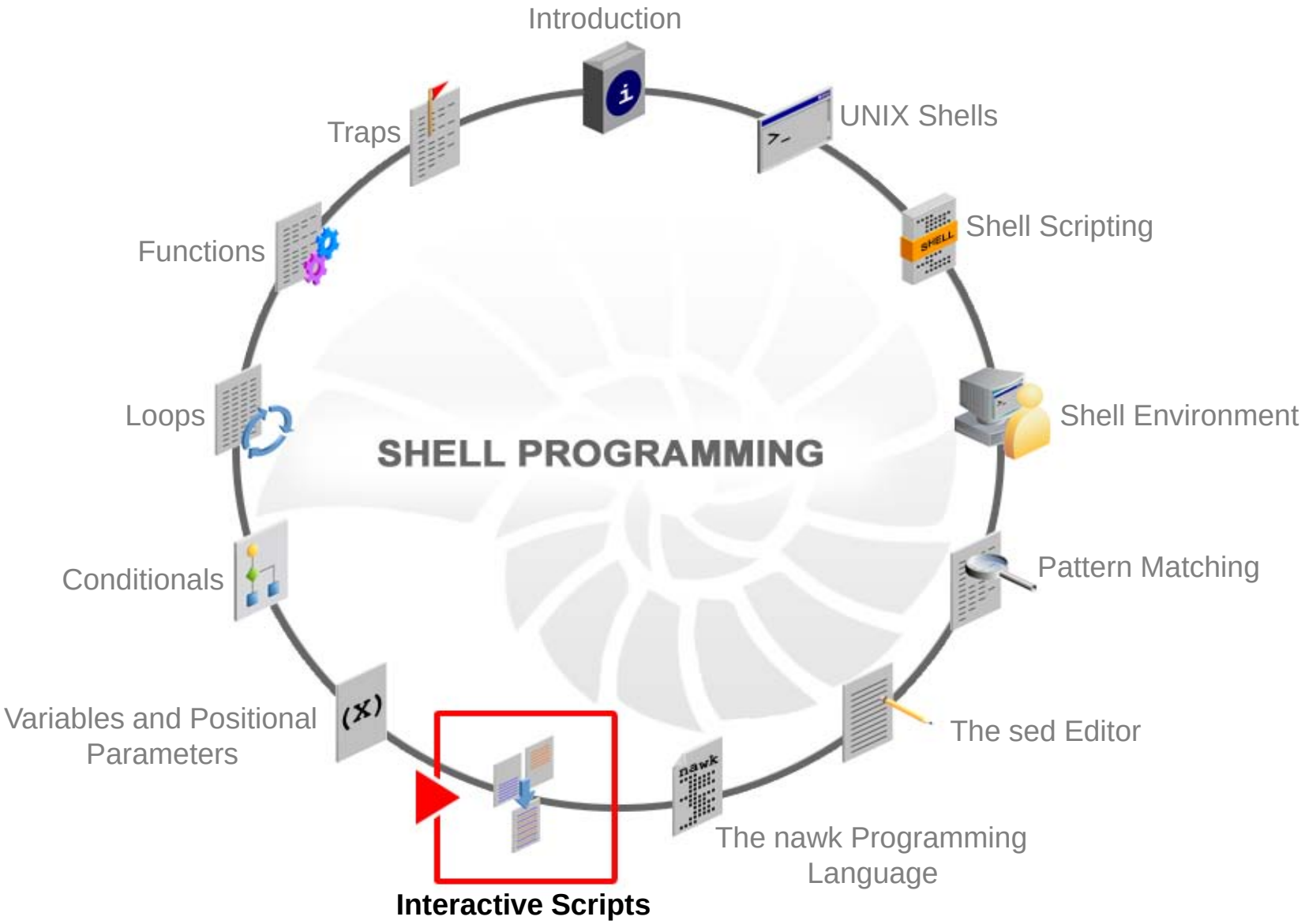
8

Interactive Scripts



Objectives

After completing this lesson, you should be able to:

- Display output by using the `printf` statement
- Accept user input by using the `read` statement
- Describe the role of file descriptors in file input/output redirection

Agenda

- Displaying output by using the `printf` statement
- Accepting user input by using the `read` statement
- Describing the role of file descriptors in file input/output redirection

Interactive Scripts

- Scripts that require input or give output to the user while running are called interactive scripts.
- Both input and output in the scripts can come from different resources as mentioned below:
  - Read command-line arguments
  - Print prompts
  - Read input
  - Test input
  - Print messages
  - File input
  - File output

The `printf` Statement

- You can use the `printf` statement to provide preformatted text output.
- You can use `printf` instead of the `echo` statement.
- The `printf` statement is more versatile than the `echo` statement.
- Syntax:

```
printf  <FORMAT> <ARGUMENTS…>
```

- Example:

```
$ printf "Empname: %s\nEmpcode: %s\n" "$EMPNAME" "$EMPCODE"
Empname:
Empcode:
```

The `printf` Statement: Format Characters

Character	Meaning
`\n`	Prints a newline character, which enables you to print a message on several lines by using one `printf` command
`\t`	Prints a horizontal tab character, which is useful when creating tables or a report
`\a`	Rings the bell on the terminal, which draws the attention of the user
`\b`	Specifies a backspace character, which overwrites the previous character
`\e`	Escape character
`\"`	Double quotation mark
`\\`	Backslash
%b	An argument provided as a string with '`\`' escapes interpretation
%%	A single %

The `printf` Statement: Examples

```
$ printf "%s\n" "This is printf" "in" "the bash"
This is printf
in
the bash

$ printf "\aWrite your details: \nName: "
Write your details:
Name:

$ printf "Phone number: "
Phone number:

$ $ printf "%5d%4d\n" 7 89 789 6789 56789
    7  89
  7896789
56789   0
```

The echo Statement: Examples

```
$ echo "Hello there.\nHow are you?"
Hello there.
How are you?

$ echo "Hello there.\\nHow are you?"
Hello there.\nHow are you?”

$ echo "-2 was the temperature this morning."
-2 was the temperature this morning.

$ echo "No newline printed here. \c"
No newline printed here. $

$ echo "Hello\tout\tthere!"
Hello   out     there!

$ echo "\007Listen to me!"
<bell rings>Listen to me!
```

Agenda

- Displaying output by using the `print` statement
- Accepting user input by using the `read` statement
- Describing the role of file descriptors in file input/output redirection

The `read` Statement

- The `read` statement reads input typed in by the user from the keyboard or from a file.
- The `read` statement is processed in the following manner:
  - Input is broken into tokens that are consecutive characters that do not contain white space.
  - Contents of the `IFS` variable are used as token delimiters.
  - The first token is saved as the value of the first variable.
  - If there are more tokens than variables, the last variable holds all remaining tokens.
  - If there are more variables than tokens, the extra variables are assigned a `null` value.
  - If no variable names are supplied to the read command, the bash shell uses the values supplied in the `REPLY` variable.

The read Statement: Examples

```
$ read var1 var2 var3
abc  def  ghi
$ echo $var1
abc
$ echo $var2
def
$ echo $var3
ghi

$ read num string junk
134 bye93;alk the rest of the line is put in 'junk'
$ print $num
134
$ print $string
bye93;alk
$ print $junk
the rest of the line is saved in 'junk'
```

The `read` Statement: Examples

```
$ read token1 token2 <enter>
One
$ echo $token1
One
$ echo $token2

$ read
What is this saved in?
$ echo $REPLY
What is this saved in?

$ read
read: missing arguments
```

The `read` Statement: Capturing a Command Result

- You can assign the command result to a variable:

```
var=`ls -l /etc/passwd`
```

- You can also capture the command output into a file and then use `read` to assign the values to variables:

```
$ ls -l /etc/passwd > file
$ read a b c d e f g h < file
$ echo $a
-rw-r--r-
$ echo $b
1
$ echo $h
16:20 /etc/passwd
```

The read Statement: Printing a Prompt

```
$ cat io1.sh
#!/bin/bash
# Script name: io1.sh
# This script prompts for input and prints messages
# involving the input received.
echo "Enter your name: \c"
read name junk
echo "Hi $name, how old are you? \c"
read age junk
echo "\n\t$age is an awkward age, $name,"
echo "   You're too old to depend on your parents,"
echo "and not old enough to depend on your children."

$ ./io1.sh
Enter your name: Murdock.
Hi Murdock, how old are you? 25
        25 is an awkward age, Murdock.
   You're too old to depend on your parents,
 and not old enough to depend on your children.
```

The read Statement: Printing a Prompt

```
$ cat io2.sh
#!/bin/bash
# Script name: io2.sh
# This script prompts for input and prints messages
# involving the input received.
print -n "Enter your name: "
read name junk
print -n "Hi $name, how old are you? "
read age junk
print "\n\t$age is an awkward age, $name,"
print " You're too old to depend on your parents,"
print "and not old enough to depend on your children."

$ ./io2.sh
Enter your name: Murdock.
Hi Murdock, how old are you? 25
25 is an awkward age, Murdock.
You're too old to depend on your parents,
and not old enough to depend on your children.
```

Agenda

- Displaying output by using the `print` statement
- Accepting user input by using the `read` statement
- Describing the role of file descriptors in file input/output redirection

File Descriptors

- File input and output are accomplished in the shell by integer handles.
- These numeric handles or values are called file descriptors.
- The best known file descriptors are:
  - `0` (stdin)
  - `1` (stdout)
  - `2` (stderr)
- Numbers 3 - 9 are for programmer-defined file descriptors.

File Redirection Syntax

Command	Description
`< file`	Takes standard input from *file*
`0< file`	Takes standard input from *file*
`> file`	Puts standard output to *file*
`1> file`	Puts standard output to *file*
`2> file`	Puts standard error to *file*
`exec fd> /some/ filename`	Assigns the file descriptor *fd* to */some/ filename* for output
`exec fd< /some/ filename`	Assigns the *fd to /some/ filename* for input
`read <&fd var1`	Reads from the *fd and stores into* variable *var1*
`cmd >& fd`	Executes *cmd and sends output to the fd*
`exec fd<&-`	Closes the *fd*

User-Defined File Descriptors

- You can also use a file descriptor to assign a numeric value to a file instead of using the file name.
- Syntax:

```
exec fd> filename
exec fd< filename
```

Where:

- exec is a built-in command in a shell
- Spaces are not allowed between the file descriptor (fd) number and the redirection symbol (> for output, < for input)

- After a file descriptor is assigned to a file, you can use the descriptor with the shell redirection operators.

```
command >&fd
command <&fd
```

File Descriptors: Example

```
$ cat readex2.sh
#!/bin/bash
# Script name: readex.sh
##### Step 1 - Copy /etc/host
cp /etc/hosts /tmp/hosts2
##### Step 2 - Strip out comment lines
grep -v '^#' /tmp/hosts2 > /tmp/hosts3
##### Step 3 - fd 3 is input file /tmp/hosts3
exec 3< /tmp/hosts3       # fd 3 is an input file /tmp/hosts3
##### Step 4 - fd 4 is output file /tmp/hostsfinal
exec 4> /tmp/hostsfinal   # fd 4 is output file /tmp/hostsfinal
##### The following read and echo statements accomplish STEP 5
read <& 3 addr1 name1 alias    # read from fd 3
read <& 3 addr2 name2 alias    # read from fd 3
exec 3<&-   # Close fd 3
echo $name1 $addr1 >& 4  # write to fd 4 (do not write aliases)
echo $name2 $addr2 >& 4  # write to fd 4 (do not write aliases)
##### END OF STEP 5 statements
exec 4<&-   # close fd 4
```

File Descriptors: Example

```
$ ./readex2.sh

$ more /tmp/hosts2
#
# Copyright 2009 Sun Microsystems, Inc.
# All rights reserved.
# Use is subject to license terms.
# Internet host table
#
::1 s11-server localhost
127.0.0.1 localhost loghost
192.168.10.10  s11-server     #Oracle Solaris server
192.168.10.20  ol6-server     #Oracle Linux Server

$ more /tmp/hosts3
::1 s11-server localhost
127.0.0.1 localhost loghost
192.168.10.10  s11-server     #Oracle Solaris server
192.168.10.20  ol6-server     #Oracle Linux Server
```

The “here” Document

- Frequently, a script might call on another script or utility that requires input.
- To run a script without operator interaction, you must supply that input within your script.
- The “here” document provides a means to do this.
- Syntax:

```
command << Keyword
input1
input2
...
Keyword
```

The “here” Document: Example

```
$ cat termheredoc.sh
#!/bin/bash
# Script name: termheredoc.sh
print "Select a terminal type"
cat << ENDINPUT
    sun
    ansi
    wyse50
ENDINPUT
print -n "Which would you prefer? "
read termchoice
print
print "You choice is terminal type: $termchoice“
$ ./termheredoc.sh
Select a terminal type
    sun
    ansi
    wyse50
Which would you prefer? sun
You choice is terminal type: sun
```

Quiz

After a file descriptor is assigned to a file, you cannot use the descriptor with the shell redirection operators.

a. True
b. False

Summary

In this lesson, you should have learned how to:

- Display output by using the `print` statement
- Accept user input by using the `read` statement
- Describe the role of file descriptors in file input/output redirection

Practice 8 Overview: Interactive Scripts

This practice covers the following topics:

- Writing a ZFS File System Backup Script
  - You write a backup script that accepts a file system as a command-line argument.
- Modifying the `adduser` Script to Prompt for User's Information