Enter number for dessert choice: "
PS3=$main_prompt
select order in "broasted chicken" "prime rib" stuffed lobster"
dessert "Order Completed“
do
case $order in
"broasted chicken") print 'Broasted chicken with baked potato,
rolls, and salad is $14.95.';;
"prime rib") print 'Prime rib with baked potato, rolls, and
fresh vegetable is $17.95.';;
"stuffed lobster") print 'Stuffed lobster with rice pilaf,
rolls, and salad is $15.95.';;
dessert)
PS3=$dessert_menu
select dessert in "apple pie" "sherbet" "fudge cake"
"carrot cake“
do
```

Submenus: Example

```
case $dessert in
"apple pie") print 'Fresh baked apple pie is $2.95.‘
break;;
"sherbet") print 'Orange sherbet is $1.25.‘
break;;
"fudge cake") print 'Triple layer fudge cake is $3.95.‘
break;;
"carrot cake") print 'Carrot cake is $2.95.‘
break;;
*) print 'Not a dessert choice.';;
esac
done
PS3=$main_prompt;;
"Order Completed") break;;
*) print 'Not a main entree choice.';;
esac
done
print 'Enjoy your meal.’
```

Submenus: Example

```
$  ./submenu.sh

1) broasted chicken
2) prime rib
3) stuffed lobster
4) dessert
5) Order Completed
Main Menu: What would you like to order? 3
Stuffed lobster with rice pilaf, rolls, and salad is $15.95.
Main Menu: What would you like to order? 4
1) apple pie
2) sherbet
3) fudge cake
4) carrot cake
Enter number for dessert choice: 3
Triple layer fudge cake is $3.95.
Main Menu: What would you like to order? 5
Enjoy your meal.
```

Quiz

The default value of the PS3 variable is:

a. ?#

b. .

c. #?

d. #

Agenda

- Describing the `for`, `while`, and `until` looping constructs
- Creating menus by using the `select` looping statement
- Providing a variable number of arguments to the script by using the `shift` statement
- Parsing script options by using the `getopts` statement

The `shift` Statement

- The `shift` statement is used to shift command-line arguments to the left.
- Syntax:

```
shift [ num ]
```

- Example:

```
shift 4
```

Note: If no number is supplied as an argument to the `shift` statement, the number is assumed to be `1`.

The shift Statement: Example

```
$ cat shift.sh
#!/bin/bash
# Script name: shift.sh

USAGE="usage: $0 arg1 arg2 ... argN"
if (( $# == 0 ))
then
        print $USAGE
        exit 1
print "The arguments to the script are:"
while (($#))
do
        print $1
        shift
done
print ’The value of $* is now:’ $*
```

The `shift` Statement: Example

```
$ ./shift.sh one two three four
The arguments to the script are:
one
two
three
four
The value of $* is now:
```

Quiz

Which of the following is the correct syntax of the `shift` statement?

a. `shift [ var ]`

b. `shift [ options ]`

c. `shift [ num ]`

d. `shift`

Agenda

- Describing the `for`, `while`, and `until` looping constructs
- Creating menus by using the `select` looping statement
- Providing a variable number of arguments to the script by using the `shift` statement
- Parsing script options by using the `getopts` statement

The `getopts` Statement

- The `getopts` statement is a built-in command that retrieves options and option arguments from a list of parameters.
- Syntax:

```
getopts options [opt_args] var
```

- Options or switches are single-letter characters preceded by a + or a – sign.
- A – sign means to turn some flag on, whereas a + sign means to turn some flag off.

Using the `getopts` Statement

- The `getopts` statement is most often used as the condition in a `while` loop.
- It is followed by the `case` statement, which is used to specify the actions to be taken for the various options that can appear on the command line.

```
while getopts xy opt_char
do
    case $opt_char in
    x) print "Option is -x";;
    y) print "Option is -y";;
    +x) print "Option is +x";;
    +y) print "Option is +y";;
esac
done
```

Using the `getopts` Statement

- The options on the command line can be given in different ways.
- The following are some valid possibilities for a script that accepts x and y as options:

```
    scriptname -x -y
    scriptname -xy
    scriptname +x -y
    scriptname +x +y
    scriptname +xy
```

Handling Invalid Options

- To process invalid options given on the command line, precede the list of single-character options with a colon.
- For example:

```
while getopts :xy opt_char
```

- The beginning colon:
  - Sets the value of the *`opt_char`* variable to ?
  - Sets the value of the `OPTARG` reserved variable to the name of the invalid option

Handling Invalid Options

```
$ cat getoptsex.sh
#!/bin/bash
# Script name: getoptsex.sh
USAGE="usage: $0 -x -y"
while getopts :xy opt_char
do
        case $opt_char in
        x)
                echo "You entered the x option"
                ;;
        y)
                echo "You entered the y option"
                ;;
        \?)
                echo "$OPTARG is not a valid option."
                echo "$USAGE"
                ;;
        esac
done
```

Specifying Arguments to Options

- If an option requires an argument, place a colon (`:`) immediately after the option in the `getopts` statement.

```
while getopts :x:y opt_char
```

- The colon after the `x` tells the `getopts` statement that an argument must immediately follow.
- After it is executed, the required argument to the `x` option is assigned to the `OPTARG` variable.

The getopts Statement: Examples

```
$ cat getopts1.sh
#!/bin/bash
# Script name: getopts1.sh
USAGE="usage: $0 [-d] [-m month]“
year=$(date +%Y)
while getopts :dm: opt_char
do
        case $opt_char in
        d)
             print -n "Date: “ # -d option given
             date
             ;;
         m)
              cal $OPTARG $year # -m option given with an
              arg
              ;;
         \?)
              print "$OPTARG is not a valid option."
              print "$USAGE"
              ;;
         esac
done
```

The `getopts` Statement: Examples

```
$ ./getopts1.sh -dk
Date: Monday, March 10, 2014 04:10:34 PM IST
k is not a valid option.
usage: ./getopts1.sh [-d] [-m month]
$ ./getopts1.sh -d
Date: Wed Nov 25 18:43:11 IST 2009
```

```
$ ./getopts1.sh –m
$ ./getopts1.sh -m 6
   June 2009
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

```
$ ./getopts1.ksh -d filex
Date: Wed Nov 25 18:44:49 IST 2009
```

The `getopts` Statement: Examples

```
$ cat getopts2.sh
#!/bin/bash
# Script name: getopts1.sh
USAGE="usage: $0 [-d] [-m month]“
year=$(date +%Y)
while getopts :dm: opt_char
do
        case $opt_char in
        d)
             print -n "Date: “ # -d option given
             date
             ;;
       m)
            cal $OPTARG $year # -m option given with an
            arg
            ;;
       \?)
            print "$OPTARG is not a valid option."
            print "$USAGE"
            ;;
```

The `getopts` Statement: Examples

Continued…

```
        :)
                print "The $OPTARG option requires an argument."
                print "$USAGE"
                ;;
        esac
done
```

```
$ ./getopts2.sh -m
The option requires an argument.
usage: ./getopts2sh [-d] [-m month]
```

Quiz

If an option requires an argument, you need to place a colon (`:`) immediately before the option in the `getopts` statement.

a. True

b. False

Summary

In this lesson, you should have learned how to:

- Describe the `for`, `while`, and `until` looping constructs
- Create menus by using the `select` looping statement
- Provide a variable number of arguments to the script by using the `shift` statement
- Parse script options by using the `getopts` statement

Practice 11 Overview: Loops

This practice covers the following topics:

- Using `for` Loops
  - You write a script with a loop construct.
- Using Loops and Menus
  - You modify scripts to use loops and menus.