Shell Programming

Student Guide

D85153GC10
Edition 1.0
May 2014
D86727

ORACLE

Author

Pardeep Kumar Sharma

Technical Contributors and Reviewers

Joel Goodman
Harald Van Breederode
Vijetha Malkai
Pranamya Jain
Uma Sannasi

Editors

Raj Kumar
Vijayalakshmi Narasimhan
Daniel Milne

Graphic Designer

Maheshwari Krishnamurthy

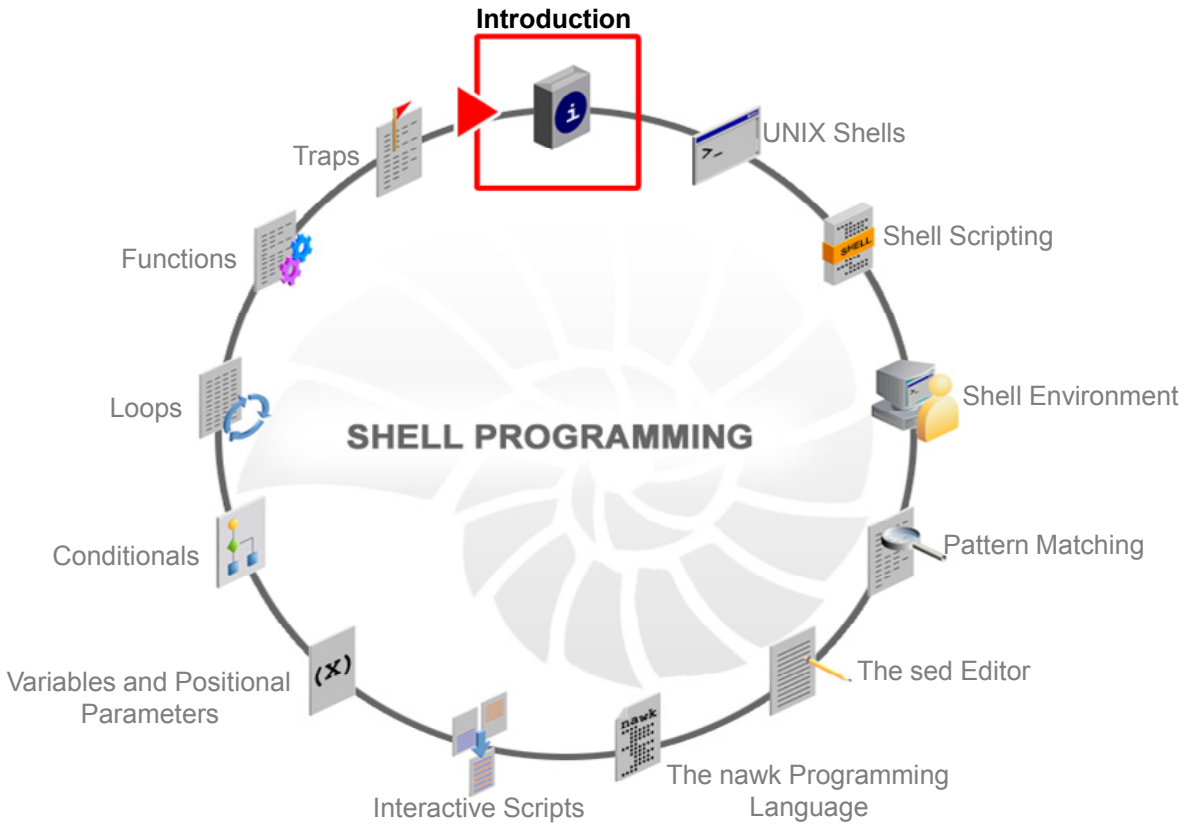
Publishers

Syed Imtiaz Ali
Jayanthy Keshavamurthy

Contents

1

Introduction



Overview

- Course goals
- Course agenda
- Introductions
- Your learning center
- Your practice environment

Welcome to the *Shell Programming* course. The course provides you with the skills to read, write, and debug UNIX shell scripts. The course begins by describing simple scripts to automate frequently executed commands and continues further by describing conditional logic, user interaction, loops, menus, traps, and functions.

This course is intended for system administrators who have mastered the basics of any flavor of the UNIX OS, such as Oracle Solaris or Oracle Linux, and would like to interpret the various boot scripts as well as create their own scripts to automate their day-to-day tasks.

Course Goals

The purpose of this course is to provide you with the knowledge and skills necessary for developing shell scripts to automate basic and advanced system administration–related tasks.

Course Objectives

After completing this course, you should be able to:

- Create scripts to automate system administration tasks
- Set local and environmental variables
- Automate tasks by using regular expression characters with the `grep`, `sed`, and `nawk` utilities
- Create interactive scripts by using flow control constructs
- Perform string manipulation and integer arithmetic on shell variables
- Debug errors in scripts

ORACLE

Course Agenda: Day 1

- Lesson 1: Introduction
- Lesson 2: UNIX Shells
  - Describes the role of a shell in a UNIX environment
  - Describes the various UNIX/Oracle Solaris shells
- Lesson 3: Shell Scripting
  - Describes the structure of a shell script
  - Describes how to create a simple shell script
  - Explains how to implement the various debugging options in a shell script

Note

- The class is from 9:00 AM to 5:00 PM each day.
- There will be several short breaks throughout the day with an hour's break for lunch.

ORACLE

Day 1 covers three lessons that discuss shell basics, script analysis, and design.

- Lesson 1: Introduction
  - This preface introduces the course objectives and course agenda.
- Lesson 2: UNIX Shells
  - This lesson describes the fundamentals of UNIX shells including various UNIX and Oracle Solaris shells.
- Lesson 3: Shell Scripting
  - This lesson introduces you to the structure of a shell script.

Course Agenda: Day 2

- Lesson 4: Shell Environment
  - Describes the role of startup scripts in initializing the shell environment
  - Describes the various types of shell variables
  - Explains command-line parsing in a shell environment
- Lesson 5: Pattern Matching
  - Describes the `grep` command
  - Explains the role of regular expressions in pattern matching
- Lesson 6: The `sed` Editor
  - Describes the `sed` editor
  - Describes how to perform noninteractive editing tasks by using the `sed` editor

ORACLE

Day 2 starts with a recap of the previous day and focuses on the following lessons that cover useful shell utilities and pattern matching concepts:

- Lesson 4: Shell Environment
  - This lesson explains the role of startup scripts that initialize the shell environment. The lesson also describes the various shell variables and command-line parsing in a shell environment.
- Lesson 5: Pattern Matching
  - This lessons explain the mechanism to search by using regular expressions and the `grep` command
- Lesson 6: The `sed` Editor
  - The lesson describes editing input streams and performing noninteractive tasks by using the `sed` editor.

Course Agenda: Day 3

- Lesson 7: The `nawk` Programming Language
  - Describes `nawk` as a programming language
  - Displays output by using the print statement
  - Explains how to perform pattern matching by using regular expressions
  - Explains the use of the `nawk` built-in and user-defined variables
- Lesson 8: Interactive Scripts
  - Displays output by using the `print` and `echo` statements
  - Explains how to accept user input by using the `read` statement
  - Describes the role of file descriptors in file input and output

ORACLE

Day 3 starts with a recap of the previous day. This is followed by the next two lessons that discuss the shell programming language and its constructs.

- Lesson 7: The `nawk` Programming Language
  - This lesson describes steps to create scripts using the `nawk` programming language and its built-in features.
- Lesson 8: Interactive Scripts
  - This lesson describes how to create interactive scripts that perform multiple `operations`.

Course Agenda: Day 4

- Lesson 9: Variables and Positional Parameters
  - Describes the various types of scripting variables
  - Defines positional parameters for accepting user input
- Lesson 10: Conditionals
  - Describes the role of the if statement in testing conditions
  - Describes the syntaxes for the if/then/else and if/then/elif/else statements
  - Describes how to choose from alternatives by using the case statement
  - Explains how to perform numeric and string comparisons
  - Explains how to compare data by using the &&, ||, and ! Boolean operators
  - Explains the difference between the exit status and the exit statement

ORACLE

Day 4 starts with a recap of the previous day. This is followed by the next two lessons that discuss the programming constructs used in creating advanced shell scripts.

- Lesson 9: Variables and Positional Parameters
  - This lesson discusses the various types of scripting variables and positional parameters for accepting user input.
- Lesson 10: Conditionals
  - This lesson discusses the programming techniques to include decision making points within scripts by using conditionals.

Course Agenda: Day 5

- Lesson 11: Loops
  - Describes the `for`, `while`, and `until` looping constructs
  - Explains how to create menus by using the `select` looping statement
  - Describes variable number of arguments to a script by using the `shift` statement
  - Describes the role of the `getopts` statement in parsing script options
- Lesson 12: Functions
  - Explains how to create user-defined functions in a shell script
  - Describes the use of the typeset and unset statements in a function
  - Explains how to autoload a function file into a shell script
- Lesson 13: Traps
  - Describes the role of shell signals in interprocess communication
  - Explains how to catch signals and user errors with the `trap` statement

ORACLE

Day 5 starts with a recap of the previous day. This is followed by the last three lessons in this course that discuss the advanced programming constructs and few useful scripts for system administrators.

- Lesson 11: Loops
  - This lesson introduces you to loops that help automate routine operations.
- Lesson 12: Functions
  - This lesson introduces you to the functions that help automate and reuse existing code.
- Lesson 13: Traps
  - This lesson discusses traps, a facility to debug the scripts.

How Prepared Are You?

A Yes as an answer to the following questions indicates that you are prepared to take this course:

- Can you install, configure, and maintain Oracle Solaris Zones and Oracle Solaris 11 operating systems?
- Can you administer users, packages, SMF services, and applications on Oracle Solaris 11 systems?

Introductions

Now that you have been introduced to the course format, introduce yourself to the other students and the instructor, addressing the following items:

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to the topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

Your Learning Center

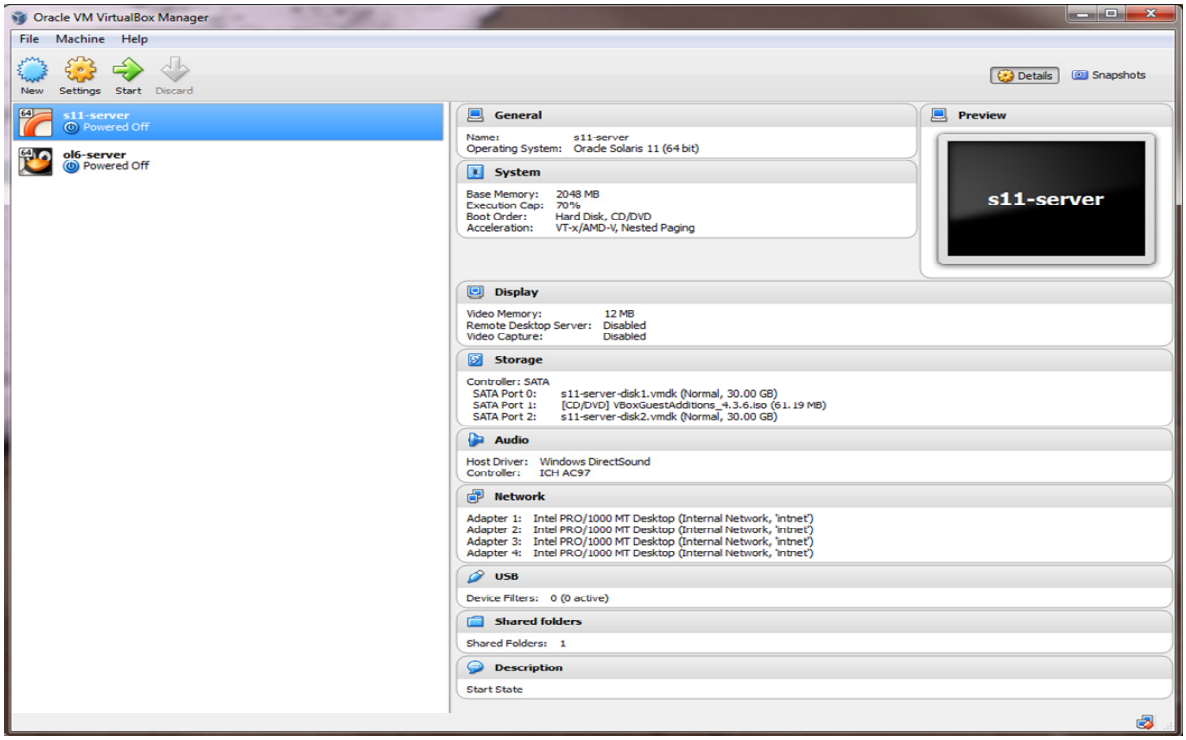
The instructor will acquaint you with the following details:

- Layout of the training facility
  - Restrooms
  - Break rooms and designated smoking areas
  - Cafeterias and restaurants in the area
- Emergency evacuation procedures
- Instructor contact information
- Cell phone usage
- Online course attendance confirmation form

ORACLE

Now that you have an idea of what you will be doing over the next five days, let's get started with an introduction to the practice environment.

Your Practice Environment



ORACLE

As part of each lesson, you will be given the opportunity to practice in a lab environment. The environment that you use in this course is based on the Oracle VM VirtualBox virtualization software, an example of which is shown in the slide. VirtualBox is a cross-platform virtualization application. It extends the capabilities of your existing computer so that it can run multiple operating systems inside multiple virtual machines at the same time.

Open your Activity Guide to “Practices for Lesson 1: Introduction.” Your instructor will walk you through the material, and you will have a chance to familiarize yourself with the practice environment, configuration, and setup.

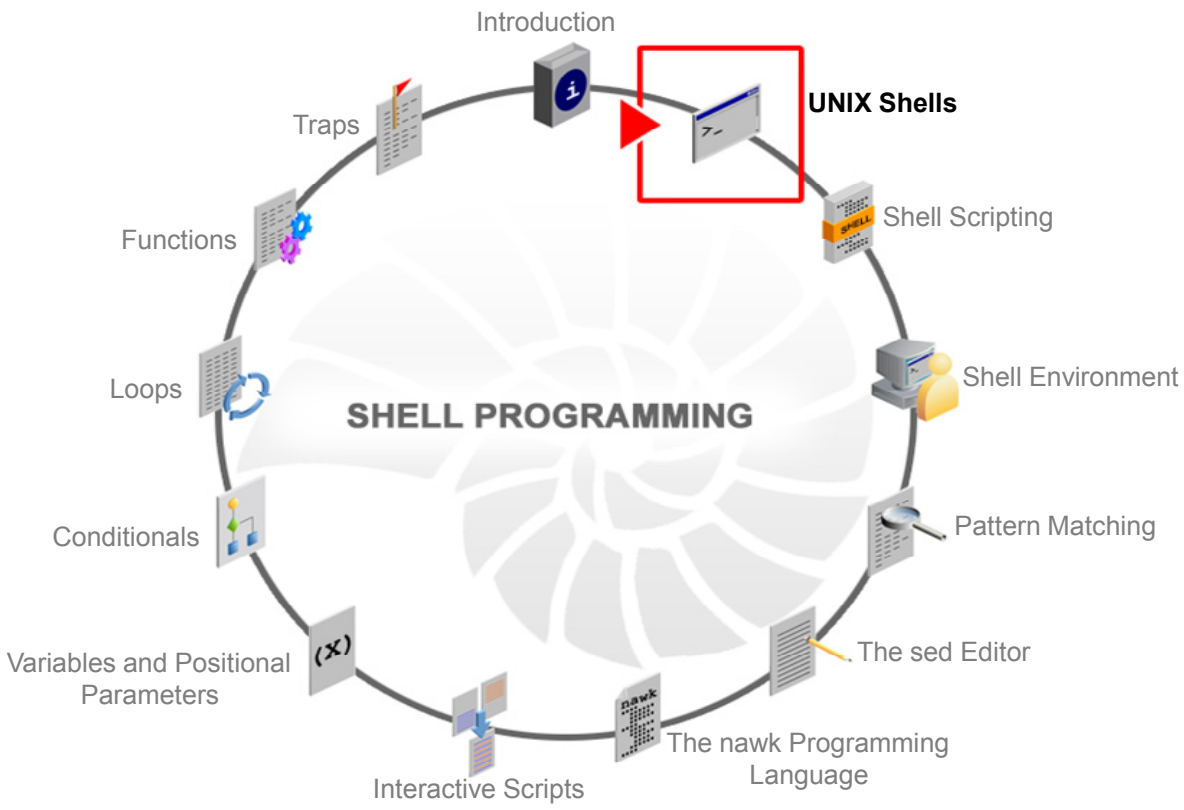
Practice 1 Overview: Introduction

This practice covers the following topic:

- Getting Familiar with Your Practice Environment

2

UNIX Shells



Objectives

After completing this lesson, you should be able to:

- Describe the role of a shell in the UNIX environment
- Describe the various UNIX shells

Agenda

- Describing the role of a shell in the UNIX environment
- Describing various UNIX shells

What Is a Shell?

- A shell is a program that provides an interface between a user and an operating system (OS) kernel.
- An OS starts a shell for each user when the user logs in or opens a terminal or console window.
- The shell's primary function is to read commands that are typed into a console or terminal window and then execute them.
- All OSs have shells:

OS	Shell
DOS	command.com
Oracle Solaris	Bash
Oracle Linux	Bash
MAC OS X	Bash
MS Windows	cmd.exe

ORACLE

As a user logs in or switches to another user including `root`, the shell starts a shell command, such as `sh`, `ksh`, or `csh`, depending on the shell in use. These commands accept user input, interpret them, and provide output.

Functions of a Shell

The main functions of a shell include the following:

- Command-line interpreter
- Programming language
- User environment

Command-Line Interpreter

When you type a command on the command line, the shell:

1. Interprets the command by parsing the command line and addressing metacharacters, redirection, and control.
2. Searches for and executes the command.
3. Analyzes each command and initiates execution of the requested program.

For example, when you issue the `ps -ef | sort +1 | more` set of commands (pipeline), the shell does the following:

1. It breaks the command line into pieces called *words:*
   - `ps`, `-ef`, `|`, `sort`, `+1`, `|`, and more
2. It determines the *purpose* of the word*:*
   - `ps`*,* `sort`*,* and `more` are *commands*
   - `-ef` and `+1` are *arguments.*
   - `|` is an *input/output (I/O) operation.*
3. Based on the shell-parse order, the shell sets up the `ps` output to be the input to `sort`, and the `sort` output to be the input to `more`.
4. It locates the `ps`, `sort`, and `more` commands and then executes them in order, applying the arguments as specified on the command line.

Programming Language

- You can type the commands directly into the shell at the prompt, or the shell can read commands from a file.
- A shell can contain two types of commands:
  - Built-in: `cd`, `exit`, `break`
  - Binary: `cat`, `cp`, `ls`
- A file containing shell commands is called a shell program or a shell script.
- Apart from commands, a shell script can contain programming constructs.
- A shell script is a text file that is interpreted, not compiled.
- The shell reads a line in the shell script and processes all statements found on that line before reading the next line.

As you perform a task repetitively, you might type the same commands over and over again. You can automate this task by placing these commands in a file and executing them as a shell script.

A shell script is a text file that is interpreted, not compiled. Compiled programs written in programming languages such as C and C++ are text files that are preprocessed by a compiler to generate a stand-alone binary executable. Binary executables are loaded into memory at the time of execution and their stand-alone nature means they do not require an initial load of an interpreter to process their statements. This is the reason that even the fastest shell programs run slower than an equivalent program written in a compiled language.

User Environment

- The shell also provides a user environment that you can customize by using initialization files.
- These files contain settings for user environment characteristics, such as:
  - Search paths for finding commands
  - Default permissions on new files
  - Values for variables that other programs use
  - Values that you can customize

Note: You will learn more about initialization files in the lesson titled “Shell Environment.”

Agenda

- Describing the role of a shell in a UNIX environment
- Describing various UNIX shells

Types of Shells

UNIX-like OSs support a variety of shells:

Shell	Program File Name	File Path	Non-root default prompt	Root default prompt
Bash	`bash`	`/bin/bash`	`$`	`#`
Bourne	`sh`	`/bin/sh`	`$`	`#`
Korn	`ksh`	`/bin/ksh`	`$`	`#`
Z	`zsh`	`/bin/zsh`	`$`	`#`
Note: Above shells are Bourne-compatible				
C	`csh`	`/bin/csh`	`%`	`#`
TC	`tcsh`	`/bin/tcsh`	`%`	`#`

ORACLE

In this course, you learn about the GNU Bourne-Again shell (bash), which is the default shell in Oracle Solaris 11.x and Oracle Linux 6.x.

The Bourne Shell

- The Bourne shell (`sh`):
  - Is the original UNIX shell written by Steve Bourne at AT&T Bell Labs
  - Is the preferred shell for shell programming because of its compactness and speed
- The command full-path name is `/bin/sh` and `/sbin/sh`.
- The default prompt:
  - For a non-root user is `$`
  - For a root user is `#`

Drawbacks of the Bourne shell:

- Lacks features for interactive use, such as the ability to recall previous commands (history)
- Lacks built-in arithmetic and logical expression handling

Note: The Bourne and C-shell families are almost completely incompatible from a programming standpoint.

The C Shell

- The C shell (`csh`):
  - Is a UNIX enhancement written by Bill Joy at the University of California at Berkeley
  - Has features for interactive use such as aliases and command history
  - Includes convenient programming features such as built-in arithmetic and a C-like expression syntax
- The command full-path name is `/bin/csh`.
- The default prompt:
  - For a non-root user is `%`
  - For a root user is #

ORACLE

The syntax of the C shell programs is similar to the C programming language, thus the name. The C shell is quite preferred for interactive use. However, it is larger and slower than the Bourne shell.

The Korn Shell

- The Korn shell (`ksh`):
  - Is a superset of the Bourne shell written by David Korn at AT&T Bell Labs
  - Supports everything in the Bourne shell
  - Has interactive features comparable to those in the C shell
  - Includes convenient programming features such as built-in arithmetic and C-like arrays and functions
  - Is faster than the C shell
  - Runs scripts written for the Bourne shell
- The command full-path name is `/bin/ksh`.
- The default prompt:
  - For a non-root user is `$`
  - For a root user is `#`

ORACLE

The Bash Shell

- The Bash shell:
  - Is compatible with the Bourne shell
  - Includes useful features from the Korn and C shells
  - Provides arrow keys that are automatically mapped for command recall and editing
- The command full-path name is `/bin/bash`.
- The default prompt:
  - For a non-root user is `bash-x.xx$`
  - For a root user is `bash-x.xx#`

The default prompts for non-root and root users are `bash-x.xx$` and `bash-x.xx#`, respectively:

- Where `x.xx` indicates the shell version number
- For example: `bash-3.50$` and `bash-3.50$#`

The Z Shell

- The Z shell (`zsh`):
  - Closely resembles the Korn shell
  - Includes built-in spelling correction and programmable command completion
- The command full-path name is `/bin/zsh`.
- The default prompt:
  - For a non-root user is `%`
  - For a root user is `#`

The Enhanced C Shell

- The Enhanced C shell (`tcsh`):
  - Is compatible with the C shell
  - Has command-line editor, programmable word completion, and spelling correction features
  - Has arrow keys that are automatically mapped for command recall and editing
- The command full-path name is `/bin/tcsh`.
- The default prompt:
  - For a non-root user is `>`
  - For a root user is `#`

Shell Features Comparison

Feature	Bash	Bourne	Korn	C	TC
Programming language	Yes	Yes	Yes	Yes	Yes
Shell variables	Yes	Yes	Yes	Yes	Yes
Command alias	Yes	No	Yes	Yes	Yes
Command history	Yes	No	Yes	Yes	Yes
File name completion	Yes	No	Yes*	Yes*	Yes
Command line editing	Yes	No	Yes*	No	Yes
Job control	Yes	No	Yes	Yes	Yes

Note: * means not a default setting for this shell.

The table summarizes the features available with popular shells. Features pertaining to the bash shell will be discussed in subsequent lessons in this course.

Using a Different Shell

- To change your current shell, type in an alternative shell (command) at the prompt.
- For example, if you are in the bash shell, type `csh` at the prompt to change your shell to the C shell.
- To determine your current shell, execute the `ps` command.
- If you have opened a series of shells, use the `ps | sort` command to sort the shells in the order in which you invoked them.

Summary

In this lesson, you should have learned how to:

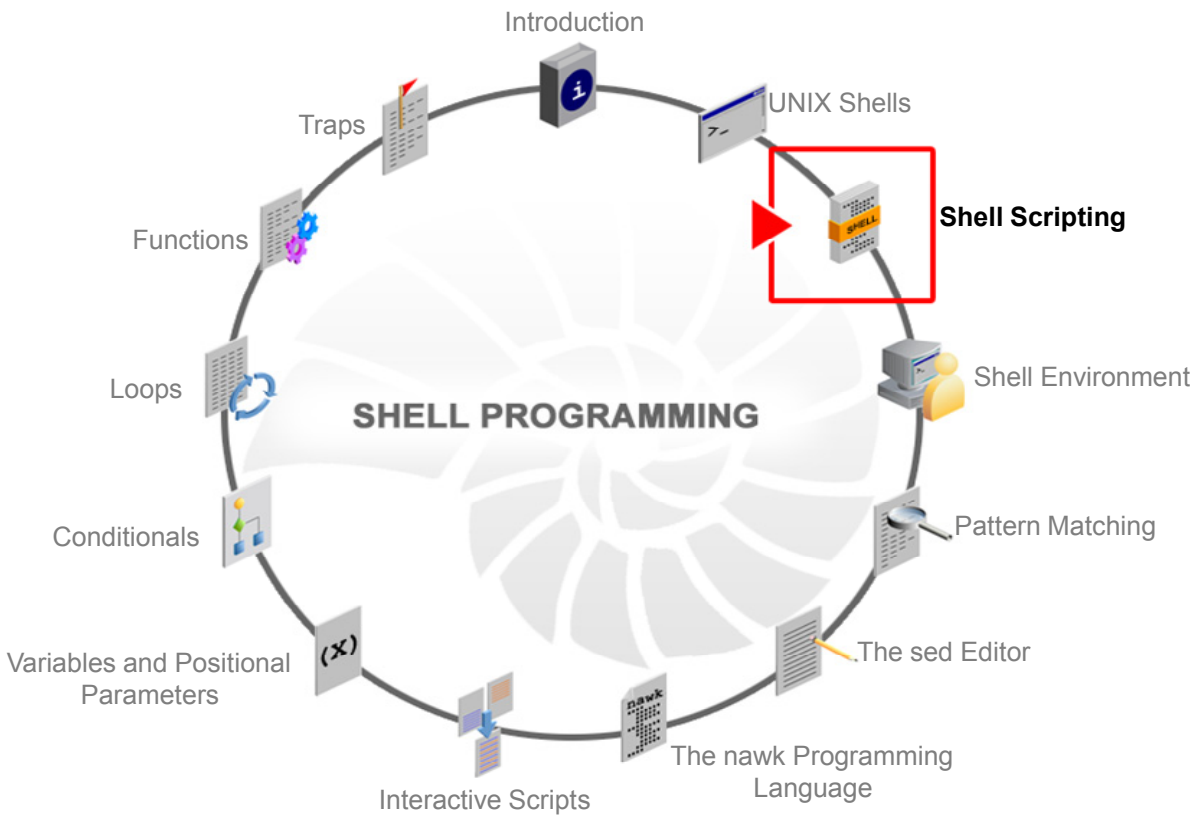
- Describe the role of shells in the UNIX environment
- Describe the standard shells

Practice 2 Overview: UNIX Shells

This practice covers the following topic:

- Reviewing Unix Shells

3 Shell Scripting



ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe the structure of a shell script
- Create a simple shell script
- Implement the various debugging options in a shell script

Agenda

- Describing the structure of a shell script
- Creating a simple shell script
- Implementing the various debugging options in a shell script

What Is a Shell Script?

- A shell script is a noncompiled program written for the shell or the command-line interpreter of an operating system (OS).
- Shell scripts are:
  - Text files that contain shell and UNIX commands
  - Programs that have a specific purpose
  - Interpreted by a shell (for example, bash shell)
  - Useful in automating repetitive tasks

ORACLE

Similar to compiled executable programs, the shell script has a specific purpose and could be reusable. When the commands in the shell script are ordered and error-free, you can execute the shell script to carry out specific tasks.

All scripts, programs, and procedures use the following rules:

- They should execute without error.
- They perform the task for which they are intended.
- They ensure that the program logic is clearly defined or apparent.
- They should not incorporate unnecessary work.

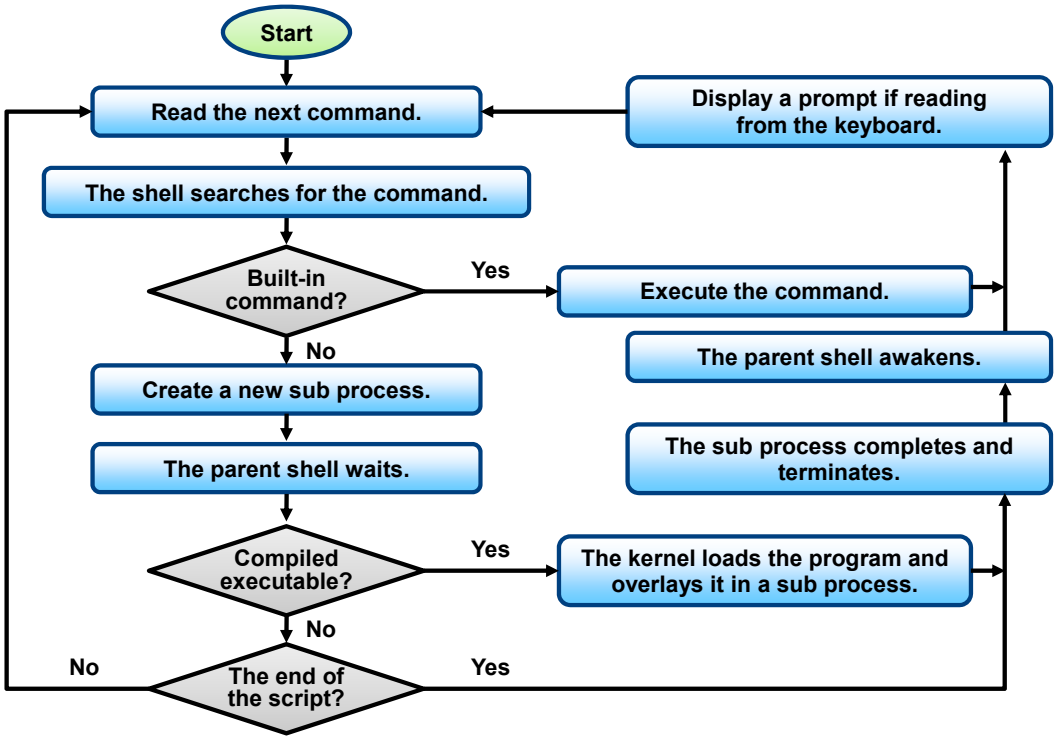
Components of a Shell Script

- For a script to execute, you must ensure that it has correct logic and flow control.
- Beyond that, the structure of a shell script is flexible and some of the components include:
  - Comments
  - Information displays
  - Conditional testing
  - Looping constructs
  - Arithmetic operations
  - String manipulation
  - Variable manipulation and testing
  - Argument and option handling

ORACLE

Note: You learn more about each of these components in subsequent lessons.

Execution Order of a Shell Script



The shell first determines the type of program that is to be executed. It is either a compiled (executable) program or a script file for one of the available shells. If it is an executable program, a new process is created for the program by means of a *fork*, which makes a copy of the shell process. The new process is called a *child process* or *subshell*. The kernel loads the program into the child process (called `exec`'ing the program). The program then begins running, and the shell process waits until the child process finishes.

If the program being executed is a shell script, the execution process is a little different. The shell creates a new shell process (also by *forking*). This new process (subshell) reads the lines from the shell script file one at a time. The subshell reads, interprets, and executes each line as if it had come from the keyboard. The entire execution process that is described in this module is repeated for each line of the shell script.

While the subshell processes the lines of the shell script, the parent shell waits for it to finish. When there are no more lines in the shell script file to read, the subshell terminates and the parent shell displays a new prompt.

Agenda

- Describing the structure of a shell script
- Creating a simple shell script
- Implementing the various debugging options in a shell script

Simple Shell Script: Example

In this sample shell script named `my_script`:

- `#!/bin/bash`: Specifies the shell the script is written for. In this case it is the bash shell.
- `# My first script`: Is a comment entry, which is not interpreted by the shell
- `echo`: Is a command to display text on the screen
- `exit`: Is a command that denotes the end of script and tells the shell to exit the script

```
# cat my_script.sh
#!/bin/bash
# My first script

echo "Hello World!"
exit 0
```

Requirements: Tools and Resources

Before you begin to write a shell script, you need to:

- Choose an OS
  - UNIX-based OS (Oracle Solaris 11.1, Oracle Linux 6.5)
- Choose an editor
  - `vi`, `vim`, `sed`, `gedit` (graphical)
- Set up the environment
  - The shell profile, aliases, history by using the `vi` editor

ORACLE

When you start to develop your script, determine the type of information needed before writing the script. For example, your script might:

- Prompt the user for the directory name
- Read and store the directory name in a variable
- Verify that the directory exists and that the user has permissions for the directory
- Execute the `tar` command and create the archive file (perhaps in a temporary directory)
- Compress the archive file (into the current directory)
- Notify the user of the name of the compressed file

Programming Terminologies

Term	Meaning
Logic flow	The overall design of the program or procedure. The logic flow determines the logical sequence of the tasks that are involved in the program or procedure so that a successful and controlled outcome is achieved.
Loop	A portion of the procedure that is performed zero or more times.
User input	Information that is provided by an external source, during the running of the program or procedure, that is used within the program or procedure.
Conditional branch	A logical point in the program or procedure when a condition determines the actions that are subsequently performed.
Command control	Testing the `exit` status of a command to control whether a portion of code should be performed.

ORACLE

Note: You learn more about each of these terminologies and how they are used in shell scripting in subsequent lessons.

Instance of a Logic-Flow Design

- When developing a script, use the logic-flow design.
- For example:
  1. Do you want to add a user?
     a. If Yes:
        1. Enter the user's name.
        2. Choose a shell for the user.
        3. Determine the user's home directory.
        4. Determine the group to which the user belongs.
     b. If No, go to Step 3.
  2. Do you want to add another user?
     a. If Yes, go to Step 1.a.
     b. If No, go to Step 3.
  3. Exit.

ORACLE

A number of standard methods are used in computing. One of the simplest methods of logic-flow design is itemized lists. By itemizing the list of processes involved in the program, you can describe each task, or process, and refer to each individual task by its associated number.

If you start by writing the tasks in a descriptive format using the English language, the program is in an understandable form. Later, you can transform the English-language statements into the appropriate computer-language statements.

The example in the slide shows a possible repetitive loop, which is controlled by the number of users, between steps 1.a and 2.a. The user must be provided with information, such as "These are the available shells (sh, ksh, csh). Which would you prefer?"

Additionally, input from the user must be obtained and stored; for example, the choice for user name, shell, and group.

The `echoscript1.sh` Script: Example

```
# cat echoscript1.sh
#!/bin/bash

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME"
Echo

echo "Todays date is: \c"
date '+%m/%d/%y'

echo "and the current time is: \c"
date '+%H:%M:%S%n'

echo "Now a list of the processes in the current shell"
ps

echo "SCRIPT FINISHED!!"
```


The `echoscript1.sh` script uses some familiar commands to display information about the workstation.

Note: The specification of full path names to commands is not necessary if you plan to be logged in when you execute the script and you are sure that your `PATH` variable uses the correct version of every command (`/bin/ps` instead of `/usr/ucb/ps`).

If you are executing this script from `cron`, you must supply full path names and redirect output and errors.

Creating a Shell Script

To create a shell script, perform the following steps:

1. Create a file.
2. Invoke the shell.
3. Include commands in the script file.
4. Add comments to the script file, if required.
5. Execute the script file.

Creating a File

- A script file can have any name that follows the conventions of regular file names in the OS environment.
- When naming your shell scripts, avoid using names that conflict with existing UNIX commands or shell functions, unless you wish to modify the behavior of an existing command.

Invoking the Shell

- The first line of the script contains the shebang characters that invoke the shell interpreter.
- The shebang characters on the first line are `#!`, followed by the path name for the subshell to execute the script.

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

Note: The shebang characters are *the first two characters on the first line,* meaning that there must be no blank lines or spaces before these characters.

In the above example, the `#!` characters confirm the file type and utility used by the OS before attempting to execute it. `/bin/bash` is the absolute path name of the utility that is used to execute the script.

Including Commands in a Shell Script

UNIX commands include standard utilities, user programs, and the names of other scripts that you need in order to accomplish your task.

Note: If you put the names of other scripts in your script, ensure they have `read` and `execute` permissions so that they will run.

Adding Comments in a Shell Script

- Any line other than the first line in the script that starts with a # is a comment.
- Comments provide useful documentation to programmers and others who need to understand or debug the code.
- For example:

```
$ cat hello.sh
#!/bin/bash
# This is a hello world program
```


It is a good practice to put comments into programs and shell scripts. Comments are meant to explain the purpose of the script and should describe any specific lines that might be especially confusing.

Comments can be included anywhere in the script, wherever a developer feels the need to explain a particular step or procedure.

The example script `hello.sh` has a comment at the beginning of the script saying what kind of a program it is.

Executing a Shell Script

There are various ways to execute a shell script:

- To execute a shell script in the subshell, first assign the `execute` permission to the shell script.

```
$ chmod 744 scriptname
$ scriptname
```

- To execute a shell script in the current shell, you do not need the `execute` permission on the script.

```
$ . ./scriptname
```

- To invoke certain options in the script, execute a script by using the shell command.

```
$ bash -s scriptname
```


There are various ways to execute a shell script:

- To execute a shell script in the subshell, first assign the `execute` permission to the shell script by using the `chmod` command as displayed in the slide. This is the preferred and most common method for executing a shell script. A subshell is created to execute the specified script. Give `execute` permission only to users who need to execute the script. When a script is executed in a subshell, the variables, aliases, and functions created and used in the script are known only in the subshell. After the script finishes and control returns to the parent shell, the variables, functions, and other changes to the state of the shell made by the script are no longer known.
- To execute a shell script in the current shell, you do not need the `execute` permission on the script. Any operations you perform on shell variables or aliases take effect in the current shell and are seen even after the shell script terminates.
- To invoke certain options on a script that you do not have permission to execute (or do not want to execute), execute the script by using the `shell` command. Again, a subshell is created to execute the specified script. This is used in situations when you want the shell to start with specific options not explicitly specified in the script. The typical situation is when you want to turn on and off debugging for the entire script. In the example in the slide, debugging is being turned on with the `-s` option.

Example: `hello.sh` Script

```
$ vi hello.sh

#!/bin/bash
# This and following line is a comment
# This is a hello world program

echo "Hello World"
~
~
~
:wq

$ chmod 700 hello.sh

$ ./hello.sh
Hello World
```


The first line in the example script tells the system to use the bash interpreter to run this script. After saving the file, `execute` permission is assigned to the script. On executing the script, "`hello world`" is displayed as output on the screen.

Quiz

What is the purpose of the shebang characters?

a. Creating a file
b. Invoking the shell
c. Adding comments to the script file, if required
d. Executing the script file

Answer: b

Agenda

- Describing the structure of a shell script
- Creating a simple shell script
- Implementing the various debugging options in a shell script

Debugging a Shell Script

- The shell provides a debugging mode, which allows you to locate problems in a script.
- To run an entire script in the debug mode:
  - Add `-x` after `#!/bin/bash` on the first line.

```
#!/bin/bash -x
```

- To run an entire script in the debug mode from the command line:
  - Add `-x` to the bash command used to execute the script.

```
$ bash -x scriptname
```


Note: You can run an entire script or a portion of the script in the debug mode.

Debug Statement Options

To run several portions of a script in the debug mode, place `set -option` at the debugging start point and `set +option` where it needs to stop.

Option	Description
`set -x`	Prints the statements after interpreting metacharacters and variables
`set -v`	Prints the statements before interpreting metacharacters and variables
`set -f`	Disables file name generation (using metacharacters)

To turn on the option, you must use `set -`*`option`*, where *option* is one, or a combination, of the letters `x`, `v`, and `f`.

To turn off the option, you must use, `set +option`, where *option* is one, or a combination, of the letters `x`, `v` and `f`.

- The `set -x` option allows you to see the statement line after the shell interprets any metacharacters and performs any statement, variable, or file name substitutions. For the portion of the script file running in this debug mode, each statement is output, preceded by a `+`, before the statement is executed.

- The `set -v` statement is similar to `set -x`, except that it shows the statement line before the shell performs any interpretation or substitution. When you use these two options, with `set -xv`, the shell displays the statement lines both before and after interpretation and substitution, so that you can see how the shell has used the statement line in the script.

  For the following statement in a shell script:

  ```
  echo $HOME              # echo the HOME directory path
  ```

  The statement line, before interpretation (`set -v`) is:

  ```
  echo $HOME              # echo the HOME directory path
  ```

  The statement line, after interpretation (`set -x`) is:

  ```
  + echo /export/home/user200
  ```

  If both options are turned on, the `-v` line is always displayed before the `-x` line. Also, the `+` character does not precede the `-v` line.
- The `set -f` option turns off the shell file name substitution (globbing) capability. This disables the special meaning of characters, such as `*` and `?`, when used as part of a file or directory name.

Example: Debug Mode Specified on the #! Line

```
$ cat debug1.sh
#!/bin/bash -x

echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo
```


In this example, the debug mode is turned on for the entire script by appending `-x` to the first line of the script.

Example: Debug Mode Specified on the #! Line

```
$ ./debug1.sh
+ echo Your terminal type is set to: ansi
Your terminal type is set to: ansi
+ echo

+ echo Your login name is: root
Your login name is: root
+ echo

+ echo Now we will list the contents of the /etc/security directory
Now we will list the contents of the /etc/security directory
+ ls /etc/security

audit                   audit_user      dev             policy.conf
audit_class             audit_warn      device_policy   priv_names
audit_control           auth_attr       exec_attr       prof_attr
audit_event             bsmconv         extra_privs     spool
audit_record_attr       bsmunconv       kmfpolicy.xml   tsol
audit_startup           crypt.conf      lib
+ echo
```


In the debug mode, each command is displayed with a + in front of it, and then it is executed. If you remove all lines with a + at the beginning, the remainder is the output from the normal running of the script.

Example: Debug Mode with the `set -x` Option

```
$ cat debug2.sh
#!/bin/bash

set -x
echo "Your terminal type is set to: $TERM"
echo
set +x

echo "Your login name is: $LOGNAME"
cho

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo
```


The example in the slide shows the debug mode turned on for only a portion of the script.

Example: Debug Mode with the `set -x` Option

```
$ ./debug2.sh
+ echo Your terminal type is set to: ansi
Your terminal type is set to: ansi
+ echo

+ set +x
Your login name is: root

Now we will list the contents of the /etc/security directory
audit                   audit_user       dev              policy.conf
audit_class             audit_warn       device_policy    priv_names
audit_control           auth_attr        exec_attr        prof_attr
audit_event             bsmconv          extra_privs      spool
audit_record_attr       bsmunconv        kmfpolicy.xml    tsol
audit_startup           crypt.conf       lib
```


On executing the file, the commands for which the debugging option was set are displayed with a + in front of it, and then they are executed.

Example: Debug Mode with the `set -v` Option

```
$ cat debug3.sh
#!/bin/bash

set -v
echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo
```


The example in the slide shows the `-v` debug mode turned on for the entire script. This option:

- Prints the statement prior to executing but without a leading `+` (unlike what the `-x` option displays)
- Does not translate metacharacters; therefore, it prints the statement exactly as it is in the script
- Prints comment lines

Example: Debug Mode with the `set -v` Option

```
$ ./debug3.sh
echo "Your terminal type is set to: $TERM"
Your terminal type is set to: ansi
echo

echo "Your login name is: $LOGNAME"
Your login name is: root
echo

echo "Now we will list the contents of the /etc/security directory"
Now we will list the contents of the /etc/security directory
ls /etc/security
audit audit_              user             dev              policy.conf
audit_class               audit_warn       device_policy    priv_names
audit_control             auth_attr        exec_attr        prof_attr
audit_event               bsmconv          extra_privs      spool
audit_record_attr         bsmunconv        kmfpolicy.xml    tsol
audit_startup             crypt.conf       lib
echo
```

Quiz

Which of the following commands would you use to run an entire script in the debug mode?

a. `$ bash -x scriptname`

b. `#!/bin/bash -x`

c. `set -x`

d. `set -v`

Answer: b

Summary

In this lesson, you should have learned how to:

- Describe the structure of a shell script
- Create a simple shell script
- Implement the various debugging options in a shell script

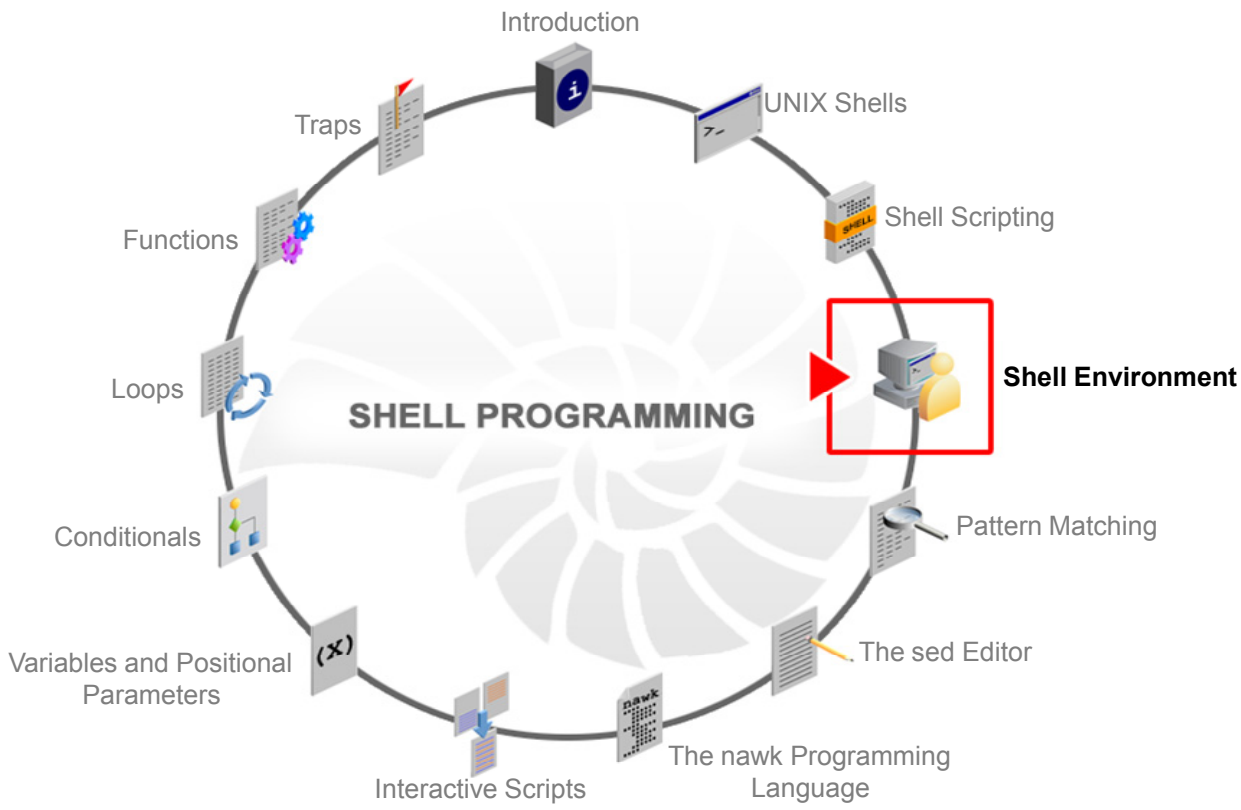
Practice 3 Overview: Shell Scripting

This practice covers the following topics:

- Reviewing Shell Scripts
- Writing and Debugging Shell Scripts

4

Shell Environment



Objectives

After completing this lesson, you should be able to:

- Explain the role of startup scripts in initializing the shell environment
- Describe the various types of shell variables
- Explain command-line parsing in a shell environment

Agenda

- Explaining the role of startup scripts in initializing the shell environment
- Describing the various types of shell variables
- Explaining command-line parsing in a shell environment

The Shell Environment

- The shell environment is defined by variables.
- Some of these variables are set by any of the following:
  - The system
  - The user
  - The shell
  - A program that loads another program

The Startup Scripts

When you log in to the system, the shell program `/bin/bash` executes a series of startup scripts/files in the following order to set up the environment:

- System-wide configuration files
  - `/etc/profile`
  - `/etc/bashrc`
- Individual user configuration files
  - `~/.bash_profile` and `~/. bashrc`
  - `~/.bash_login`
  - `~/.profile`

The `/etc/profile` File

- Is a script maintained by the system administrator
- Is the same file for all users of a system
- Is invoked only for login shells
- Contains shell initialization information and a few commands such as setting a default `umask`

The `/etc/bashrc` File

- Contains system-wide definitions for shell functions and aliases
- Might be referred to in the `/etc/profile` file or in individual user shell initialization files

The `.bash_profile` and `.bashrc` Files

- On user login, either of the following files is executed to configure the user environment individually:
  - `bash_profile` is executed for login shells.
  - `.bashrc` is executed for interactive nonlogin shells.
- If the files do not exist, you can create them.

You can also reference `.bashrc` from within `.bash_profile`. To do that, add the following lines to the `.bash_profile` file:

```
if [ -f ~/.bashrc ]; then
   source ~/.bashrc
fi
```

Now when you log in to your machine, `.bashrc` will be called.

The `bash_login` file

- Is read in the absence of the `~/.bash_profile` file
- Contains user profile and references to programs that get executed every time a user logs in

The `.profile` file

- Is read in the absence of the `~/.bash_profile` and `~/.bash_login` files
- Resides in the user home directory
- Is run each time a user logs in
- Does not exist by default
- Contains environment variables, such as a user's preferred editor, the default printer, and the application software location

Modifying a Configuration File

- Any changes to `/etc/profile` or `~.profile` files are not read by the shell until the next time you log in.
- To avoid having to log in again, use the `dot` command to tell the shell to reread the `.profile` file.

```
$ . $HOME/.profile
```


By invoking a script with the `dot` command, you tell the shell (parent) to read the commands in the *script_name* file and execute them without spawning a child process. Any definitions in the script then become part of the present environment.

Most shell scripts execute in a private environment. Any variables they set are not available to other scripts. The shell (or script) that invokes a script is called a parent, and the script that is being invoked is called the child. Variables are not inherited unless they are exported with the `export` command.

Note: You learn more about variables in the next topic.

Quiz

Which of the following files is executed for interactive, nonlogin shells?

a. `~/.bash_profile`
b. `~/.bashrc`
c. `~/.bash_login`
d. `~/.profile`

Answer: b

Agenda

- Explaining the role of startup scripts in initializing the shell environment
- Describing the various types of shell variables
- Explaining command-line parsing in a shell environment

Shell Variables

- The shell maintains two types of shell variables:
  - Local
  - Environment (Global)
- Apart from these, the shell also supports the following types of variables:
  - Reserved
  - Special

Local Variables

- Variables that are available only in the current shell are known as local variables.
- To display a list of all variables, use the `set` built-in command without any options.
- To create local variables, use the keyword `local`.

```
#!/bin/bash
HELLO=Hello
function hello {
local HELLO=World # declaring a local variable HELLO
echo $HELLO
}
```


The output of the `set` command is sorted according to the current locale and displayed in a reusable format.

In the above example, a shell or local variable called `HELLO` is declared.

Environment Variables

- Variables that are available to all shells are known as environment or global variables.
- To display all environment variables for a user in a shell, use the `printenv` or `env` command.

```
$ printenv
SHELL=/bin/bash
HISTSIZE=100
SSH_TTY=/dev/pts/1
HOME=/oracle
LOGNAME=oracle
CVS_RSH=ssh
```

Useful Bash Environment Variables

The following table contains some common environment variables that can be used in your bash shell scripts:

Variable	Purpose
HOME	Home directory of the current user
HOST	Current host name
LANG	Determine the language to communicate between the program and user
PATH	Search path of the shell, a list of directories separated by colon
PS1	Normal prompt printed before each command
PS2	Secondary prompt printed when you execute a multiline command
PWD	Current working directory
USER	Current user

ORACLE

Variables described above in the table are set or used by bash, but other shells do not normally treat them specially.

Creating Variables

- To create and set a shell variable, use the syntax:

```
var=value
```

- Do not place spaces around the = sign.
- If the value contains spaces or special characters, use single or double quotation marks.

```
$ name="Susan B. Anthony".
```

- To display the value of a variable, use the `echo` command with a `$` immediately in front of the variable name.
- When you have finished a variable, you can release the resources with the `unset` command.

Variables:

- Are used for storing data
- Are case-sensitive
- Are capitalized by default
- Can contain a mix of characters and digits
- Have a limit of 255 characters

Example:

```
$ MYNUM=21
$ MACHTYPE=sparc
$ echo $MYNUM
21
$ echo $MYNUM $MACHTYPE
21 sparc
$ unset MYNUM
$ echo $MYNUM $MACHTYPE
sparc
```

Exporting Variables

- A variable created in the shell is a local variable and is only available to the current shell.
- To pass variables outside the current shell, you need to export them by using the `export` built-in command.

```
export variable_name1 variable_name2 ...
```

- To set and export the variable at the same time, use:

```
export variable_name=value
```

Note: A subshell can change the value of a variable that it inherits from its parent. The change does not affect the value of the variable in the parent shell. Thus, subshells cannot alter values of variables in the parent shell.

The following example creates two variables in the current shell, `x` and `name`. The `name` variable is then exported to be inherited by any subshells of the parent.

```
$ x=25
$ name="Marion Morrison"
$ echo $x $name
25 Marion Morrison
$ export name
```

Reserved Variables

- Reserved variables are words that have a special meaning to the shell and are set by the shell.
- Examples:

```
! case do done elif else esac fi for function if in
select then until while { } time [[ ]]
```

- Be careful about changing the values of these variables as it might impact your interaction with the shell.

Note: For a complete list of reserved variables, read the `man` page for `bash`.

Reserved Variables: Bash Shell

Variable	Meaning
HOME	User's login directory path
IFS	Internal field separators
LOGNAME	User's login name
MAILCHECK	Duration, the mail daemon checks for mail
OPTIND	Used by the `getopts` statement during parsing of options
PATH	Search path for commands
PS1	The prompt, defaults to `$`
PS2	Prompt used when command-line continues, defaults to `>`
PS3	Prompt used within a select statement, defaults to `#?`
PS4	Prompt used by the shell debug utility, defaults to `+`
PWD	Current working directory
SHELL	Shell defined for the user in the `passwd` file
TERM	Terminal type, which is used by the `vi` editor and other commands

Special Variables

- Several shell variables are available to users.
- The shell sets the variable value at process creation or termination.

Variable	Purpose
$	Contains the process identification number of the current process
?	Contains the exit status of the most recent foreground process
!	Contains the process ID of the last background job started

The table displays the special variables used in the bash shell.

Process Identification Variable

The $ shell variable is available to users. It contains the current process ID number. Use this variable to create file names that are unlikely to already exist.

```
$ echo $$
8873
$ sh
$ echo $$
17716
$ exit
$ echo $$
8873
```

Exit Status

The exit status is the integer value of the last executed command (program, script, or shell statement). The `?` variable provides the exit status of the most recent foreground process. It:

- Determines whether the process ran successfully
- Sets the `?` variable to `0` if a process succeeded and to a nonzero value if it did not succeed

Think of this as equating success with zero errors occurring and equating failure with one or more errors occurring or the command not being able to do what was requested.

```
$ grep "root" /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
$ echo $?
0

$ grep "rot" /etc/passwd
$ echo $?
1
```

Background Process Identification

A shell allows you to run a command in the background by using the `&` operator. To find the process identification number (PID) of the last background job started, echo the `!` variable.

```
$ date &
1175
$ Mon Oct 12 06:23:11 IST 2009

$ echo $!
1175
```

Quiz

The `set` command can be used to display all environment variables for a user in a shell.

a. True
b. False

Answer: b

Agenda

- Explaining the role of startup scripts in initializing the shell environment
- Describing the various types of shell variables
- Explaining command-line parsing in a shell environment

Command-Line Parsing

- Parsing controls how a shell processes a command-line before it executes a command.
- The parse order for the bash shell is as sequenced below:
  1. Aliases
  2. Built-in commands
  3. Functions
  4. Tilde expansions
  5. Parameter expansions
  6. Command substitutions
  7. Arithmetic expansions
  8. Quoting characters

ORACLE

A shell has a predefined order in which it interprets or evaluates different tokens, such as metacharacters, spaces, and keywords, on the command line. Based on rules built into it, a shell interprets tokens in a specific order (the parse order) and then executes the command.

Note: The following slides provide more information about each of the tokens in the shell parse order.

Aliases

- An alias is a way of assigning a simple name to what might be a complicated command or series of commands.
- Aliases hold commands, whereas variables hold data.
- Aliases can specify a version of a command when more than one version of the command is on the system.
- Aliases are created and listed with the `alias` command.
- Aliases are removed with the `unalias` command.

An alias is a name that you give to a command. You can create an alias in the bash shell. Aliases are not available in the Bourne shell. Commands, scripts, or programs that require a user to perform a lot of typing to execute them are good alias candidates. Other reasons for using aliases may include the following:

- Several versions of a command exist on a system, and you want to use a particular one by default. You can create an alias that lists the entire path name to that command:

```
alias mycommand=/fullpathname/cmd
```

- You frequently spell a command incorrectly, so you make an alias of the incorrect spelling:

```
alias mroe=more
```

- You set up default options for a command:

```
alias dk='df -k'
```

Alias Inheritance

- Aliases are not inherited by subshells.
- To allow new bash shells to know about the aliases, place the aliases in the `$HOME/.bashrc` file.

Note: Aliases created on the command line are known only in the current shell.

```
$ alias lpgut='lp -d guttenberg'
$ alias lpgut
lpgut='lp -d guttenberg'
$ bash
$ alias lpgut
lpgut alias not found
$ exit
$ alias lpgut
lpgut='lp -d guttenberg'
$ unalias lpgut
$ alias lpgut
lgput alias not found
```

Built-in Aliases

The following are some of the built-in aliases available in the bash shell:

```
alias ls='ls -aF --color=always'
alias ll='ls -l'
alias search=grep
alias mcd='mount /mnt/cdrom'
alias ucd='umount /mnt/cdrom'
alias mc='mc -c'
alias ..='cd ..'
alias ...='cd ../..'
```

Built-in Commands

- Each shell also comes with its own set of built-in commands.
- These commands are local to the particular shell.
- Some useful built-in commands in the bash shell are:
  - `export`
  - `set/unset`
  - `eval`
  - `let`

Many UNIX commands may be well known to users but some others may not be so well known. Examples of well-known commands are `ls`, `cd`, `grep`, `find`, and `chmod`. These commands might vary in syntax and usage from one UNIX platform to another. They are shell-independent.

Each shell also comes with its own set of built-in commands. These commands are local to the particular shell (for example, the `history` command in the C shell and the `export` command in the bash shell). Built-in commands are platform-independent.

For more information about other built-in commands, refer to the `man` page of `bash`.

The `export` Command

- The `export` command exports a shell variable to the environment.
- Syntax:

```
$ export VARNAME=value
```

- Example:

```
$ export country=India
$ env
SESSIONNAME=Console
country=India
```


In the example in the slide, you can see that `env` displays the exported variables.

The `export -p` command also displays all the exported variables in the current shell.

```
# export -p
declare -x EDITOR="vi"
declare -x ENV="/.bashrc"
declare -x HOME="/root"
declare -x HZ=""
declare -x LANG="en_US.UTF-8"
declare -x LOGNAME="root"
declare -x MAIL="/var/mail/root"
declare -x OLDPWD
declare -x PAGER="/usr/bin/less -ins"
declare -x PATH="/usr/bin:/usr/sbin"
declare -x PWD="/root"
declare -x SHELL="/usr/bin/bash"
declare -x SHLVL="2"
declare -x TERM="xterm"
declare -x country="India"
```

The `set` Command

- The `set` command is used to set and modify shell options.
- The `set` command without any argument lists all the variables and its values.

```
$ set +o history # To disable the history storing
                 # +o disables the given options

$ set -o history # -o enables the history
```

- The `set` command is also used to set the values for the positional parameters.

The `unset` Command

The `unset` command is used for:

- Setting the shell variable to `null`
- Deleting an element of an array
- Deleting complete array values

```
$ cat unset.sh
#!/bin/bash
#Assign values and print it
var="welcome to shell scripting"
echo $var

#unset the variable
unset var
echo $var

$ ./unset.sh
welcome to shell scripting
```

The `eval` Command

- The `eval` command reads its arguments as input to the shell and executes the resulting commands.
- `eval` is usually used to execute commands generated as a result of command or variable substitution.

```
$ eval whence -v ps
ps is /usr/bin/ps
```

The `let` Command

The `let` command is used to perform arithmetic operations on shell variables.

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

let arg1=12
let arg2=11

let add=$arg1+$arg2
let sub=$arg1-$arg2
let mul=$arg1*$arg2
let div=$arg1/$arg2
echo $add $sub $mul $div

$ ./mathlet.sh
23 1 132 1
```



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

let arg1=12
let arg2=11

let add=$arg1+$arg2
let sub=$arg1-$arg2
let mul=$arg1*$arg2
let div=$arg1/$arg2
echo $add $sub $mul $div

$ ./mathlet.sh
23 1 132 1
```

Shell Functions

- Functions facilitate grouping of several commands under a single name.
- Functions can be executed later as and when required as a regular UNIX command.
- Functions allow control-flows, arguments, and few other features missing in aliases.
- Syntax:

```
function functionname()
{
commands
...
}
```


Syntax:

```
function functionname()
{
commands
...
}
```

Where:

- `function` is a keyword (optional).
- *`functionname`* is the name of a function.
- `commands` is a list of commands to be executed in the functions.

Note: You learn more about functions in the lesson titled “Functions.”

Shell Functions

```
$ function lsext()
{
find . -type f -iname '*.'${1}'' -exec ls -l {} \; ;
}

$ lsext txt
-rw-r--r-- 1 root root   75 Jan 15 12:00 Rehearse.txt
-rw-r--r-- 1 root root  296 Sep 23 10:35 Star12.txt
-rw-r--r-- 1 root root  452 Nov 25 13:03 Star19.txt
-rw-r--r-- 1 root root 3927 Apr  7 09:10 StarDB.txt
```


The example in the slide shows a function to display a long list of files with the given extension. The function `lsext` is used to find the list of files in the current directory, which has the given extension. This function uses a combination of `find` and `ls` commands.

Tilde Expansion

- Variables that are prefixed with '~' (named tilde) are called tilde expansions.
- You can use ~ at the beginning of words as follows:

Tilde Expansion	Purpose
~	Value of $HOME
~/ foo	Full path name of current user's (foo) home directory
~username	Full path name of username's home directory
~+	Full path name of a working directory
~-	Previous working directory
-	Previous working directory

Examples: The `cd` command accepts a hyphen to change the working directory to the previous working directory.

```
$ pwd
/users/oracle/opt
$ cd ../course/lesson4
$ pwd
/users/oracle/course/lesson4
$ cd -
/users/oracle/opt
```

Command Substitution

- Command substitution allows the output of one command to be used as an argument to another command.
- The bash shell supports `$(command)` and ` `` ` (back quotation marks).
- Example:

```
#!/bin/bash
DATE=`date`
echo "Date is $DATE”
```

Arithmetic Expansion

- Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result.
- Arithmetic expansion:
  - Assigns only string values to variables
  - Has no built-in arithmetic
  - Has external statement `expr` that treats the variables as numbers and performs arithmetic operations
- Syntax:

```
$(( expression ))
```


Note: The `expr` statement is space-sensitive and expects each operand and operator to be separated by white spaces.

Arithmetic Operations

Operator	Operation	Example
+	Addition	num2=`expr "$num1" + 25`
-	Subtraction	num3=`expr "$num1" - "$num2"`
*	Multiplication	num3=`expr "$num1" \* "$num2"`
/	Division	num4=`expr "$num2" / "$num1"`
%	Integer remainder	num5=`expr "$num1" % 3`
Note: Precede a multiplication operator with a backslash because it is a shell metacharacter.		

The bash shell arithmetic operations are `+`, `-`, `*`, `/`, and `%`. The `expr` statement recognizes the operators shown in the table in the slide.

Examples:

```
$ num1=5
$ echo $num1
5
$ num2=$num1+10
$ echo $num2
5+10
```

The `expr` statement must have white spaces around operators and operands. The `expr` statement automatically sends the value to standard output:

```
$ echo $num1
5
$ expr $num1 + 7
12
```

```
$ expr $num1+7
5+7
$ echo $num1
5
```

The following examples show the need for properly quoting the multiplication operator.

```
$ expr $num1 * 2
expr: syntax error
$ expr "$num1 * 2"
5 * 2
$ expr $num1 \* 2
10
```

When performing an arithmetic operation, the result is stored in a variable. This is accomplished in the bash shell by placing the `expr` statement on the right side of a variable assignment statement:

```
$ echo $num1
5
$ num2=`expr $num1 + 25`
$ echo $num2
30
$ num3=`expr $num1 + $num2`
$ echo $num3
35
$ num4=`expr $num1 \* $num2`
$ echo $num4
150
$ num5=`expr $num2 / $num1`
$ echo $num5
6
$ num6=`expr $num1 % 3`
$ echo $num6
2
```

Arithmetic Precedence

- In a bash shell script, operations execute in the order of precedence.
- The higher precedence operations execute before the lower precedence ones.
  1. Expressions within parentheses are evaluated first.
  2. `*`, `%`, and `/` have greater precedence than `+` and `-`.
  3. Everything else is evaluated from left to right.
- When there is the slightest doubt, use parentheses to force the evaluation order.

Arithmetic Bitwise Operations

Operator	Operation	Example	Result
#	Base	`2#1101010 or 16#6A`	`10#106`
<<	Shift bits left	`((x = 2#11 << 3))`	`2#11000`
>>	Shift bits right	`((x = 2#1001 >> 2))`	`2#10`
&	Bitwise AND	`((x = 2#101 & 2#110))`	`2#100`
|	Bitwise OR	`((x = 2#101 | 2#110))`	`2#111`
^	Bitwise exclusive OR	`(((x = 2#101 ^ 2#110))`	`2#11`

ORACLE

The table in the slide lists the six bitwise operators:

- The # operator designates the base of the value.
- The << operator performs a binary shift left.
- The >> operator performs a binary shift right.
- The & operator ANDs two binary numbers together.
- The | operator ORs two binary numbers together.
- The ^ operator exclusively ORs two binary numbers together.

Note: If you are not familiar with bitwise operations, examine the values in binary rather than in decimal.

The # operator designates the base of the value that follows it. For example, 2#101 means that the value 101 is a base 2 (binary) value (2#101 would be 5 in decimal base).

The << operator performs a binary shift left by as many bits as are indicated by the number that follows the operator. The expression 2#10 << 1 yields the value 2#100. The expression 2#10100<< 2 yields the value 2#1010000.

```
-------------------------------------------------------------------------------------------------------------
Initial value                    0     1     0     0
<< 2            0     1     0     0
Result                0     1     0     0     0     0
-------------------------------------------------------------------------------------------------------------
```

Vacated positions are padded with 0 or 1 based on whether the number is positive or negative, respectively.

The >> operator performs a binary shift right by as many bits as are indicated by the number that follows the operator. The expression 2#10 >> 1 yields the value 2#1. The expression 2#10100 >> 2 yields the value 2#101.

The & operator ANDs two binary numbers together. This means that the AND operation is performed on the corresponding digit of 0 or 1 in each number, resulting in a 1 if both digits are 1; otherwise, the result is 0.

```
-------------------------------------------------------------------------------------------------------------
Initial value        1     1     0     0
&               1     0     1     0
Result               1     0     0     0
-------------------------------------------------------------------------------------------------------------
```

The | operator *OR*s two binary numbers together. This means that the OR operation is performed on the corresponding digit of 0 or 1 in each number, resulting in a 1 if either digit is a 1; otherwise, the result is 0.

```
-------------------------------------------------------------------------------------------------------------
Initial value        1     1     0     0
    |           1     0     1     0
Result               1     1     1     0
-------------------------------------------------------------------------
```

The ^ operator performs an exclusive OR on two binary numbers together. This means that an exclusive OR is performed on the corresponding digit of 0 or 1 in each number, resulting in 1 if only one of the digits is a 1. If both digits are 0 or both are 1, then the result is 0.

```
-------------------------------------------------------------------------------------------------------------
Initial value        1     1     0     0
    ^           1     0     1     0
Result               0     1     1     0
-------------------------------------------------------------------------------------------------------------
```

Arithmetic Usage

```
$ cat math.sh
#!/bin/bash
# Script name: math.sh
# This script finds the cube of a number, and the
# quotient and remainder of the number divided by 4.
y=99
(( cube = y * y * y ))
(( quotient = y / 4 ))
(( rmdr = y % 4 ))

print "The cube of $y is $cube."
print "The quotient of $y divided by 4 is $quotient."
print "The remainder of $y divided by 4 is $rmdr."
# Notice the use of parenthesis to control the order of evaluating.
(( z = 2 * (quotient * 4 + rmdr) ))
print "Two times $y is $z.""

$ ./math.sh
The cube of 99 is 970299.
The quotient of 99 divided by 4 is 24.
The remainder of 99 divided by 4 is 3.
Two times 99 is 198.
```


The example script in the slide:

1. Assigns the value `99` to the variable `y`.
2. Computes the cube of the number entered, the quotient of the number entered divided by `4`, and the remainder of the number entered divided by `4`.
3. Prints the results with an appropriate message.
4. Computes the number input by using the quotient, the divisor `4`, and the remainder.
5. Multiplies the result by `2` and saves in the variable `z`.
6. Prints a message showing the `z` value.

Quoting Characters

- With the exception of letters and digits, practically every key on the keyboard is a metacharacter.
- A metacharacter is a character that when unquoted, separates words.
- For a metacharacter to be taken literally, you must instruct the shell by using the following quoting characters:
  - Single quotation marks: Turn off the special meaning of all characters.
  - Double quotation marks: Turn off the special meaning of characters except `$`, `` ` ``, `"`, and `\`.
  - Backslash: Turns off the special meaning of the consecutive character.

A Pair of Single Quotation Marks

A pair of single quotation marks, '...', turns off the special meaning of all characters enclosed by the single quotation marks, including the \ character.

Note: A single quotation mark character is not allowed to appear within a pair of single quotation marks.

```
$ echo a           b
a b

$ echo 'a     b'
a       b

$ num=25

$ echo 'The value of num is $num'
The value of num is $num
```

A Pair of Double Quotation Marks

A pair of double quotation marks (as in "...") turns off the special meaning of all characters enclosed by the double quotation marks, except for the four characters $, `, ", and \. Within double quotation marks, use the backslash to selectively turn off the special meaning of these four characters.

Note: A double quotation character is not allowed to appear within a pair of double quotation marks unless it is preceded by a \.

```
$ num=25
$ echo "The value of num is $num"
The value of num is 25
$ name=Roger
$ echo "Hi $name, I'm glad to meet you!"
Hi Roger, I'm glad to meet you!
$ echo "Hey $name, the time is `date`"
Hey Roger, the time is Fri Jul 9 16:52:50 PDT 1999
```

Backslash

If you need to have the special meaning of any character turned off and the character to have its literal value, precede it with the backslash (\) character, which is often called an escape character. Use the backslash to turn off the special meaning of any shell metacharacter.

```
$ name=jessica
$ echo "Hello $name. \
> Where are you going?"
Hello jessica. Where are you going?
```

Include the backslash (\) character before $name.

```
$ echo "Hello \$name. \
> Where are you going?"
Hello $name. Where are you going?
```

Quiz

Command substitution refers to the process of controlling how a shell processes a command line before it executes a command.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Explain the role of startup scripts in initializing the shell environment
- Describe the various types of shell variables
- Explain command-line parsing in a shell environment

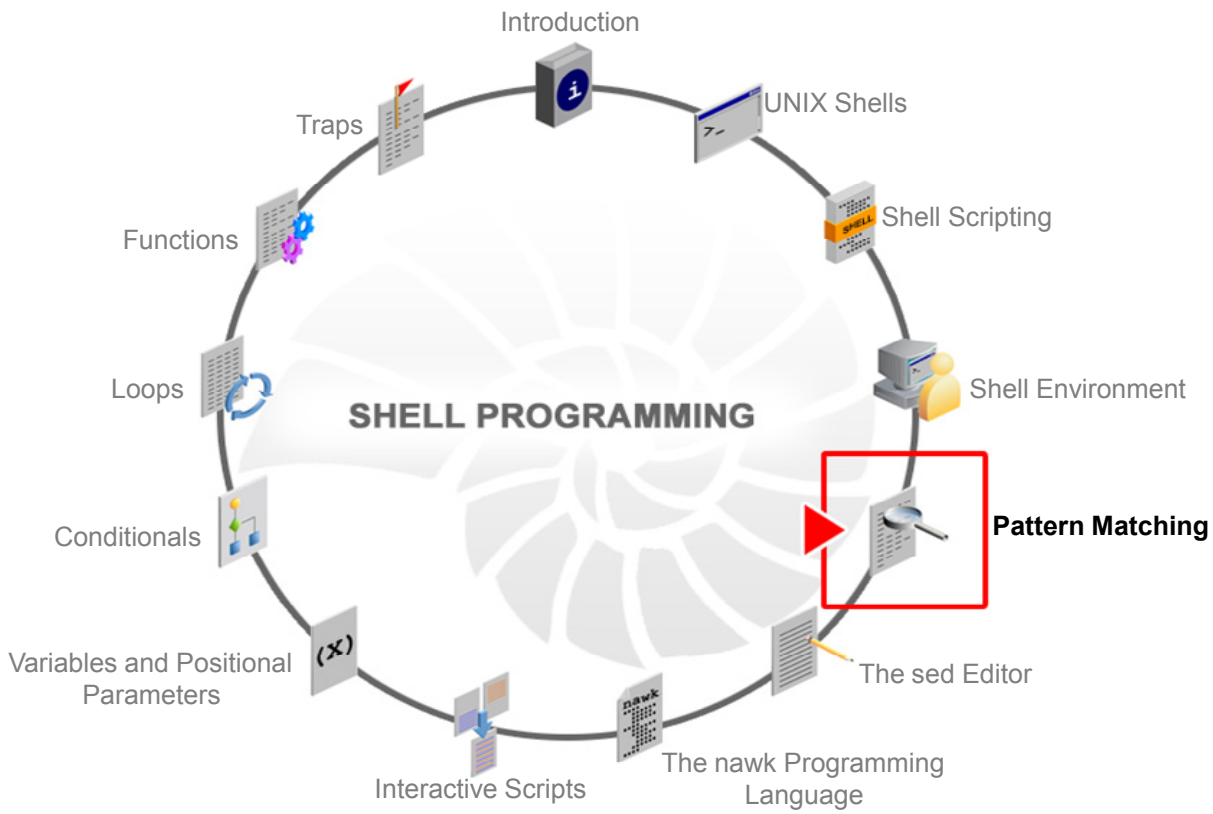
Practice 4 Overview: Shell Environment

This practice covers the following topics:

- Using the Shell and Environment Variables
- Configuring User Profiles

5

Pattern Matching



Objectives

After completing this lesson, you should be able to:

- Find patterns in a file by using the `grep` command
- Explain the role of regular expressions in pattern matching

Agenda

- Finding patterns in a file by using the `grep` command
- Explaining the role of regular expressions in pattern matching

The `grep` Command

- The `grep` command searches one or multiple text files or a pipeline for a specific pattern.
- When the pattern is found, the entire line is printed.
- The command also permits you to use regular expression characters in the search pattern.
- Syntax:

```
grep [OPTIONS] PATTERN [FILE...]
```

- Example:

```
$ ps -ef | grep msxyz
$ ps -e | grep dtterm
352 ?? 0:00 dtterm
353 ?? 0:13 dtterm
354 ?? 0:11 dtterm
1766 pts/5 0:00 dtterm
```

ORACLE

Like most UNIX commands, `grep` is a mnemonic. The `grep` mnemonic is derived from the ex editor commands. It means globally (`g`) search for a regular expression (`re`) and print (`p`) the results (`grep`). The `grep` utility searches text files for a specified pattern and prints all lines that contain that pattern. If no files are specified, `grep` assumes it will receive text from standard input.

Consider the following scenario. A user complains that the `msxyz` program on her machine has locked up her machine. She cannot get the program to halt. It is your job to find the process and stop it.

The `ps -ef` command gives you a long list of running processes. The output is probably too long to find this user's process. You want a single line or a specific process list.

In the first example in the slide, the `grep` command takes the `ps -ef` command output as its input. The `grep` command performs a search for the string `msxyz` and prints the results. This gives you specific lines to review.

Suppose that you have multiple terminal and console windows open. To see those specific processes, execute a `ps` command and search for the `dtterm` command as in the second example in the slide.

For more information about the `grep` command and its options, see the `man` page.

The `grep` Options

Option	Function
`-b`	Display the block number at the beginning of each line
`-c`	Display the number of matched lines
`-h`	Display the matched lines, but do not display the file names
`-i`	Ignore case sensitivity
`-l`	Display the file names, but do not display the matched lines
`-n`	Display the matched lines and their line numbers
`-s`	Silent mode
`-v`	Display all lines that do NOT match
`-w`	Match whole word

ORACLE

The table in the slide shows various `grep` command options. These options modify `grep` behavior.

The `-i` option specifies that `grep` ignore the case of the letters in the search pattern. In the following example, `grep` does a non-case-sensitive search for the pattern `the`.

```
$ grep -i 'the' /etc/default/login
# Set the TZ environment variable of the shell.
# ULIMIT sets the file size limit for the login. Units are disk
blocks.
# The default of zero means no limit.
# ALTSHELL determines if the SHELL environment variable should be
set
# PATH sets the initial shell PATH variable
<output truncated>
```

Compare the previous output to the output of the following command, which prints only the lines of text that match the pattern `The`.

```
$ grep 'The' /etc/default/login
# The default of zero means no limit.
# bad password is provided. The range is limited from
# The SYSLOG_FAILED_LOGINS variable is used to determine how many
  failed
```

The `-c` option counts the number of lines that match the pattern. It then prints the count and not the actual lines that matched the pattern.

```
$ grep -ci 'the' /etc/default/login
19
$ grep -c 'The' /etc/default/login
3
```

Use the `-l` option to:

- Search for a string in many files
- Have the output list only the files in which the string is found

The `-l` option is often useful when you want to feed the output of `grep` to another utility to process a list.

```
# grep -l 'grep' /etc/init.d/*
/etc/init.d/apache
/etc/init.d/cachefs.daemon
/etc/init.d/dhcp
/etc/init.d/dodatadm.udaplt
/etc/init.d/dtlogin
/etc/init.d/imq
/etc/init.d/init.wbem
/etc/init.d/ncakmod
/etc/init.d/swupboots
```

To find a search pattern in a large file, print the line number before each match by using the `-n` option. This is useful when you are editing files.

```
# grep -n 'user' /etc/passwd
18:user1:x:100:10::/export/home/user1:/bin/sh
19:user2:x:101:10::/export/home/user2:/bin/ksh
```

The `-v` option prints lines that do not contain the search pattern.

```
# grep -v 'root' /etc/group
staff::10:
sysadmin::14:
smmsp::25:
gdm::50:
webservd::80:
postgres::90:
nobody::60001:
noaccess::60002:
nogroup::65534:
```

Agenda

- Finding patterns in a file by using the `grep` command
- Explaining the role of regular expressions in pattern matching

Regular Expression

- A regular expression (RE) is a character pattern that matches the same characters in a search.
- Regular expressions:
  - Allow you to specify patterns to search in text
  - Provide a powerful way to search files for specific pattern occurrences
  - Give additional meaning to patterns through metacharacters

Regular Expression Metacharacters

Metacharacter	Function
\	Escapes the special meaning of an RE character
^	Matches the beginning of the line
$	Matches the end of the line
\<	Matches the beginning of the word anchor
\>	Matches the end of the word anchor
[]	Matches any one character from the specified set
[-]	Matches any one character in the specified range
*	Matches zero or more of the preceding character
.	Matches any single character
\{\}	Specifies the minimum and maximum number of matches for a regular expression

ORACLE

When you use regular expression characters with the `grep` command, enter quotation marks around the pattern. Some regular expression characters used by `grep` are also metacharacters to one or more shells, and a shell might use a metacharacter as a file name metacharacter. Use single (') quotation marks to hide metacharacters from a shell. The metacharacter are displayed in the table in the slide.

Regular Expressions: Example

```
$ ps -ef | grep '[A-Z]'
ID       PID      PPID     C STIME          TTY      TIME CMD
root     647      1        0 06:14:45       ?        0:00
/usr/lib/dmi/snmpXdmid -s sls-s10-host
noaccess 797      1        0 06:15:03       ?        1:34
/usr/java/bin/java -server -Xmx128m -
XX:+UseParallelGC -XX:ParallelGCThreads=4
root     708      704      4 06:14:50       ?        5:22
/usr/X11/bin/Xorg :0 -depth 24 -nobanner -auth/var/dt/A:0-9Aaayb
root     813      739      0 06:15:16       ?        0:00 /bin/ksh
/usr/dt/bin/Xsession
root     905      903      0 06:15:27       pts/2    0:00 -sh -c unset
DT; DISPLAY=:0; /usr/dt/bin/dtsession_res -merge
root     1045     1        1 06:15:51       ?        1:10
/usr/lib/mixer_applet2 --oaf-
activateiid=OAFIID:GNOME_MixerApplet_Factory --oa
root     1050     1        0 06:15:52       ?        0:01
/usr/lib/notification-area-applet --oafactivate-
iid=OAFIID:GNOME_NotificationA
root     1440     1284     0 08:20:35       pts/4    0:00 grep [A-Z]
<output truncated>
```

ORACLE

Using a regular expression, you can search the current process table (and header) for any process that contains a capital letter. You can do this by using the given range as the pattern to the `grep` command, as in the example.

For instance, if you are only interested in current processes that contain the capital letter `A` in the line, you can limit the pattern to specify that character.

```
$ ps -ef | grep 'A'
root     708   704   7     06:14:50   ?     5:26 /usr/X11/bin/Xorg :0
   -depth 24 -nobanner -auth /var/dt/A:0-9Aaayb
root     905   903   0     06:15:27   pts/2 0:00 -sh -c unset DT;
   DISPLAY=:0; /usr/dt/bin/dtsession_res -merge
root     1442  1284  0     08:21:17   pts/4 0:00 grep A
<output truncated>
```

Escaping a Regular Expression

- A \ (backslash) character escapes the regular expression characters.
- The \ interprets the next character literally, not as a metacharacter.
- Thus, a \$ matches a dollar sign and a \. matches a period.

Escaping a Regular Expression: Example

```
# grep '$' /etc/init.d/nfs.server
#!/sbin/sh
#
# Copyright 2009 Oracle, Inc. All rights reserved.
# Use is subject to license terms.
#
#ident "@(#)nfs.server 1.43 04/07/26 SMI"
# This service is managed by smf(5). Thus, this script provides
# compatibility with previously documented init.d script behaviour.
case "$1" in
'start')
        svcadm enable -t network/nfs/server
        ;;
'stop')
        svcadm disable -t network/nfs/server
        ;;
*)
        echo "Usage: $0 { start | stop }"
        exit 1
        ;;
esac
```


The example in the slide shows `$` as a regular expression character that matches the end-of-line character. The output from the example contains all the lines from the script because `$` matches the end-of-line character for each line in the script. This is verified by using the `wc` command.

```
$ grep '$' /etc/init.d/nfs.server | wc -l
24
```

You can also use `grep -c` to print only a count of the lines that contain the pattern.

```
$  grep -c '$' /etc/init.d/pppd
74
```

To display only the lines from the `nfs.server` boot script that contain the literal character `$`, you can hide its special meaning by preceding the `$` character with the `\` regular expression.

```
$ grep '\$' /etc/init.d/nfs.server
case "$1" in
echo "Usage: $0 { start | stop }"
$ grep '\$' /etc/init.d/nfs.server | wc -l
2
```

Line Anchors

Line anchors force a regular expression to match only at the start or end of a line.

- Use ^ for the beginning of the line.
- Use $ for the end of the line.

An anchor is a symbol that matches a character position on a line. The ^ and $ anchors match text patterns relative to the beginning ^ or ending $ of a line of text.

Line Anchors: Example

```
$ grep 'root' /etc/group
root::0:
other::1:root
bin::2:root,daemon
sys::3:root,bin,adm
adm::4:root,daemon
uucp::5:root

<output truncated>

$ grep '^root' /etc/group
root::0:

$ grep 'mount$' /etc/vfstab
#device device mount FS fsck mount mount
```


The first example in the slide finds all lines that contain the pattern `root` in the `/etc/group` file.

If you intend to display only one entry for the `root` group in the `/etc/group` file, then the pattern must specify that the line begins with the pattern (given the syntax of the file) as in the second example in the slide.

Note that the `^` regular expression character allows you to anchor the pattern match to the beginning of the line.

Similarly, the `$` regular expression character allows you to anchor the pattern match to the end of the line. Lines print only if the specified pattern represents the characters preceding the end-of-line character as in the third example in the slide.

Word Anchors

Word anchors are used to refer to the beginning and end of a word in regular expressions.

- Use \< for the beginning of the word.
- Use \> for the end of the word.

A backslash used with an angle bracket is a word anchor. The less-than bracket (<) marks the beginning of a word. Any text that follows this bracket is matched only when it occurs at the beginning of a word. The greater-than bracket (>) marks the end of a word. Text that precedes this bracket is matched only when it occurs at the end of a word. Words are delimited by spaces, tabs, beginnings of line, ends of line, and punctuation.

Word Anchors: Example

```
$ grep '\<uucp' /etc/group
uucp::5:root

$ grep 'user' /etc/passwd
user:x:100:1::/home/user:/bin/sh
user2:x:101:1::/home/user2:/bin/sh
user3:x:102:1::/home/user3:/bin/sh

$ grep '\<user\>' /etc/passwd
user:x:100:1::/home/user:/bin/sh
```


For example, if you wanted to print the group file entry for the `uucp` group, issuing the `grep` command without regular expression characters gives you the `uucp` and `nuucp` group entries. The first example command in the slide, however, should give only the single group entry for `uucp`.

Ensure that you use both word anchors at the same time to ensure that your pattern is a complete word by itself, rather than a substring of another word.

Note the output for the pattern `user` in the `/etc/passwd` file as in the second example in the slide. The output includes lines with `user` as a substring of words, such as `user2` and `user3`.

If you were searching for the specific user named `user`, you should use both word anchors (or the `-w` option) as in the third example in the slide.

Character Classes

A character class makes one small sequence of characters match a larger set of characters, such as the following:

- `[abc]` finds a single character in the class.
- `[a-c]` finds a single character in the range.
- `[^a-c]` finds a single character not in the range.

A string enclosed in square brackets specifies a character class. Any single character in the string is matched. For example, the `grep '[abc]' frisbee` command displays every line that contains an `a`, `b`, or `c` in the `frisbee` file.

Character Classes: Example

```
$ grep '[iu]' /etc/group
bin::2:root,daemon
sys::3:root,bin,adm
uucp::5:root
mail::6:root
nuucp::9:root
sysadmin::14:
nogroup::65534:
$ grep '[u-y]' /etc/group
sys::3:root,bin,adm
uucp::5:root
tty::7:root,adm
nuucp::9:root
sysadmin::14:
webservd::80:
nobody::60001:
nogroup::65534:
$ grep '\<[Tt]he\>' teams
The teams are chosen randomly.
```


The first example command prints lines from the `/etc/group` file that contain either the letter `i` or the letter `u`.

You can also specify a range of characters as in the second example, which prints lines that contain at least one of the specified characters in the range.

Given the content of the `teams` file, the third example shows how character classes can be used to find the word `the` or `The` in any line in the `teams` file.

```
$ cat teams
Team one consists of
Tom
Team two consists of
Fred
The teams are chosen randomly.
Tea for two and Dom
Tea for two and Tom
```

Single Character Match

The . (dot) regular expression matches any one character except the newline character.

```
$ grep 'c...h' /usr/dict/words

$ grep '^c...h' /usr/dict/words

$ grep '^c...h$' /usr/dict/words
```


The first example in the slide looks for all lines containing `c`, followed by any three characters, followed by `h`.

The second command example looks for all lines that do not have a character before c; that is, `c` is the first character, followed by any three characters, followed by `h`.

The third command example looks for all lines that do not have a character before `c`, followed by any three characters, followed by `h`, which is the end of the word; that is, five-letter words that begin with `c` and end with `h`.

Character Match by Specifying a Range

The \{ and \} expressions allow you to specify the minimum and maximum number of matches for a regular expression.

```
$ cat test
root
rooot
roooot
rooooot

$ grep 'ro\{3\}t' test
rooot

$ grep 'ro\{2,4\}t' test
root
rooot
roooot
```


The examples in the slide show the use of the \{ and \} expressions.

Closure Character (*)

- The * symbol, when used in a regular expression, is termed a closure.
- * matches the preceding character zero or more times.

```
$ grep 'Team*' teams
Team one consists of
Team two consists of
Tea for two and Dom
Tea for two and Tom
$ grep '\<T.*m\>' teams
Team one consists of
Tom
Team two consists of
Tea for two and Dom
Tea for two and Tom
$ grep '*' teams
$ grep 'abc' *
data1:abcd
```


The * symbol in the first example matches the preceding character zero or more times.

The second example finds all lines that contain a word beginning with `T` and a word ending with `m`.

An asterisk (*) has special meaning only when it follows another character. If it is the first character in a regular expression or if it is by itself, it has no special meaning. The third example searches for lines containing a literal asterisk within the file called `teams`.

The asterisk has another meaning outside of the regular expression. The last command example searches all files in the current directory for the string `abc`. In this example, the * is a shell metacharacter rather than a `grep` metacharacter.

The `egrep` Command

- The `egrep` (extended `grep`) command searches a file or a pipeline for a pattern by using full regular expressions.
- Example:

```
# grep "two | team" teams

# egrep "two | team" teams
Team two consists of
The teams are chosen randomly.
Tea for two and Dom
Tea for two and Tom
```

ORACLE

Quiz

Which of the following forces a regular expression to match only at the end of a line?

a. ^
b. /
c. $
d. *

Answer: c

Summary

In this lesson, you should have learned how to:

- Find patterns in a file by using the `grep` command
- Explain the role of regular expressions in pattern matching

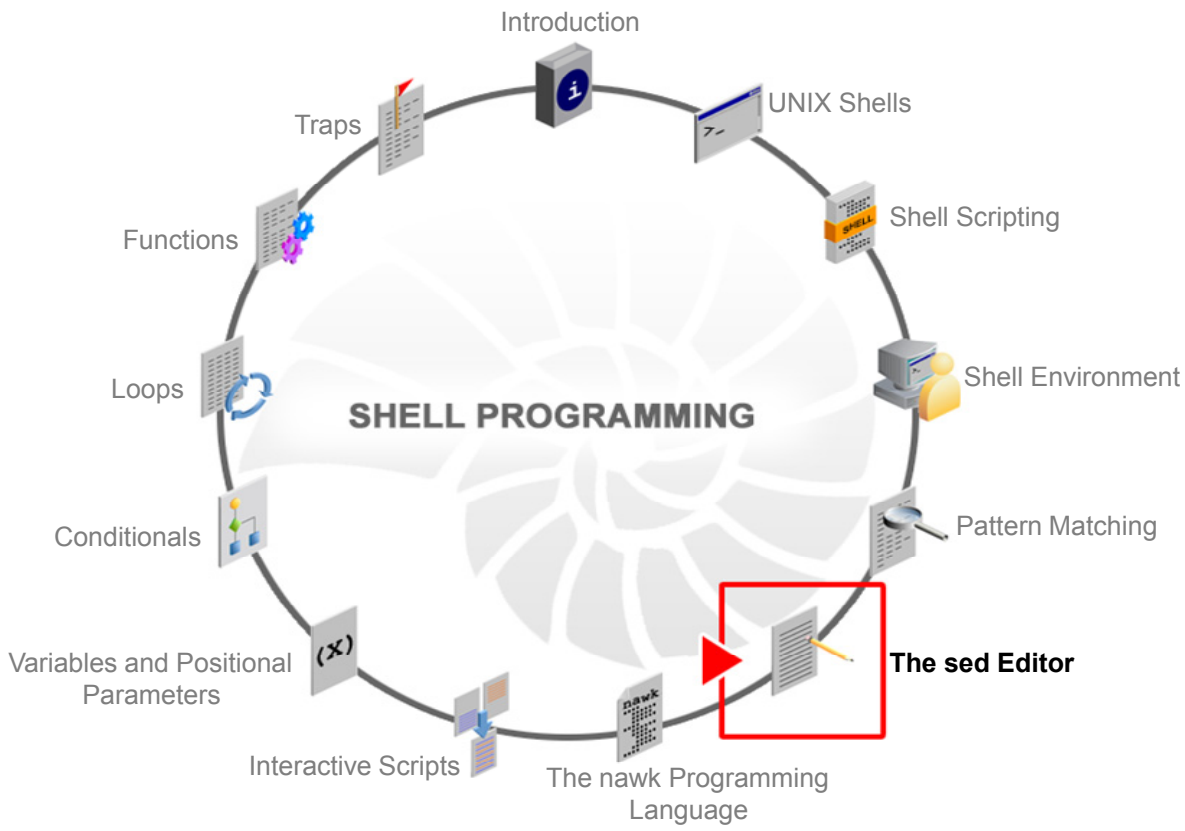
Practice 5 Overview: Pattern Matching

This practice covers the following topics:

- Using Regular Expressions and the `grep` Command
  - You use regular expressions to search for a pattern.
  - You use the `grep` command to search for specific string within text files.

6

The sed Editor



ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe the `sed` editor
- Perform noninteractive editing tasks by using the `sed` editor

Agenda

- Describing the `sed` editor
- Performing noninteractive editing tasks by using the `sed` editor

Introduction to the `sed` Editor

- The term `sed` stands for stream editor, which:
  - Is nondestructive
  - Is noninteractive
  - Uses regular expressions
- Syntax:

```
sed [options] '[addresses] action [args]' files [ > outfile]
```

Note: The `sed` syntax allows for an input file to be specified on the command line, whereas the output file specification can be accomplished through output redirection.

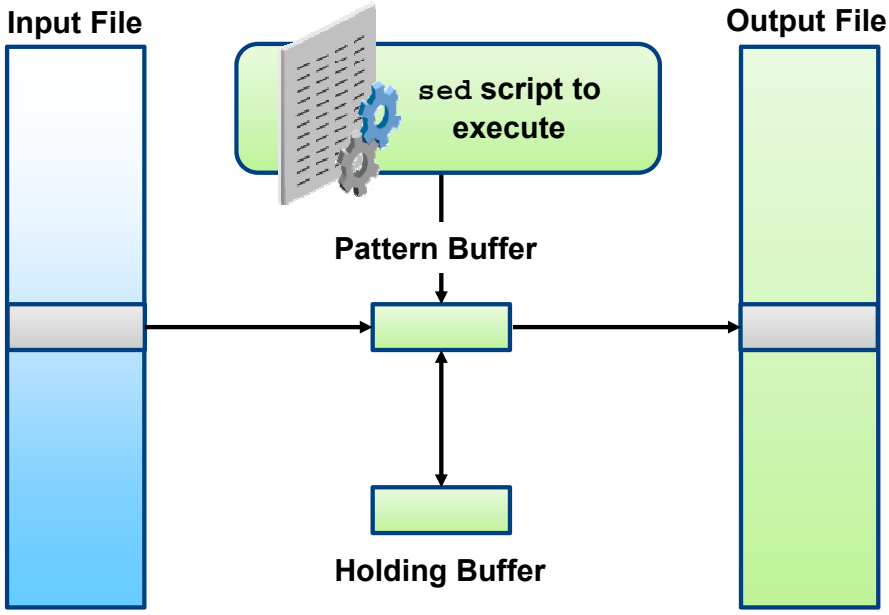
A stream is a source or destination for bytes, which means `sed` can take its input from standard in, apply the requested edits on the stream, and automatically put the results to standard out. The `sed` syntax allows for an input file to be specified on the command line.

You do not need to interact with the `sed` editor while it is running; therefore, it has also been termed a batch editor. This is in contrast to such editors as `vi` and `ed`, which are interactive. `sed` does not require interaction; therefore, you can place `sed` commands in a script. You can call the script file and run it against the data file to perform repetitive editing operations.

The `sed` editor is capable of performing text-pattern substitutions and text-pattern deletions by using regular expression syntax. These are the same regular expression characters used by `grep`.

The `sed` command offers capabilities that are an extension of interactive text editing. If you need to search and replace text strings in a large number of files, `sed` is most useful.

How `sed` Works



As shown in the diagram in the slide, the `sed` editor edits a file by reading the file line-by-line into a pattern buffer, modifying the line, and then outputting the buffer to standard out (`stdout`), which can be redirected to another file. The original file is not modified.

The holding buffer is used for advanced operations. It is limited to a `copy`, `append`, `compare`, or `retrieval` command. You can use the holding buffer to find duplicate lines in a sorted input file or to concatenate multiple lines together for future output.

Editing Commands

- The `sed` commands:

Command	Function
`d`	Deletes a line or lines
`p`	Prints a line or lines
`r`	Reads a file
`s`	Substitutes one string for another
`w`	Writes to a file

- The `sed` options:

Option	Function
`-n`	Suppresses the default output
`-f`	Reads `sed` commands from a script file

ORACLE

The `sed` editor uses editing commands that are similar to those you would use for `vi` and `ed`. The `sed` command has two options as displayed in the table in the slide.

Addressing

- The `sed` editor processes all lines of the input file unless you specify an address.
- The address can be in any of the following forms:
  - A single line number
  - A range of line numbers
  - A regular expression
    - `$` for last line of file
    - `/` (forward slash) to delimit a regular expression
  - A combination of range and regular expression

When you specify a line number or a range of line numbers, the line numbers represent lines from the input file against which the specified changes are applied. All other lines from the input file are displayed, unchanged, to standard out.

Similarly, when a pattern, which may contain a regular expression, is used to select lines, only lines containing the pattern are edited. All other lines from the input file are displayed, unchanged, to standard out.

If two patterns are specified, this creates a range, beginning with the first line of the file containing the first pattern, including all subsequent lines of the file up to and including the one containing the second pattern or the end of the file.

Note: A `$` (dollar character) represents the last line of a file when used in an address. The `$` character represents the end of line (EOL) when used in a regex.

Agenda

- Describing the `sed` editor
- Performing noninteractive editing tasks by using the `sed` editor

Performing Noninteractive Tasks

You can use `sed` to perform noniterative tasks, such as:

- Printing text
- Substituting text
- Reading from a file of new text
- Deleting text
- Reading `sed` commands from a file
- Writing output files

Printing Text

The `p` command in `sed` prints lines.

```
$ sed '3,5p' data.file
northwest   NW     Joel Craig     3.0 .98 3       4
western     WE     Sharon Kelly   5.3 .97 5       23
southwest   SW     Chris Foster   2.7 .8 2        18
southwest   SW     Chris Foster   2.7 .8 2        18
southern    SO     May Chin       5.1 .95 4       15
southern    SO     May Chin       5.1 .95 4       15
southeast   SE     Derek Johnson  5.0 .70 4       17
southeast   SE     Derek Johnson  5.0 .70 4       17
eastern     EA     Susan Beal     4.4 .8 5        20
northeast   NE     TJ Nichols     5.1 .94 3       13
north       NO     Val Shultz     4.5 .89 5       9
central     CT     Sheri Watson   5.7 .94 5       13
```

ORACLE

Given the `data.file` file as shown below, the range is specified by a starting address followed by a comma and then the ending address as in the example in the slide.

```
$ cat data.file
northwest     NW Joel Craig    3.0 .98    3     4
western       WE Sharon Kelly  5.3 .97    5     23
southwest     SW Chris Foster  2.7 .8     2     18
southern      SO May Chin      5.1 .95    4     15
southeast     SE Derek Johnson 5.0 .70    4     17
eastern       EA Susan Beal    4.4 .8     5     20
northeast     NE TJ Nichols    5.1 .94    3     13
north         NO Val Shultz    4.5 .89    5     9
central       CT Sheri Watson  5.7 .94    5     13
```

Printing Text

```
$ sed -n '3,5p' data.file
southwest   SW      Chris Foster   2.7 .8          2       18
southern    SO      May Chin       5.1 .95         4       15
southeast   SE      Derek Johnson  5.0 .70         4       17

$ sed -n '/west/p' data.file
northwest   NW      Joel Craig     3.0 .98         3       4
western     WE      Sharon Kelly   5.3 .97         5       23
southwest   SW      Chris Foster   2.7 .8          2       18

$ sed -n '/west/,/southern/p' data.file
northwest   NW      Joel Craig     3.0 .98         3       4
western     WE      Sharon Kelly   5.3 .97         5       23
southwest   SW      Chris Foster   2.7 .8          2       18
southern    SO      May Chin       5.1 .95         4       15
```

ORACLE

The default output of `sed` is each line that it reads. To suppress the default output, you can use the `-n` option as in the first example in the slide.

The second example prints all lines with the pattern `west` in it. The slash (`/`) character is used to delimit the regular expression.

The third command prints the first line containing the pattern `west`, up to and including the next line containing the pattern `southern`.

Printing Text

```
$ sed -n '/Chris/,$p' data.file
southwest   SW      Chris Foster   2.7 .8          2       18
southern    SO      May Chin       5.1 .95         4       15
southeast   SE      Derek Johnson  5.0 .70         4       17
eastern     EA      Susan Beal     4.4 .8          5       20
northeast   NE      TJ Nichols     5.1 .94         3       13
north       NO      Val Shultz     4.5 .89         5       9
central     CT      Sheri Watson   5.7 .94         5       13

$ sed -n '/^s.*5$/p' data.file
southern    SO      May Chin       5.1 .95         4       15
```


The first command in the example prints the first line containing the pattern `Chris`, up through the last line of the file.

The pattern might contain the regular expression characters used by `grep`. The second example prints all lines that begin with an `s` and end with a `5`.

Substituting Text

- The `s` command in `sed` performs search and substitution operations on the text.

```
$ sed 's/oldstring/newstring/' file
```

- The substitution operations can:
  - Substitute a new string for an old string
  - Perform global substitution by using `g`
  - Include the `oldstring` in the `newstring` by using `&`

The `s` command uses a pattern search and a literal string replacement. The replacement string characters are taken literally without metacharacter expansion.

Substituting Text

```
$ sed 's/3/X/' data.file
northwest   NW     Joel Craig     X.0 .98        3       4
western     WE     Sharon Kelly   5.X .97        5       23
southwest   SW     Chris Foster   2.7 .8  2      18
southern    SO     May Chin       5.1 .95        4       15
southeast   SE     Derek Johnson  5.0 .70        4       17
eastern     EA     Susan Beal     4.4 .8  5      20
northeast   NE     TJ Nichols     5.1 .94        X       13
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       1X
$ sed 's/3/X/g' data.file
northwest   NW     Joel Craig     X.0 .98        X       4
western     WE     Sharon Kelly   5.X .97        5       2X
southwest   SW     Chris Foster   2.7 .8  2      18
southern    SO     May Chin       5.1 .95        4       15
southeast   SE     Derek Johnson  5.0 .70        4       17
eastern     EA     Susan Beal     4.4 .8  5      20
northeast   NE     TJ Nichols     5.1 .94        X       1X
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       1X
```

ORACLE

In the first example, the first occurrence of the old string is substituted with the new string, leaving subsequent occurrences of the old string within the same line unchanged.

The second example shows the `g` (global) command with the `s` command, and it replaces all occurrences of the old string with the new string.

Substituting Text

```
$ sed -n '/ [0-9]$/p' data.file
northwest     NW       Joel Craig       3.0 .98  3        4
north         NO       Val Shultz       4.5 .89  5        9

$ sed 's/ [0-9]$/& Single Digit/' data.file
Northwest     NW       Joel Craig       3.0 .98  3 4 Single Digit
western       WE       Sharon Kelly     5.3 .97  5 23
southwest     SW       Chris Foster     2.7 .8   2 18
southern      SO       May Chin         5.1 .95  4 15
southeast     SE       Derek Johnson    5.0 .70  4 17
eastern       EA       Susan Beal       4.4 .8   5 20
northeast     NE       TJ Nichols       5.1 .94  3 13
north         NO       Val Shultz       4.5 .89  5 9 Single Digit
central       CT       Sheri Watson     5.7 .94  5 13
```


The first example shows tabs separating fields from each other to help identify the lines with single-digit numbers in the last field.

In the second example, all lines that end with a single digit in the last field are replaced with the single-digit number plus the string `Single Digit`.

Occasionally with a search and substitute, the old string will be part of the new replacement string, which you can accomplish by placing an `&` (ampersand) in the replacement string. The location of the `&` determines the location of the old string in the replacement string.

Reading from a File for New Text

- Instead of inserting a line of text once, you might want to repeat the procedure several times, either in the same file or across multiple files.
- The `r` (read) command specifies a file name, and the contents of the file are inserted into the output after the lines specified by the address.
- The address may be a line number or pattern combination.

Reading from a File for New Text

```
$ cat northmesg
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************

$ sed '/north/r northmesg' data.file
northwest   NW     Joel Craig     3.0 .98        3 4
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
western     WE     Sharon Kelly   5.3 .97        5 23
southwest   SW     Chris Foster   2.7 .8         2 18
southern    SO     May Chin       5.1 .95        4 15
southeast   SE     Derek Johnson  5.0 .70        4 17
eastern     EA     Susan Beal     4.4 .8         5 20
northeast   NE     TJ Nichols     5.1 .94        3 13
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
north       NO     Val Shultz     4.5 .89        5 9
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
central     CT     Sheri Watson   5.7 .94        5 13
```


Note: The space following the `r` (read) command is required. You must also immediately follow the file name with a closing quotation mark.

Deleting Text

- The `d` (delete) command is used to:
  - Delete lines containing the search expression
  - Delete lines in the address range
- When used with `!`, it means do not delete those lines.

Deleting Text

```
$ sed '4,8d' data.file
northwest   NW     Joel Craig     3.0 .98        3       4
western     WE     Sharon Kelly   5.3 .97        5       23
southwest   SW     Chris Foster   2.7 .8         2       18
central     CT     Sheri Watson   5.7 .94        5       13

$ sed '/west/d' data.file
southern    SO     May Chin       5.1 .95        4       15
southeast   SE     Derek Johnson  5.0 .70        4       17
eastern     EA     Susan Beal     4.4 .8         5       20
northeast   NE     TJ Nichols     5.1 .94        3       13
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       13
```


The first example deletes lines 4 to 8 from the output.

The second example deletes any line containing the pattern west.

Deleting Text

```
$ sed '/^west/d' data.file
northwest   NW     Joel Craig     3.0 .98        3       4
southwest   SW     Chris Foster   2.7 .8         2       18
southern    SO     May Chin       5.1 .95        4       15
southeast   SE     Derek Johnson  5.0 .70        4       17
eastern     EA     Susan Beal     4.4 .8         5       20
northeast   NE     TJ Nichols     5.1 .94        3       13
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       13

$ sed '/south/,/north/d' data.file
northwest   NW     Joel Craig     3.0 .98        3       4
western     WE     Sharon Kelly   5.3 .97        5       23
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       13
```

In the first example, any line beginning with the pattern `west` is deleted.

The second example deletes the range of lines beginning with the first line containing the pattern `south`, up through the next line of the file containing `north`.

Reading `sed` Commands from a File

- Multiple `sed` commands can be put in a file and executed by using the `-f` option.
- When placing commands in a file:
  - Do not use quotation marks around the action and address
  - Ensure that there is no trailing white space at the end of each line

```
$ cat script1.sed
1,4d
s/north/North/
s/^east/East/

$ sed -f script1.sed data.file
southeast   SE      Derek Johnson   5.0 .70         4       17
Eastern     EA      Susan Beal      4.4 .8          5       20
Northeast   NE      TJ Nichols      5.1 .94         3       13
North       NO      Val Shultz      4.5 .89         5       9
central     CT      Sheri Watson    5.7 .94         5       13
```


The commands in the first example script:

- Delete the first four lines
- Replace any instances of `north` with `North`
- Change any line that starts with `east` to `East`

These commands can then be applied to the `data.file` file by using the command in the second example script.

Writing Output Files

- The `w` (write) command writes the specified records to a named file.
- The `w` command is followed by the path name of the file.

Using `sed` to Write Output Files

```
$ cat script5.sed
/north/w northregions
s/9[0-9]/& Great job!/w topperformers
$ sed -n -f script5.sed data.file
$ more northregions topperformers
::::::::::::::
northregions
::::::::::::::
northwest      NW      Joel Craig     3.0 .98         3       4
northeast      NE      TJ Nichols     5.1 .94         3       13
north          NO      Val Shultz     4.5 .89         5       9
::::::::::::::
topperformers
::::::::::::::
northwest      NW      Joel Craig     3.0 .98 Great job! 3   4
western        WE      Sharon Kelly   5.3 .97 Great job! 5   23
southern       SO      May Chin       5.1 .95 Great job! 4   15
northeast      NE      TJ Nichols     5.1 .94 Great job! 3   13
central        CT      Sheri Watson   5.7 .94 Great job! 5   13
```

ORACLE

Different `sed` commands can write to different files, such as shown in the examples in the slide.

The first example script of `sed` commands writes to two different output files. The first file, `northregions`, contains all lines with the pattern `north`. The second file, `topperformers`, has all lines containing a `9` followed by another digit. This output contains the matched number, followed by the string `Great job!`.

Note: The space following the `w` (write) command is required. This should be immediately followed by the file name.

Quiz

Which element's location in the `s` command determines the location of the old string in the replacement string?

a. /
b. $
c. !
d. &

Answer: d

Summary

In this lesson, you should have learned how to:

- Describe the `sed` editor
- Perform noninteractive editing tasks by using the `sed` editor

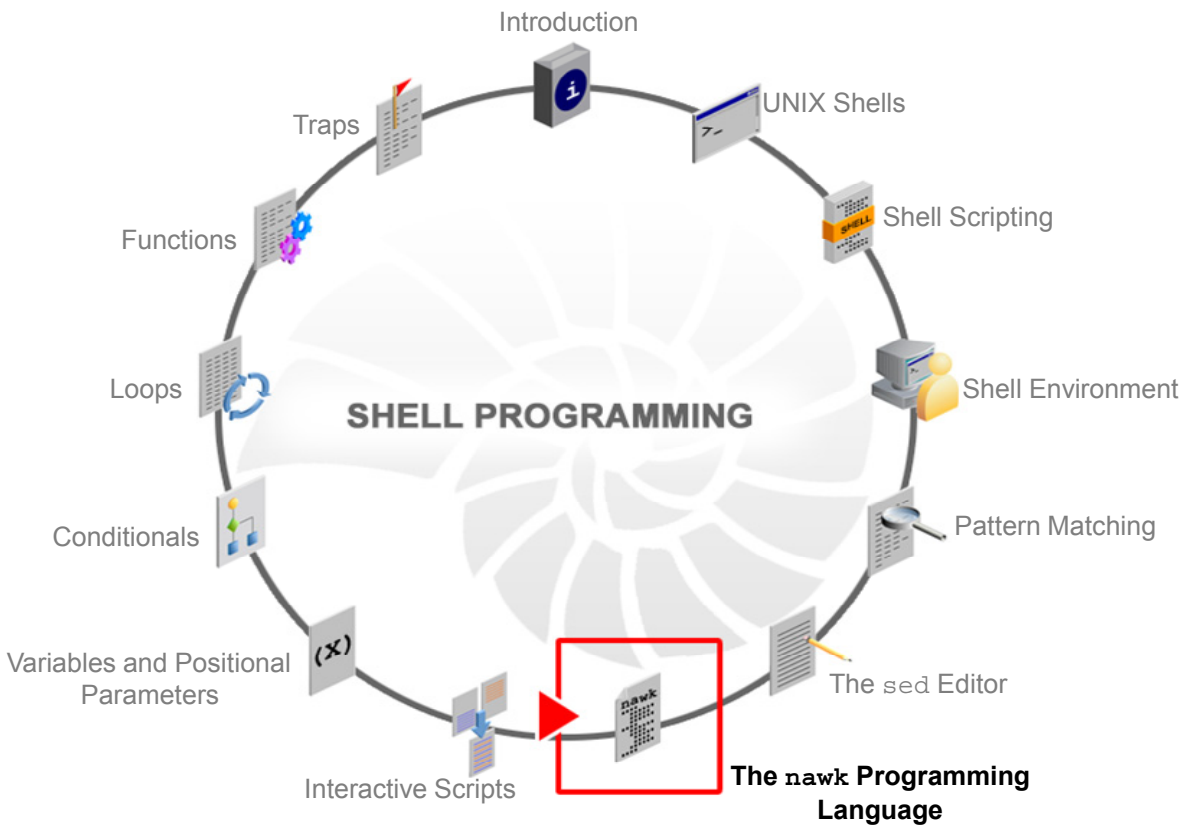
Practice 6 Overview: The `sed` Editor

This practice covers the following topics:

- Using the `sed` Editor
  - You delete, write, search, and substitute text patterns using regular expressions.
  - You print lines from the input file to the standard output or to a specified file.

7

The nawk Programming Language



Objectives

After completing this lesson, you should be able to:

- Describe the capabilities of the `nawk` programming language
- Display output by using the `print` statement
- Perform pattern matching by using regular expressions
- Use the `nawk` built-in and user-defined variables

Agenda

- Describing the capabilities of the `nawk` programming language
- Displaying output by using the `print` statement
- Performing pattern matching by using regular expressions
- Using the `nawk` built-in and user-defined variables

awk Programming Language

- awk is named for its authors Aho, Weinberger, and Kernighan of AT&T Bell Labs.
- The awk programming language is:
  - A record-oriented language
  - A text processing, data extraction, and reporting tool
  - A language executed by the awk interpreter
- The language has three variations:
  - awk is the original and oldest version.
  - nawk (New awk) is the improved version adapted by most UNIX vendors.
  - gawk is the free GNU version.

The awk programming language grew out of the recognition that many data processing problems are specialized applications of the concept of filtering, where the data is structured into records to which transformations are repetitively applied.

Note: Oracle Linux supports gawk, not nawk. This lesson discusses the nawk programming language. nawk is shipped as part of the Oracle Solaris Operating System (OS).

`nawk` Programming Language

The `nawk` programming language:

- Looks at data by records and fields
- Uses regular expressions
- Uses numeric and text variables and functions
- Uses command-line arguments

`nawk` Capabilities

Applications written in the `nawk` programming language provide the following capabilities:

- Filtering
- Numerical processing on rows and columns of data
- Text processing to perform repetitive editing tasks
- Report generation

Using the `nawk` programming language, you can develop programs within a script. Many programming concepts, such as conditionals, looping, variables, and functions, are included in the `nawk` programming language. This module focuses on the basic concepts of `nawk`.

Unlike `sed`, `nawk` looks at data by records and fields. By default, records are delimited by newline characters, and the fields within them are delimited by spaces or tabs, but these can be set to the delimiters that are built into your data, such as colons or commas.

`nawk` Command Format

- Commands have the form:

```
nawk 'statement' input.file
```

- Scripts are executed with:

```
nawk -f scriptfile input.file
```


In the `nawk 'statement' input.file` command format, the `statement` is enclosed in single quotation marks and might be in one of three forms:

- `pattern { ACTION }`

  The action is taken on those records that match the pattern.
- `pattern`

  All records that match the pattern are printed.
- `{ ACTION }`

  The action is taken on all records in the input file.

While executing `nawk` scripts by using the `nawk -f scriptfile input.file` format, you can combine repetitive `nawk` commands into `nawk` scripts that consist of one or more lines of the form:

- `pattern { ACTION }`

Where `pattern` is commonly an `RE` enclosed in slashes (`/RE/`) and `ACTION` is one or more statements of the `nawk` language.

Agenda

- Describing the capabilities of the `nawk` programming language
- Displaying output by using the `print` statement
- Performing pattern matching by using regular expressions
- Using the `nawk` built-in and user-defined variables

Printing Selected Fields

- The `print` statement outputs data from the file.
- Command conventions:
  - Enclose the command in single quotation marks.
  - Enclose the command in braces `{}`.
  - Specify individual fields with `$1`, `$2`, `$3`, and so on.
  - Specify individual records with `$0`.

When `nawk` reads a record, it divides the record into fields based on the `IFS` (input field separator) variable. This variable is predefined in `nawk` to be one or more spaces or tabs. The variables `$1`, `$2`, `$3` hold the values of the first, second, and third fields. The variable `$0` holds the value of the entire line.

Printing Selected Fields

```
$ cat data.file
northwest   NW      Joel Craig      3.0 .98         3       4
western     WE      Sharon Kelly    5.3 .97         5       23
southwest   SW      Chris Foster    2.7 .8          2       18
southern    SO      May Chin        5.1 .95         4       15
southeast   SE      Derek Johnson   5.0 .70         4       17
eastern     EA      Susan Beal      4.4 .8          5       20
northeast   NE      TJ Nichols      5.1 .94         3       13
north       NO      Val Shultz      4.5 .89         5       9
central     CT      Sheri Watson    5.7 .94         5       13
$ nawk '{ print $3, $4, $2 }' data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT
```


Given the `data.file` file, in the example in the slide, field 3 (first name), field 4 (last name), and field 2 (office) are printed.

Formatting with the `print` Statement

- Anything in double quotation marks is a string constant that can be used:
  - In `print` statements
  - As values to be assigned to variables among other things
- You can also use some special formatting characters:

Character	Octal Value	Meaning
\t	\011	Tab
\n	\012	Newline
	\007	Bell
	\042	“
	\044	$
	\045	%

ORACLE

By adding tabs or other text inside double quotation marks, you can format the output neatly. Some special formatting characters use letters, such as `\t` for tab. You can also specify an octal value, such as `\011` for tab. Some of the formats you can use are shown in the table in the slide.

Formatting with the `print` Statement

```
$ nawk '{ print $3, $4 "\t" $2 }' data.file
Joel Craig            NW
Sharon Kelly          WE
Chris Foster          SW
May Chin              SO
Derek Johnson         SE
Susan Beal            EA
TJ Nichols            NE
Val Shultz            NO
Sheri Watson          CT
```


The example in the slide adds a single tab character between field 4 and field 2. Notice that it was not necessary to use the comma before or after the tab (`\t`) in the previous `print` statement. The tab forces the output that follows to begin at the next tab position.

Note: If the fields in the `print` statement are separated by commas (`,`), then the fields are separated by a space when they are printed. The comma is not a required part of the syntax. In a `print` statement, the comma actually represents the value of the `nawk` variable output field separator (OFS). The default value of the OFS variable is a single space.

Agenda

- Describing the capabilities of the `nawk` programming language
- Displaying output by using the `print` statement
- Performing pattern matching by using regular expressions
- Using the `nawk` built-in and user-defined variables

Using Regular Expressions

Regular expression metacharacters can be used in the pattern.

```
$ nawk '/east/' data.file
southeast   SE      Derek Johnson   5.0 .70         4       17
eastern     EA      Susan Beal      4.4 .8          5       20
northeast   NE      TJ Nichols      5.1 .94         3       13

$ nawk '/east/ { print $1, $5, $4 }' data.file
southeast 5.0 Johnson
eastern 4.4 Beal
northeast 5.1 Nichols

$ nawk '/east/ { print $1, $5 "\t" $4 }' data.file
southeast 5.0       Johnson
eastern 4.4         Beal
northeast 5.1       Nichols

$ nawk '/^east/' data.file
eastern     EA      Susan Beal      4.4 .8          5       20
```

ORACLE

In the first example, `nawk` searches for the pattern `east`, and prints all lines containing that pattern.

The second example prints only fields 1, 5, and 4 from lines containing the pattern `east`.

The third example prints fields 1 and 5, and then a tab before field 4, for all lines containing the pattern `east`.

The string can contain regular expression characters. In the last example, the pattern `east` must be at the beginning of the line.

Using Regular Expressions

```
$ nawk '/.9/' data.file
northwest   NW      Joel Craig      3.0 .98         3       4
western     WE      Sharon Kelly    5.3 .97         5       23
southern    SO      May Chin        5.1 .95         4       15
northeast   NE      TJ Nichols      5.1 .94         3       13
north       NO      Val Shultz      4.5 .89         5       9
central     CT      Sheri Watson    5.7 .94         5       13

$ nawk '/\.9/' data.file
northwest   NW      Joel Craig      3.0 .98         3       4
western     WE      Sharon Kelly    5.3 .97         5       23
southern    SO      May Chin        5.1 .95         4       15
northeast   NE      TJ Nichols      5.1 .94         3       13
central     CT      Sheri Watson    5.7 .94         5       13
```


The `.` specifies any single character. In the first example, any character followed by a `9` would be a match.

Thus, the highlighted box points to a line of output you might not want if you are trying to identify lines that contain `.9`. To take away the special meaning of a regular expression character, precede it with a backslash (`\`) as in the second example in the slide.

The Special Patterns

There are two special patterns that are not used to match text in a file.

- `BEGIN`: An action to take before reading any lines
- `END`: An action to take after all lines are read and processed

There are two special patterns that are not used to match text in a file.

- The `BEGIN` pattern (all uppercase) indicates an action that occurs before any of the input lines are read. It is commonly used for printing a heading or title line for the report before any data is processed and to assign values to built-in variables.
- The `END` pattern (all uppercase) indicates an action that occurs after all input records have been read and fully processed. It is commonly used to print summary statements or numeric totals.

`BEGIN` and `END` statements can each occur multiple times in any `nawk` program and in any order. If there are multiple occurrences of either pattern, they are executed in the order in which they are found in the file.

The Special Patterns

```
$ nawk 'BEGIN { print "Eastern Regions\n" }; /east/ {
print $5, $4 }' data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichol

$ nawk 'BEGIN {
> print "Eastern Regions\n"}; /east/ {print $5, $4}'
data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
```


The first example has a `BEGIN` statement to add a header to the output.

Although you can use multiple lines for the `nawk` command, the beginning brace of the action for the `BEGIN` and `END` patterns must appear on the same line as the keyword `BEGIN` or `END`.

The second example in the slide shows the correct use. An example of incorrect use is:

```
$ nawk 'BEGIN
> { print "Eastern Regions\n" }; /east/ { print $5, $4 }'
data.file
nawk: syntax error at source line 2
context is
BEGIN >>>
<<<
nawk: bailing out at source line 2
```

The Special Patterns

```
$ nawk 'BEGIN { print "Eastern Regions\n"}; /east/ {print
$5, $4}
> END {print "Eastern Region Monthly Report"}' data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```


The END pattern allows the action to occur at the end of the input file as displayed in the first example.

Using `nawk` Scripts

- A `nawk` script is a collection of `nawk` statements (patterns and actions) stored in a text file.
- To instruct `nawk` to read the script file, use the command:

```
nawk -f script_file data_file
```


A `nawk` script reduces the chance for errors because the commands are stored in a file and are read from the file each time they are needed.

Using `nawk` Scripts

```
$ cat report.nawk
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}

$ nawk -f report.nawk data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```

Using `nawk` Scripts

```
$ cat report2.nawk
BEGIN {print "** Acme Enterprises **"}
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}

$ nawk -f report2.nawk data.file
** Acme Enterprises **
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```


Using a `nawk` script makes it easy to make changes or additions. In the example in the slide, a second `BEGIN` statement is added to print an overall heading for the report. Remember that `BEGIN` statements are executed in order.

Agenda

- Describing the capabilities of the `nawk` programming language
- Displaying output by using the `print` statement
- Performing pattern matching by using regular expressions
- Using the `nawk` built-in and user-defined variables

Built-in Variables

- As `nawk` processes an input file, it uses several variables.
- You can provide a value to some of these variables, whereas other variables are set by `nawk` and cannot be changed.
- A variable value can be a number, a string, or a set of values in an array.

Name	Default Value	Description
`FS`	Space or tab	The input field separator
`OFS`	Space	The output field separator
`NR`		The number of records from the beginning of the first input file

ORACLE

The table in the slide lists some of the built-in variables.

Input Field Separator (FS)

- The default input field separator (FS) is white space, which can be either a space or a tab.
- Frequently, other characters can separate the input, such as a colon or comma.
- You can set the input field separator variable with the -F option or set the value with an assignment.

```
nawk -F: 'statement' filename
nawk 'BEGIN { FS=":" } ; statement' filename
```


When using the -F option or the FS variable, you can specify more than one field separator either by placing the value in square brackets (creating a character class), or by separating the values with a | (OR statement) within double quotation marks.

```
nawk -F"[ :]" 'statement' filename
nawk -F" |:" 'statement' filename
nawk 'BEGIN { FS="[ :]" } next_statement' filename
nawk 'BEGIN { FS=" |:" } next_statement' filename
```

Input Field Separator (FS): Example

```
$ nawk 'BEGIN { FS=":" }; { print $1, $3 }' /etc/group
root 0
other 1
bin 2
sys 3
adm 4
uucp 5
mail 6
tty 7
lp 8
nuucp 9
staff 10
daemon 12
sysadmin 14
nobody 60001
noaccess 60002
nogroup 65534
```


This example sets the FS variable before processing the first record of the /etc/group file.

Input Field Separator (FS): Example

```
$ cat report3.nawk
BEGIN { FS=":" }
{ print $1, $3 }
$ nawk -f report3.nawk /etc/group
root 0
other 1
bin 2
sys 3
adm 4
uucp 5
mail 6
tty 7
lp 8
nuucp 9
staff 10
daemon 12
sysadmin 14
nobody 60001
noaccess 60002
nogroup 65534
```


This example shows how the previous command is set into a `nawk` script.

Output Field Separator (OFS)

- The default `OFS` is a space.
- In the `print` statement, a comma specifies using the `OFS`.
- If you omit the comma, the fields run together.
- You can also specify a field separator directly in the `print` statement, as in the following three lines:

```
$ nawk '{ print $3 $4 $2 }' data.file
$ nawk '{ print $3, $4, $2 }' data.file
$ nawk '{ print $3, $4 "\t" $2 }' data.file
```

ORACLE

Output Field Separator (OFS): Example

```
$ nawk 'BEGIN { OFS="\t" } ; { print $3, $4, $2 }'
data.file
Joel        Craig          NW
Sharon      Kelly          WE
Chris       Foster         SW
May         Chin           SO
Derek       Johnson        SE
Susan       Beal           EA
TJ          Nichols        NE
Val         Shultz         NO
Sheri       Watson         CT
```


To set the output field separator, place the assignment within a BEGIN statement as in the example in the slide.

Number of Records (NR)

- The number of records (NR) variable counts the number of input lines read from the beginning of the first input file.
- The variable's value is updated each time another input line is read.

Number of Records (NR): Example

```
$ more report4.nawk
{ print $3, $4, $2 }
END { print "The number of employee records is " NR }

$ nawk -f report4.nawk data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT
The number of employee records is 9
```


In the example in the slide:

- Inside the `BEGIN` pattern the value of `NR` is zero
- Inside the `END` pattern the value of `NR` is the number of the last record processed

User-Defined Variables

- `nawk` allows you to create your own variables.
- Variable names should not conflict with the `nawk` variables or function names.
- The value assigned to a variable can be:
  - A string (string of characters)
  - A numeric
  - A literal value (”`hello`”)
  - The contents from a field from the input file

The first time `nawk` encounters a reference to a given variable, the variable is created and initialized to a null string. The null string evaluates to `0` if it is used in a numeric expression. For all subsequent references, the value of the variable is whatever value was assigned last.

User-Defined Variables: Example

```
$ cat numexample.nawk
{ counter = counter + 1 }
{ print $0 }
END { print "*** The number of records is " counter }

$ nawk -f numexample.nawk data.file
northwest   NW     Joel Craig     3.0 .98        3       4
western     WE     Sharon Kelly   5.3 .97        5       23
southwest   SW     Chris Foster   2.7 .8         2       18
southern    SO     May Chin       5.1 .95        4       15
southeast   SE     Derek Johnson  5.0 .70        4       17
eastern     EA     Susan Beal     4.4 .8         5       20
northeast   NE     TJ Nichols     5.1 .94        3       13
north       NO     Val Shultz     4.5 .89        5       9
central     CT     Sheri Watson   5.7 .94        5       13
*** The number of records is 9
```


You can assign a value directly, or use the current value of the variable and use a combination operation on the variable as in the example in the slide.

User-Defined Variables: Example

```
$ cat numexample2.nawk
{ total = total + $8 }
{ print "Field 8 = " $8 }
END { print "Total = " total }

$ nawk -f numexample2.nawk data.file
Field 8 = 4
Field 8 = 23
Field 8 = 18
Field 8 = 15
Field 8 = 17
Field 8 = 20
Field 8 = 13
Field 8 = 9
Field 8 = 13
Total = 132
```


Each field of a record has a variable assigned to it. The first field is `$1`, the second field `$2`, and so forth. The value of these variables change each time `nawk` reads a new record. To sum the values of a field, create a new variable (for example, name the variable `total`). Add the value of the field to be summed to the variable each time a record is read.

The example in the slide prints the value in field 8 for each record, and adds that value to the `total` variable. At the end of file, it prints the accumulated total of all values found in field 8.

User-Defined Variables: Example

```
$ cat numexample3.nawk
{ total = total + $8 }
{ print $0 }
END { print "The total of field 8 is " total }

$ nawk -f numexample3.nawk data.file
northwest   NW      Joel Craig     3.0 .98         3       4
western     WE      Sharon Kelly   5.3 .97         5       23
southwest   SW      Chris Foster   2.7 .8          2       18
southern    SO      May Chin       5.1 .95         4       15
southeast   SE      Derek Johnson  5.0 .70         4       17
eastern     EA      Susan Beal     4.4 .8          5       20
northeast   NE      TJ Nichols     5.1 .94         3       13
north       NO      Val Shultz     4.5 .89         5       9
central     CT      Sheri Watson   5.7 .94         5       13
The total of field 8 is 132
```


A total can be written at the end of the file as displayed in this example.

Additional Examples

```
$ nawk '/N[EOW] { print NR, $0 }' data.file
1 northwest        NW      Joel Craig     3.0 .98 3       4
7 northeast        NE      TJ Nichols     5.1 .94 3       13
8 north            NO      Val Shultz     4.5 .89 5       9

$ nawk 'BEGIN { count = 0 }
> /N/ { print NR, $0; count = count + 1 }
> END { print "count of North regions is", count }'
data.file
1 northwest        NW      Joel Craig     3.0 .98 3       4
7 northeast        NE      TJ Nichols     5.1 .94 3       13
8 north            NO      Val Shultz     4.5 .89 5       9
count of North regions is 3
```


The first example in the slide prints the record number followed by the entire record of any record containing an `NE`, `NO`, or `NW`.

The second example prints the record number followed by the entire record if it contains an `N`. In addition, at the end of the file, it prints the number of records that matched the specified pattern.

Additional Examples

```
$ nawk '{ print "Record:", NR, $NF }' data.file
Record: 1 4
Record: 2 23
Record: 3 18
Record: 4 15
Record: 5 17
Record: 6 20
Record: 7 13
Record: 8 9
Record: 9 13
```


This example uses `$NF` rather than `$8` to obtain the value for the last field in each record. This works because, in `nawk`, the `NF` variable returns the number of fields in a record. `nawk` variables are not preceded by a `$` and, therefore, the `$` is used to convert the result from `8` to `$8`. `NF` equals `8` and `$NF` equals `$`; therefore, `$NF` returns the value in field `8`.

The `nawk` variable `NF` is set for each record to the number of fields in the record. Therefore, if your data file is not guaranteed to have the same number of fields per record, this variable always reports that unknown count.

Additional Examples

```
$ nawk '{ print "Record:", NR, "has", NF, "fields." }' \
raggeddata.file
Record: 1 has 8 fields.
Record: 2 has 6 fields.
Record: 3 has 8 fields.
Record: 4 has 7 fields.
Record: 5 has 5 fields.
Record: 6 has 6 fields.
Record: 7 has 7 fields.
Record: 8 has 5 fields.
Record: 9 has 6 fields.
```


Given the `raggeddata.file` file, the example in the slide prints the record number followed by the number of fields in the record, for each record in the input file.

```
$ cat raggeddata.file
northwest NW Joel Craig 3.0 .98 3 4
WE Sharon Kelly 5.3 .97 23
southwest SW Chris Foster 2.7 .8 2 18
southern SO May Chin 5.1 .95 15
southeast SE Derek 5.0 17
eastern Susan Beal 4.4 .8 20
NE TJ Nichols 5.1 .94 3 13
Val Shultz 4.5 5 9
central CT Sheri Watson .94 5
```

Additional Examples

```
$ nawk '{ print "Field 1 has", length($1), "letters." }'
raggeddata.file
Field 1 has 9 letters.
Field 1 has 2 letters.
Field 1 has 9 letters.
Field 1 has 8 letters.
Field 1 has 9 letters.
Field 1 has 7 letters.
Field 1 has 2 letters.
Field 1 has 3 letters.
Field 1 has 7 letters.
```


The `length()` function returns the number of characters in a variable value. The following example prints the string `Field 1` has, followed by the number of characters in `$1`, followed by the string `letters`.

Writing Output to Files

In any statement that generates an output, it is possible to have that output redirected to a file name instead, as follows:

- Use the redirection symbol, >, to send data to a file.

```
$ nawk '{ print $2, $1 > "textfile" }' data.file
$ cat textfile
NW northwest
WE western
SW southwest
SO southern
SE southeast
EA eastern
NE northeast
NO north
CT central
```

- Use two redirection symbols, >>, to append to a file.

In the example in the slide, the second and first fields are written to the `textfile` file.

Note: Enclose the file name in quotation marks if it is a string of characters. If the file name is stored in a variable or field, use the field number (preceded by the $ character) or the variable name.

The `printf()` Statement

- The `printf()` statement allows you to print a character string.
- This string can contain place holders that represent other values.
- These place holders can represent integers, floating points, characters, and character strings.
- Syntax:

```
printf ("string_of_characters" [ , data_values ] )
```

Note: For every place holder in the character string of `printf()`, there must be a value following the character string. These values must be separated by a comma.

ORACLE

The syntax of `printf()` is:

```
printf ("string_of_characters" [ , data_values ] )
```

Where *data_values* are used to fill in placeholders in the "*string_of_characters*" argument. Some of the placeholders are:

- `%d` Integer value
- `%c` Character value
- `%s` String value

Examples:

```
printf( "Hello World\n" )
printf( "The value is %d\n", num )
printf( "My name is %-10s %20s\n", $3, $4 )
```

Note: Notice the use of `\n` in the preceding examples. The `printf` statement does not generate a newline character by default.

The `printf()` Statement

```
$ nawk '{ printf "%10s %3d \n", $4, $7 }' data.file
Craig       3
Kelly       5
Foster      2
Chin        4
Johnson     4
Beal        5
Nichols     3
Shultz      5
Watson      5
```

Note: The `printf` statement does not use the `OFS` variable because of the `printf` statement's inherent ability to format the output line. So adjusting the fields to appear in column format is done by preceding the format specification with a digit, which refers to the number of spaces that the value should consume.

Formatting options are available for each of the placeholders:

- These options indicate a field width. The value is right-justified within the field.
  - `%numberd`
  - `%numbers`
  - `%numberc`
- These options indicate that the value is left-justified within the indicated field width.
  - `%-numberd`
  - `%-numbers`
  - `%-numberc`

The following is an example of formatting with the placeholders:

```
printf( "%d, %20s\n", 54, "John Smith" )
```

This example prints:

```
54, John Smith
```

Summary

In this lesson, you should have learned how to:

- Describe the capabilities of the `nawk` programming language
- Display output by using the `print` statement
- Perform pattern matching by using regular expressions
- Use the `nawk` built-in and user-defined variables

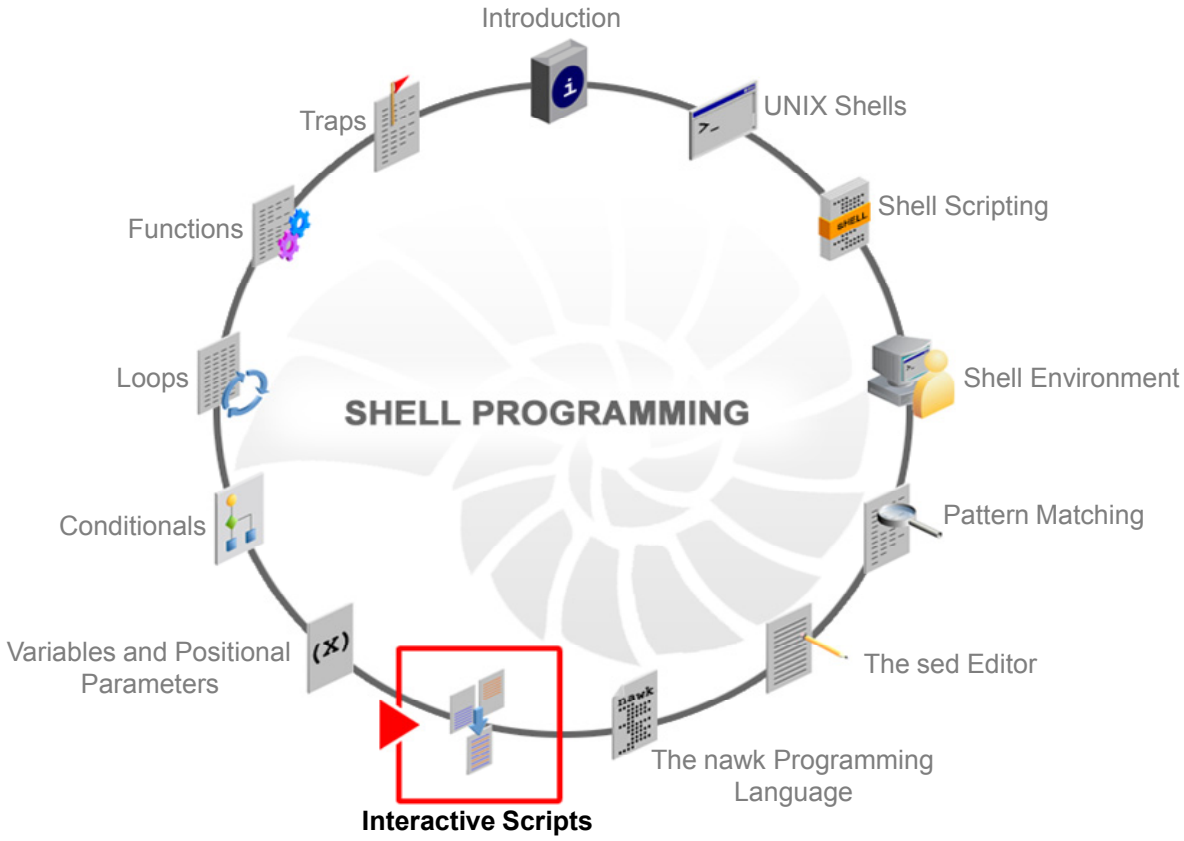
Practice 7 Overview: The `nawk` Programming Language

This practice covers the following topics:

- Using `nawk` and Regular Expressions
- Using `nawk` to Create a Report

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

Advantages of no interaction:

- The script runs the same way every time.
- The script can run in the background.

Advantages of interactive scripts:

- The script is more flexible.
- The user can customize the script as it runs.
- The script can report its progress as it runs.

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:
```


In the `printf` syntax:

- `FORMAT` is a string describing the format to use to write the remaining operands.
- `ARGUMENTS` are the strings to be written to standard output, under the control of format.

For more details about `printf`, refer to the `man` page.

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 examples in the slide display the use of various options of the `printf` statement and special characters on the command line.

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!
```


The `echo` statement can output one or more strings, whereas the `printf` statement can only output one string and always returns `1`.

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.

ORACLE

A *token* is any group of consecutive characters that do not contain white spaces. Thus, the first token begins with the first non-white-space character and ends when a white-space character is encountered. The second token begins with the next non-white-space character and ends with the first white-space character thereafter, and so on. White space can be included if quotation marks are used around the token.

Note: By default, the value of IFS is a white space, a tab, and a newline; however, you can change this to include other characters, such as commas, colons, or other field separators.

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'
```


In the first example in the slide, the number of variables supplied to `read` matches the number of tokens typed on the command line.

In the second example, the number of variables supplied to `read` is less than the number of tokens typed on the command line.

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
```


In the first example, the number of variables supplied to `read` is more than the number of tokens typed on the command line.

If variable names are not supplied to `read` when using the bash shell, it populates the `REPLY` variable with the user input, as in the second example.

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
```


Alternatively, you can pipe the output of a command to the `read` statement:

```
# ls -l /etc/passwd | read x y z p q r
# echo $x
-rw-r--r-
# echo $y
1
# echo $z
root
```

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.
```

ORACLE

The example in the slide shows how to prompt a user for some input by using `\c` in the output string. A `read` statement obtains the input. The input is read and saved in variables by using the `read` statement. An extra variable, `junk`, is supplied with both `read` statements to pick up any extra input by the user. The input is used in a message that prints to the screen with the `echo` statement.

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.
```

ORACLE

Similarly, you could prompt for input by using the `print` statement with the `-n` option. The `-n` option leaves the cursor at the end of the text line.

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.

A script can receive input from a file and send output to a file so that the script can run without user interaction. Later, a user can review the output file, or another script can use the output file as its input. File input and output are accomplished in the shell by integer handles that the kernel uses to track all open files in a process.

The numbers 3 through 9 are for programmer-defined file descriptors. You can use these to associate numeric values to path names.

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*

In a program, if there is a need to read or write multiple times to the same file, a shorthand file descriptor value might reduce errors in referencing the file name. Also, a change to the file name must be done only once if the file is subsequently accessed through the file descriptor. The table in the slide shows the syntax for file redirection.

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
```

Note: The file descriptor assigned to a file is valid in the current shell only.

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
```

ORACLE

The following sequence of activities is captured in the example in the slide:

1. Copying the `/etc/hosts` file to the `/tmp/hosts2` file.
2. Using `grep` to read the `/tmp/hosts2` file, striping out the comment lines, and sending the output to the `/tmp/hosts3` file.
3. Assigning file descriptor `3` to the `/tmp/hosts3` file for input, and then each `read` statement issued to file descriptor `3` reads a record from the file. The statement used to associate file descriptor `3` to the input file named `/tmp/hosts3` is `exec 3< /tmp/hosts3`.
4. Assigning file descriptor `4` to the `/tmp/hostsfinal` file for output. If the output file does not exist, it is created. If it exists, its size is truncated to `0` bytes. The statement that associates file descriptor `4` to the output file is: `exec 4> /tmp/hostsfinal`.

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
```


5. Reading the `/tmp/hosts3` file and outputting two fields to the `/tmp/hostsfinal` file. Reading from the input file is accomplished by specifying the file descriptor number to the `<&` argument of the `read` statement. This causes a line of the input file to be read for each `read` operation beginning with the first line, and continuing on sequentially.

   Writing text to the output file by specifying the file descriptor number to the `>&` argument of the `print` statement. This causes the output to be written as a line in the output file. The output file is written sequentially.
6. Closing the input file when it is no longer needed. This is good practice. The OS closes the file automatically when the process terminates if the program fails to close it.
7. When all writes to the output file are complete, close the file.

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
```

ORACLE

Note: All lines of the “here” document must be left-justified. Do not use leading spaces. The ending keyword must be on a line by itself.

Quiz

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

a. True
b. False

Answer: b

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

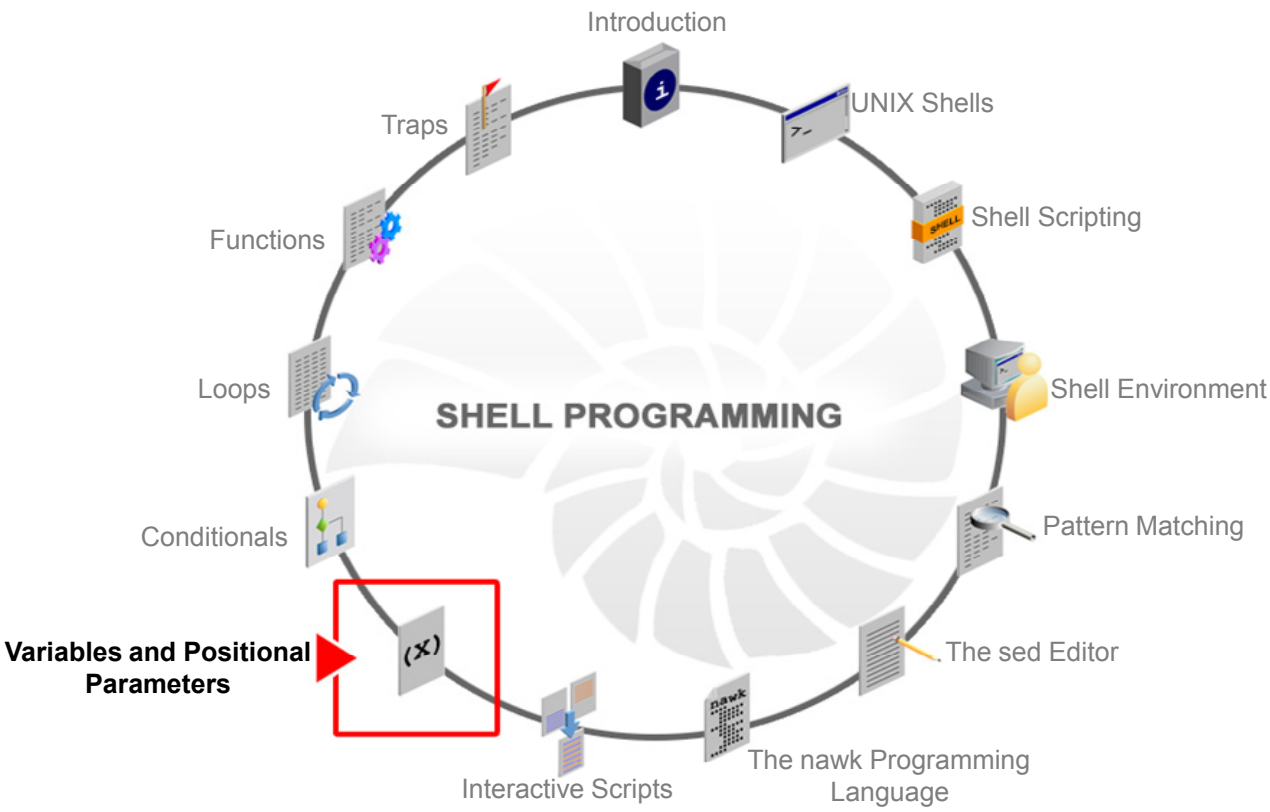
Practice 8 Overview: Interactive Scripts

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

9

Variables and Positional Parameters



Objectives

After completing this lesson, you should be able to:

- Describe the various types of scripting variables
- Define positional parameters for accepting user input

Agenda

- Describing the various types of scripting variables
- Defining positional parameters for accepting user input

Scripting Variables

- In the bash shell, there are four variable types:
  - String
  - Integer
  - Constant
  - Array
- All variables are of type string unless declared otherwise.
- A variable data type determines the values that can be assigned to a variable.

In the bash shell, there are four variable types:

- **String:** You can assign any value to a string variable.
- **Integer:** Integer variables can hold only number values.
- **Constant:** A constant is a read-only variable that is assigned a value when it is declared and the value cannot be changed.
- **Array:** Arrays might be restrictive, depending on whether an array is an array of integers or strings.

Accessing Variable Values

- When you access the value of a variable by preceding its name with a `$`, you might need to isolate the variable name from the characters that immediately follow it.
- For example:

```
$ flower=rose
$ printf "$flower $flowers $flowerbush"
rose
$ printf "$flower ${flower}s ${flower}bush"
rose roses rosebush
```

The `typeset` Statement

- The `typeset` statement sets variable attributes.
- In some cases, it changes a variable value, such as a right- or left-justification.
- Following are some of the `typeset` statement options:

Syntax	Description
`typeset -LZ var`	Strips leading zeros from the string `var`.
`typeset -Lnum var`	Left-justifies `var` within the field width specified by num.
`typeset -Rnum var`	Right-justifies `var` within the field width specified by num.
`typeset -i var`	It specifies that `var` can contain only integer values.
`typeset -r var`	It specifies that `var` is read-only; the value in `var` cannot be changed by subsequent assignment.

ORACLE

Note: The `typeset` command in the bash shell is supplied for compatibility with the Korn shell.

The `typeset` Statement: Example

```
$ cat strman1.sh
#!/bin/bash
# Script name: strman1.sh
typeset -r word="happy"
typeset -l word1="depressed"

print "123456789"
print "$word"
print
print "123456789"
print "$word1"

$ ./strman1.sh
123456789
    happy

123456789
depressed
```


To see a right-justified string with leading spaces, quote the variable in the output statement as in the example in the slide.

The `typeset` Statement: Example

```
$ cat strman2.sh
#!/bin/bash
# Script name: strman2.sh
typeset -r word="happy"
typeset -r word1="depressed"

print "123456789"
print $word
Print
print "123456789"
print $word1

$ ./strman2.sh
123456789
happy

123456789
depressed
```


Without quotation marks, leading spaces are not printed, as in the example in the slide.

The `declare` Statement

- The `declare` statement is an alternative to the `typeset` statement.
- Syntax:

```
declare [-afFrxi] [-p] [name[=value]]
```

- Following are some of the `declare` statement options:

Options	Description
`-a`	Treat each name as an array variable.
`-f`	Use function names only.
`-F`	Inhibit the display of function definitions.
`-i`	Treat the variable i as an integer.
`-r`	Make names read-only.

ORACLE

The `declare` statement in bash is available as an alternative to `typeset`. If no *name*s are given, then `declare` displays the values of variables instead.

Using `+` instead of `-` turns off the attribute instead.
When used in a function, `declare` makes each name local, as with the `local` command.

The `declare` Statement: Example

```
$ cat strman1.sh
!/bin/bash
# Script name: strman1.sh
declare -r word="happy"
declare -l word1="depressed"

echo "123456789"
echo "$word"
echo

echo "123456789"
echo "$word1"

$ ./strman1.sh
123456789
happy

123456789
depressed
```


Using the `echo` command to display variable value with the `declare` statement as in the example.

Removing Portions of a String

Syntax	Description
`${str_var%pattern}`	Removes the smallest right-most substring of string `str_var` that matches pattern
`${str_var%%pattern}`	Removes the largest right-most substring of string `str_var` that matches pattern
`${str_var#pattern}`	Removes the smallest left-most substring of string `str_var` that matches pattern
`${str_var##pattern}`	Removes the largest left-most substring of string `str_var` that matches pattern

The table in the slide summarizes the syntax used to remove portions of a string. You can use any of the shell metacharacters in a pattern following the `%`, `%%`, `#`, and `##` symbols.

Note: When the pattern does not contain any metacharacters, then `${str_var%pattern}` and `${str_var%%pattern}` have the same value, as do `${str_var#pattern}` and `${str_var##pattern}`.

Removing portions of a string is useful when working with path names. Sometimes only the name of a file is needed. You must strip everything in the path name except for the last component (the group of characters following the last `/` in the path name).

Sometimes the parent directory is required for a file for which you have the absolute path name. You must remove the last `/` and the characters that follow it.

The following page captures a few examples of removing portions of a string.

This example sets the value of the variable `stringx` to `/usr/bin/local/bin`.

```
$ stringx=/usr/bin/local/bin
```

The *smallest* right-most substring that matches `/bin` and `/bin*` in both these cases is `/bin`. Thus, the result of the second example is `/usr/bin/local`:

```
$ printf ${stringx%/bin}
/usr/bin/local
$ printf ${stringx%/bin*}
/usr/bin/local
```

The *largest* right-most substring that matches `/bin` is `/bin`. Thus, the result of the following example is the same as the preceding two cases.

```
$ printf ${stringx%%/bin}
/usr/bin/local
```

The *largest* right-most substring that matches `/bin*` is `/bin/local/bin`. Thus, the result of the following example is the string `/usr`:

```
$ printf ${stringx%%/bin*}
/usr
```

The *smallest* left-most substring that matches `/usr/bin` and `*/bin` is `/usr/bin` in both these cases. Thus, the result of the following example is `/local/bin`:

```
$ printf ${stringx#/usr/bin}
/local/bin
$ printf ${stringx#*/bin}
/local/bin
```

The *largest* left-most substring that matches `/usr/bin` is `/usr/bin`. Thus, the result of the following example is the same as the proceeding two cases:

```
$ printf ${stringx##/usr/bin}
/local/bin
```

The *largest* left-most substring that matches */`bin` is `/usr/bin/local/bin`. Thus, the result of the following example is the blank (null) string:

```
$ printf ${stringx##*/bin}
```

The *largest* left-most substring that matches `*/` is `/usr/bin/local/`. Thus, the result of the following example is the string `bin`:

```
$ printf ${stringx##*/}
bin
```

Declaring an Integer Variable

- You can declare an integer variable in two ways:
  - By using `typeset -i` or `declare -i`
  - By using integer in front of the variable name

```
typeset -i int_var1[=value] int_var2[=value] ...
int_varn[=value]
```

```
declare -i int_var1[=value] int_var2[=value] ...
int_varn[=value]
```

- An integer variable can have only integer numbers assigned to it.
- An assignment of a number with a decimal part results in the decimal being truncated.

Using Integer Variables: Examples

```
$ typeset -i num
$ num=5
$ print $num
5
$ typeset -i num
$ num=25.34
$ print $num
25
$ typeset -i num # base 10 integer
$ num=27
$ print $num
27
$ typeset -i8 num # change to base 8
$ print $num
8#33
$ num=two
/usr/bin/ksh: two: bad number
$ print $num
8#33
```

Using Integer Variables: Examples

```
$ typeset -i num
$ num=5
$ print $num
5

$ declare -i num
$ num=5
$ echo $num
25

$ typeset -i num # base 10 integer
$ num=27
$ print $num
27

$ declare –i number
$ number=3
$ echo "Number = $number”
number=3
```


In the examples, observe the declaration of integer variables by using `typeset` and `declare` statements.

Arithmetic Operations on Integer Variables

Operator	Operations	Example	Result
+	Addition	((x = 24 + 25))	49
-	Subtraction	((x = 100 - 25))	75
*	Multiplication	((x = 4 * 5))	20
/	Division	((x = 10 / 3))	3
%	Modulo (remainder)	((x = 10 % 3))	1
#	Base	2#1101010 16#6A	10#106
<<	Shift bits left	((x = 2#11 << 3))	2#11000
>>	Shift bits right	((x = 2#1001 >> 2))	2#10
&	Bit-wise AND	((x = 2#101 & 2#110))	2#100
|	Bit-wise OR	((x = 2#101 | 2#110))	2#111
^	Bit-wise exclusive OR	(((x = 2#101 ^ 2#110))	2#11

Creating Constants

- The value of a constant (read-only) variable cannot be changed.
- A variable can be made read-only with the following syntax:

```
readonly var[=value]
```


The square brackets around `=value` mean that the assignment of a value is not always necessary. For instance, if the variable had previously been created and assigned a value, and you now want to make it read-only (and not change its current value), do not use `=value`.

If the variable did not previously exist, and you make it read-only, you may never assign a value to the variable.

```
$ sh
$ var=constant
$ readonly var
$ unset var
Sh; unset: warning: var: is read only

$ var=new_value
Sh: var: is read only
```

Declaring Arrays

- Arrays:
  - Contain multiple values
  - Are created when you use them
  - Are zero-based, the first element is indexed with the number `0`
- You can create arrays of strings or integers.
- By default, an array contains strings.
- To create an array of integers:
  - Declare a variable as an integer
  - Assign integer values to the array elements

```
integer my_array
my_array[1]=5
my_array[12]=16
```

ORACLE

Arrays are variables that contain more than one value. Do not declare arrays explicitly or explicitly set an array size. An array is created when you first use it. It is treated as an array when you first assign a value to one element of the array variable. Each array can hold up to 1024 values.

You can create arrays of strings or integers. By default, an array contains strings. To create an array of integers, declare a variable as an integer and then use it as an array by assigning integer values to the elements. For example:

```
integer my_array
my_array[1]=5
my_array[12]=16
```

The first element of an array is indexed by `0`, the second element is indexed by `1`, and so on. Therefore, the largest index value that is valid is 4095.

All arrays in the bash shell are one-dimensional. Enclose an array reference in braces for the shell to recognize it as an array. For example, use `${arr[1]}` instead of `$arr[1]`.

The syntax for accessing the value of the `i`th array element is the following, where `i` is an integer:

```
{array_name[i]}
```

To print the values of all array elements, use the following syntax:

```
{array_name[*]}
```

To print the number of elements (assigned values) in the array, use the following syntax:

```
{#array_name[*]}
```

You do not have to assign array elements in order or assign values to any elements. Skipped array elements are not assigned a value and are treated as though they do not exist.

```
arr[2]=two
arr[4]=4
arr[8]=eight
arr[16]=16
```

Note: You cannot use the `export` statement with arrays in the bash shell.

Using Arrays: Examples

- Creating an array of three strings:

```
arr[0]=big
arr[1]=small
arr[2]="medium sized"
```

- Creating an array of three strings with the `set` command:

```
set arr big small "medium sized"
```

- Creating an array of five integers:

```
declare -a integer num
num[0]=0
num[1]=100
num[2]=200
num[3]=300
num[4]=400
```


The first two examples in the slide illustrate how to create arrays of strings. You do not need to enclose the values assigned to the various array elements within double quotation marks unless the string value contains special characters, such as spaces or tabs.

In the third example, note the creation of the array of integers. The variable `num` is first declared as an integer variable and then it is used as the name of an array to assign integers to the first five elements of the array. If you try to assign a value other than an integer to any of the elements of the array, you get a `bad number` error message.

Using Arrays: Examples

- Printing the number of array elements in array `num`:

```
$ printf ${#num[*]}
5
```

- Printing the values of all array elements in array `arr`:

```
$ printf ${arr[*]}
big small medium sized
```

- Destroying an array `arr`:

```
$ unset arr
```

Quiz

All variables, whether strings, integers, constants, or arrays are of type string unless declared otherwise.

a. True
b. False

Answer: a

Agenda

- Describing the various types of scripting variables
- Defining positional parameters for accepting user input

Positional Parameters

- The operation of a script varies depending on what arguments are given to the script on the command line.
- The shell automatically assigns special variable names, called *positional parameters*, to each such argument.

Parameter Name	Description
`$0`	The name of the script
`$1`	The first argument to the script
`$2`	The second argument to the script
`$9`	The ninth argument to the script
`$#`	The number of arguments to the script
`$@`	A list of all arguments to the script where each parameter is seen as a word
`$*`	A list of all arguments to the script where each parameter is quoted as a string

ORACLE

The execution of scripts is made versatile by the ability to run a script based on arguments supplied on the command line. The positional parameter names and meanings are displayed in the table in the slide.

The `shift` Statement

- By default, the `shift` statement shifts the values held in the positional parameter.
- The `shift` statement drops the first value (`$1`) and shifts all other values to the left.

The `shift` Statement: Example

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

echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 $10 are: ' $1 $2 $9 $10
echo

shift
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

shift 2
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9
echo '${10}: ' ${10}
```


In the example in the slide, the value in `$2` is shifted or copied into `$1`, and the value in `$3` is shifted or copied into `$2`, and so on.

The `shift` Statement: Example

```
$ ./argtest.sh a b c d e f g h i j k l m n
'14: ' 14
'a b c d e f g h i j k l m n: ' a b c d e f g h i j k l m n
'a b c d e f g h i j k l m n: ' a b c d e f g h i j k l m n

'a b i a0 are: ' a b i a0

'13: ' 13
'b c d e f g h i j k l m n: ' b c d e f g h i j k l m n
'b c d e f g h i j k l m n: ' b c d e f g h i j k l m n

'b c j are: ' b c j
'11: ' 11
'd e f g h i j k l m n: ' d e f g h i j k l m n
'd e f g h i j k l m n: ' d e f g h i j k l m n

'd e l are: ' d e l
'm: ' m
./argtest.sh: bad substitution
```


The `echo` of `${10}` causes the error message at the end of the output because the bash shell cannot access command-line parameters beyond `$9`.

The `set` Statement

- The `set` statement allows you to assign values to positional parameters.
- Syntax:

```
set value1 value2 ... valueN
```

Where:

- `$1` has value `value1`
- `$2` has value `value2`, and so on
- `$0` is the name of the script

- Use the `set` statement to create a parameter list by using the statement or variable substitution.

```
set $(cal)
set $var1
```

ORACLE

To sort the positional parameters lexicographically, use the `set -s` statement. If you sort the list of values in lexical order, the first value on the list is assigned to `$1`, the second to `$2`, and so on.

The statement `set --` unsets all positional parameters. Thus, `$1`, `$2`, and so on, have no values. The value of `$0` is still the script name.

The `set -s` and `set --` statements work on positional parameters regardless of how the positional parameter names were assigned their values.

These statements work whether values were assigned from command-line arguments or by using the `set` statement. Statement or variable substitutions, combined with the `set` statement, are useful.

For example, to find out how many days are in the current month, use the `cal` statement. The `cal` statement outputs a value for each day in the month plus an additional nine values: the month, the year, and the day-of-week values.

The `set` Statement: Example

```
$ cat pospara2.sh
#!/bin/bash
# Script name: pospara2.sh

print "Executing script $0 \n"
print "$1 $2 $3"

set uno duo tres
print "One two three in Latin is:"
print "$1"
print "$2"
print "$3 \n"

textline="name phone address birthdate salary"
set $textline
print "$*"
print 'At this time $1 =' $1 'and $4 =' $4 "\n"

set -s
print "$* \n"

set --
print "$0 $*"
```


The `pospara2.sh` example script in the slide sets the values of positional parameters by using the `set` statement.

The values `a`, `b`, and `c` are passed into the script and assigned to positional parameters `$1`, `$2`, and `$3` respectively. The values of positional parameters are displayed by using the `print` statement and then given new values with the `set` statement.

The `textline` variable is assigned a value, which is a string consisting of several words. The statement `set $textline` assigns new values to positional parameters. Because `$textline` is replaced by its value (a string of several words), the words become the positional parameter values.

The script prints the positional parameter values by using the `print $*` statement. The script then prints the values of the `$1` and `$4` positional parameters.

Next, the positional parameters are sorted by the `set` statement (no options or arguments) and then `print $*` is used to print the sorted list. Lastly, the `set --` statement unsets positional parameters so that when the `print $0 $*` statement is executed, only the name of the script (which is held in `$0`) is printed.

The `set` Statement: Example

```
$ ./pospara2.sh a b c
Executing script ./pospara2.sh

a b c
One two three in Latin is:
uno
duo
tres

name phone address birthdate salary
At this time $1 = name and $4 = birthdate

address birthdate name phone salary
./pospara2.sh
…
…
…
<output omitted>
```

The Values of the"$@" and "$*" Positional Parameters

The values of $@ and $* are identical, but the values of "$@“ and "$*" are different.

- "$@" expands to "$1" "$2" "$3" ... "$n“
- "$*" expands to "$1x$2x$3x...$n“

The values of $@ and $* are identical, but the values of "$@" and "$*" are different.

The expansion of "$@" is a list of strings consisting of the values of positional parameters. The expansion of "$*" is one long string containing the positional parameter values separated by the first character in the set of delimiters of the IFS variable. For example:

- "$@" expands to "$1" "$2" "$3" ... "$n"; that is, n separate strings.
- "$*" expands to "$1x$2x$3x...$n", where x is the first character in the set of delimiters for the IFS variable. This means that "$*" is one long string.

Using the Values of "$@" and "$*": Example

```
$ cat arginfo.sh

#!/bin/bash
# Purpose: Print out information about the positional parameters.
#
# Name: arginfo.sh
echo "The name of the script is: $0"

echo "Positional Parameter      Value"
echo "     \$1                    $1"
echo "     \$2                    $2"
echo "     \${10}                 ${10}"

echo "The number of positional parameters is: $#"
echo
if (( $# >= 5 )); then
     echo "Now shift all values over by 5 places."
     shift 5

     echo "Positional parameters remaining:"
     echo $*
     echo

     echo "The number of positional parameters is: $#"
fi
```

Using the Values of "$@" and "$*": Example

```
$ ./arginfo.sh
The name of the script is: ./arginfo.sh
Positional Parameter      Value
     $1
     $2
     ${10}
The number of positional parameters is: 0
```


When you use quotation marks to group tokens into a string, the string sets the value of a single positional parameter as in the example in the slide.

Quiz

The `$0` positional parameter refers to which of the following?

a. The list of all arguments to the script
b. The first argument to the script
c. The name of the script
d. The number of arguments to the script

Answer: c

Summary

In this lesson, you should have learned how to:

- Describe the various types of scripting variables
- Define positional parameters for accepting user input

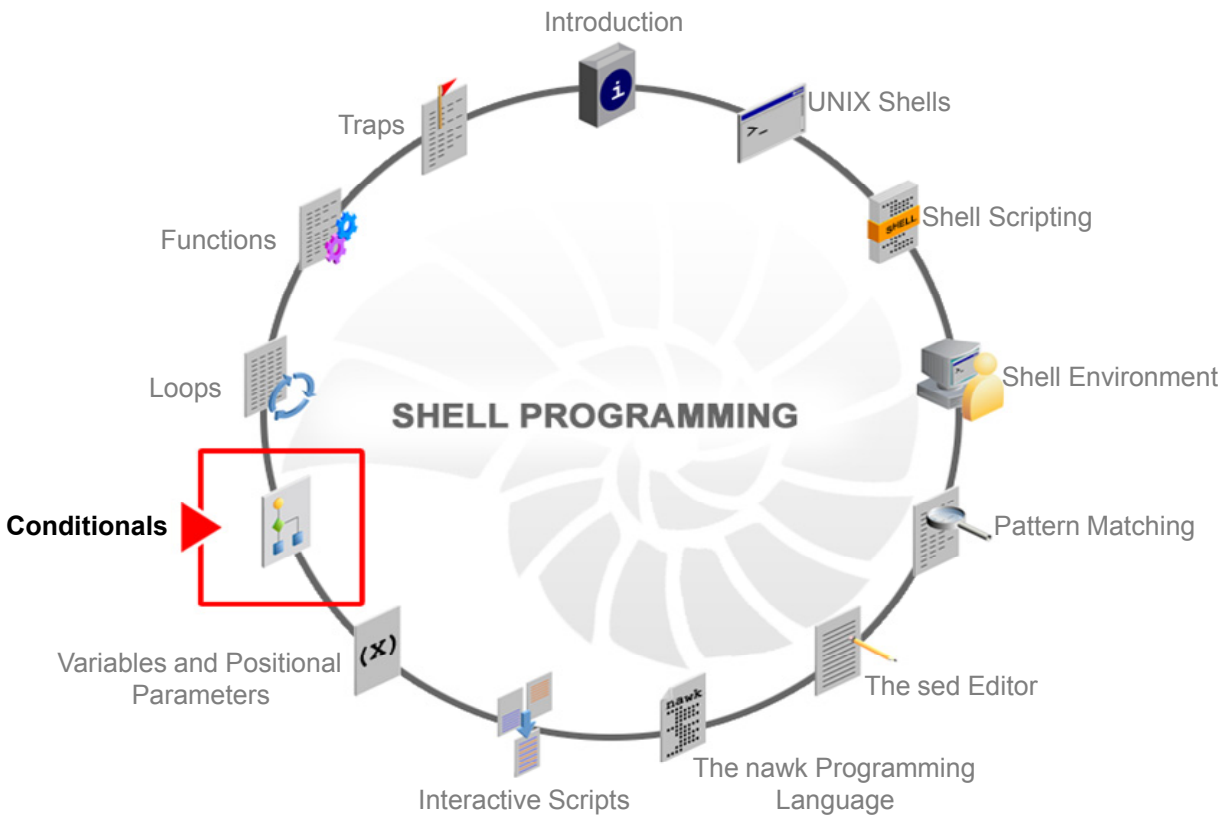
Practice 9 Overview: Variables and Positional Parameters

This practice covers the following topic:

- Using Advanced Variables, Parameters, and Argument Lists

10

Conditionals



Objectives

After completing this lesson, you should be able to:

- Describe the various forms of `if` statements and their role in testing conditions
- Explain the use of `if` statements through examples
- Describe the role of the `case` statement in choosing from alternatives

Agenda

- Describing the various forms of `if` statements and their role in testing conditions
- Explaining the use of `if` statements through examples
- Describing the role of the `case` statement in choosing from alternatives

The `if` Statement

- Allows you to specify courses of action in a shell script, depending on the success or failure of some command
- Is a conditional statement that allows a test before performing another statement
- Has several forms.
  - The simple `if` statement
  - The `if/then/else` statement
  - The `if/then/else/elif` statement
  - The nested `if` statement
- In its simplest form, an `if` statement has the following syntax:

```
if command
then
            block_of_statements
fi
```

Parts of the `if` Statement

The `if` statement contains three parts:

- Command
- Block of statements (if-block)
- End of the `if` statement

The `if` statement contains three parts:

- **Command:** The command in the `if` statement often involves a numerical or string test comparison, but it can also be any command that returns a status of `0` when it succeeds and some nonzero status when it fails. Some of the conditions frequently used within an `if` are:
  - The `[ . . . ]` command performs arithmetic or string comparisons
- **Block of statements:** The statements that follow the `then` statement can be any valid UNIX command, executable user program, executable shell script, or shell statement with the exception of `fi`.
- **End of the `if` statement:** End every `if` statement with the `fi` statement.

The `if` Statement: Example

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

# Script name: snoopy.sh

name=snoopy

if [ "$name" = "snoopy" ]
then
        echo "It was a dark and stormy night."
Fi

$ ./snoopy.sh
It was a dark and stormy night.
```


Note: When you compare a variable in a condition test, always use double quotation marks around the variable to ensure that it is evaluated properly. Without the double quotation marks, problems arise if a string value has a space or if the value of the variable is null.

When there are double quotation marks around a variable in the `if` statement, and the variable is null, and the quotation marks are not present, the error message is `test: argument expected`.

The `if/then/else` Statement

- If the command in the `if` statement succeeds, the block of statements after the `then` statement are executed.
- If the command in the `if` statement fails, then the block of statements after the `then` statement are skipped, and statements following the `else` are executed.
- In both the above cases, execution continues with the statement following the `fi` statement.
- Syntax:

```
if command
then
block_of_statements
else
block_of_statements
fi
```


In a conditional, you frequently have tasks to perform when the tested command succeeds or fails. The shell can accommodate this with the `if/then/else` syntax.

The `if/then/else` Statement: Example

```
$ cat snoopynap.sh
#!/bin/bash
# Script name: snoopynap.sh
name=snoopy
if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
else
        echo "Snoopy is napping."
fi
$ cat findroot.sh
#!/bin/bash
# Script name: findroot.sh
if grep root /etc/passwd > /dev/null
then
        echo "Found root!"
else
        echo "root not in the passwd!"
        echo "Do not logout until the passwd file is repaired!"
fi
```


The examples in the slide are samples of the `if/then/else` statement implementation.

The `if/then/else/elif` Statement

- The shell first evaluates the commands in sequence.
- The statements associated with the first successful command are executed, followed by statements after `fi`.
- If none of the commands succeeds, then the statements following the `else` statement are executed.
- Execution continues with any statements following `fi`.
- Syntax:

```
if command1
then
    block_of_statements
elif command2
then
    block_of_statements
else
    block_of_statements
fi
```

ORACLE

In the `if/then/elif/else` form of the `if` statement, the first `else` becomes another `if` statement or “else if” instead of a simple `else`. The shell first evaluates command1, then command2, and so on, stopping with the first command that succeeds. The statements associated with that successful command are then executed, followed by any statements after the `fi` statement.

If none of the commands succeed, then the statements following the `else` statement are executed. Execution then continues with any statements following the `fi` statement.

The `if/then/else/elif` Statement: Example

```
$ cat snoopy2.sh
#!/bin/bash
# Script name: snoopy2.sh
name=snoopy
if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
        echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
        echo "The doctor is in."
elif [[ "$name" == "schroeder" ]]
then
        echo "In concert."
else
        echo "Not a Snoopy character."
fi
```


The `if/then/elif/else` syntax can have as many `elif` command statements as needed before the final `else` statement. Follow each `elif` by a `then` statement.

The `if/then/else/elif` Statement: Example

```
$ cat termcheck.sh
#!/bin/bash
# Script name: termcheck.sh
if [[ "$TERM" == "sun" ]]
then
        echo "You are using the sun console device."
elif [[ "$TERM" == "vt100" ]]
then
        echo "You are using a vt100 emulator."
elif [[ "$TERM" == "dtterm" ]]
then
        echo "You are using a dtterm emulator."
else
        echo "I am not sure what emulator you are using."
fi
```


In the example in the slide, the value of the `TERM` variable is tested by comparing it against different known values. As soon as one of the tests succeeds, the statements that follow the `then` statement are executed. If all comparisons of `TERM` fail, the statements following the `else` statement are executed.

Nested `if` Statements

- You can use an `if` statement inside another `if` statement.
- You can have as many levels of nested `if` statements as you can track.
- Each `if` statement requires its own `fi` statement.

Nested `if` Statements: Example

```
$ cat leap.sh
#!/bin/bash
# Script name: leap.sh
# Make sure the user enters the year on the
# command line
# when they execute the script.
if [ $# -ne 1 ]
then
echo "You need to enter the year."
exit 1
fi
year=$1

(Continued…)
```


In the example in the slide:

1. User input is required for the `year` as a command-line argument.
2. The script assigns the value of `$1` to the `year` variable.
3. The rest of the script determines whether the year entered is a leap year and prints an appropriate message if it is.

The second `if` statement contains an `if` statement as one of its statements, which is where the nesting occurs. For the script to print the year as a leap year:

- The condition that the year entered be evenly divisible by 4 must be true
- The condition that the year not be evenly divisible by 100 must also be true

The script is then run several times from the command line. The numbers input to the script are chosen to test that something gets printed each time. It is not necessary to always have an `else` with an `if`. However, you must include an `fi` statement for each `if` statement.

Nested `if` Statements: Example

```
if (( (year % 400) == 0 ))
then
        print "$year is a leap year!"
elif (( (year % 4) == 0 ))
then
        if (( (year % 100) != 0 ))
        then
                print "$year is a leap year!"
        else
                print "$year is not a leap year."
        fi
else
        print "$year is not a leap year."

$ ./leap.sh 2000
2000 is a leap year!
$ ./leap.sh 2014
2014 is not a leap year.
$ ./leap.sh 2050
2050 is not a leap year.
```

The `exit` Status

- When a statement executes, the statement returns a numeric value called an `exit` status.
- The `exit` status is an integer variable and is saved in the `?` shell reserved variable.
- A statement that executes successfully, returns an `exit` status of `0`, meaning zero errors.
- A statement that executes unsuccessfully returns an `exit` status of nonzero, meaning one or more errors occurred.

Therefore, when the statement following the `if` keyword executes successfully and returns a `0` exit status, the statements following the `then` statement are executed.

However, when the statement following the `if` keyword executes unsuccessfully and returns a nonzero exit status, the statements following the `fi` statement are executed.

The `then` is a separate statement in a shell. It must appear on a separate line unless it is preceded by a `;` (semicolon). This is similar to the way several OS commands are provided on one line as long as a `;` separates each command. For example, the statement:

```
if command; then
```

is treated as being the same as:

```
if command
then
```

Note: The `exit` statement, alternatively, terminates execution of the entire script. It is most often used if the input requested from the user is incorrect, a statement ran unsuccessfully, or some other error occurred.

The `exit` statement takes an optional argument that is an integer. This number is passed back to the parent shell, where it is stored in the shell reserved `$?` variable. A nonzero integer argument to the `exit` statement means that the script ran unsuccessfully. A zero argument means the script ran successfully.

The `exit` Status: Examples

- The following example uses `grep` to search the `/etc/passwd` file for the `root` string.

```
$ grep root /etc/passwd
root:x:0:0:Super-User:/root:/usr/bin/bash
$ echo $?
0
```

- You may redirect or pipe `stdout` as part of the output.

```
$ if grep root /etc/passwd > /dev/null
> then
>           echo "Found root!"
> fi
Found root!
```

Agenda

- Describing the various forms of `if` statements and their role in testing conditions
- Explaining the use of `if` statements through examples
- Describing the role of the `case` statement in choosing from alternatives

Using `if` Statements

The `if` statements can be used with various types of scripting parameters and operators for testing conditions such as:

- Numeric and string comparison operators
- Pattern-matching metacharacters
- Positional parameters
- Boolean `AND`, `OR`, and `NOT` operators

The following slides provide some examples of using the `if` statements with various types of scripting parameters and operators for testing conditions.

Numeric and String Comparison Operators

Shell scripting provides integer and string comparison capabilities in the form of comparison operators.

- For numeric comparison, use `[]` and spaces.
- For string comparison, use `[ … ]` and spaces.

When you compare numbers in the bash shell, use single parentheses `[ ]`. There is no requirement about the white space around any of the components. Use white space around variables and operators for improved readability.

When using square brackets `[ ]`, the shell requires a space after the opening `[` and before the closing `]`. For example, `[$a -eq $b]` is invalid. `[ $a -eq $b ]` is the correct way to use it.

Numeric Comparison Operators

Bash Shell	Returns true (0) if:
`[ $num1 -eq $num2 ]`	*num1* **equals** *num2*
`[ $num1 -ne $num2 ]`	*num1* **does not equal** *num2*
`[ $num1 -lt $num2 ]`	*num1* **is less than** *num2*
`[ $num1 -gt $num2 ]`	*num1* **is greater than** *num2*
`[ $num1 -le $num2 ]`	*num1* **is less than or equal to** *num2*
`[ $num1 -ge $num2 ]`	*num1* **is greater than or equal to** *num2*

ORACLE

Numeric Comparison Operators: Example

```
$ num=21

$ if [ num > 15 ]; then
>     echo "You are old enough to drive in most places."
> fi
You are old enough to drive in most places.
```


The example in the slide sets a variable and tests the value of the variable by using the double parentheses in an `if` statement. The `then` statement is placed on the same line with `if`.

The `[ ]` syntax is the alternative form to the `let` statement. The `[ ]` syntax is preferred because it is more readable. The same example can also be written as:

```
$ if let "num > 15"; then
>       echo "You are old enough to drive in most places."
> fi
You are old enough to drive in most places.
```

String Comparison Operators

Bash Shell	Returns `true` (`0`) if:
`[ str1 = str2 ]`	*str1* **equals** *str2*
`[ str1 != str2 ]`	*str1* **does not equal** *str2*
`[ str1 < str2 ]`	*str1* precedes *str2* **in lexical order**
`[ str1 > str2 ]`	*str1* follows *str2* in lexical order
`[ -z str1 ]`	`str1` has length zero (holds null value)
`[ -n str1 ]`	`str1` has nonzero length (contains one or more characters)

ORACLE

When comparing strings in the bash shell, use double square brackets `[[ ... ]]`. Here, the shell is particular about white space. You must use white space around every component within `[[ ... ]]`. This includes putting a white space after `[[` and before `]]`.

When comparing a string to a pattern, the string must appear on the left side of the equal sign and the pattern must appear on the right.

For string comparisons with either the `[ ]` or `[[ ]]` statement, use quotation marks around literal strings or variables. The quotation marks are necessary when the variable value or string value has embedded spaces.

Note: Lexical order means that lowercase letters have greater value than uppercase letters. As such, attempting to compare mixed-case strings might produce unexpected results.

String Comparison Operators: Example

```
$ name=fred

$ if [ "$name" == "fred" ]
> then
>    echo "fred is here"
> fi
fred is here
```


The example in the slide sets a variable and then tests the value of the variable by using double square brackets in an `if` statement.

Pattern Match Metacharacters

Metacharacter	Meaning
?	Matches any one single character
[ ]	Matches one character in the specified set
*	Matches zero or more occurrences of any characters
?(*pat1*|*pat2*|...|*patn*)	Matches zero or one of the specified patterns
@(*pat1*|*pat2*|...|*patn*)	Matches exactly one of the specified patterns
*(*pat1*|*pat2*|...|*patn*)	Matches zero, one, or more of the specified patterns
+(*pat1*|*pat2*|...|*patn*)	Matches one or more of the specified patterns
!(*pat1*|*pat2*|...|*patn*)	Matches any pattern except the specified patterns

ORACLE

When comparing a variable or string against a pattern, the pattern is not quoted. Using quotation marks hides the pattern-match capability of the pattern metacharacters. The table in the slide shows the metacharacters used for comparison.

It is important that the contents of the double-square-bracket conditional tests start and end with a space. Surround the conditional operators by at least one space. Also, as the square brackets act as a quoting mechanism, do not use quotation marks to surround the pattern text.

Pattern Match Metacharacters: Example

```
$ name=fred
$ if [ "$name" == f* ]; then
>    echo "fred is here"
> fi
fred is here
```


The example in the slide sets a variable and then compares the value of the variable against a pattern. The pattern must be on the right side of the operator.

Incorrect quoting is shown in the following example:

```
$ name=fred

$ if [ "$name" == "f*" ] ; then
> echo "fred is here"
> fi
```

Pattern Match Metacharacters: Example

```
$ cat monthcheck
#!/bin/bash
# Script name: monthcheck
mth=$(date +%m)
if (( mth == 2 ))
then
        echo "February usually has 28 days."
        echo "If it is a leap year, it has 29 days."
elif [[ $mth = @(04|06|09|11) ]]
then
        echo "The current month has 30 days."
else
        echo "The current month has 31 days."
fi

$ date
Fri Mar 13 13:28:24 GMT 2000

$ ./monthcheck.sh
The current month has 31 days.
```

ORACLE

This example script:

- Uses the `date` statement to set the variable `mth` to an integer representing the current month (`01` is January, `12` is December).
- Tests variable `mth` value:
  - If the current month is February (`mth` is `02`), it prints messages concerning the number of days in February.
  - If the current month is April, June, September, or November, it prints the message: `The current month has 30 days.`
  - If the current month is January, March, May, July, August, October, or December—all have 31 days—it uses the `else` statement.

The specific syntax used in the example requires that the value of `mth` exactly match one of the specified patterns.

Positional Parameters

Parameter Name	Description
`$0`	The name of the script
`$1`	The first argument to the script
`$2`	The second argument to the script
`$9`	The ninth argument to the script
`$#`	The number of arguments to the script
`$@`	A list of all arguments to the script
`$*`	A list of all arguments to the script

ORACLE

The execution of scripts is made versatile by the ability to run a script based on arguments supplied on the command line. In this way, the operation of the script varies depending on what arguments are given to the script. The shell automatically assigns special variable names, called *positional parameters*, to each argument supplied to a script on the command line.

The positional parameter names and meanings are displayed in the table in the slide.

Positional Parameters: Example

```
$ cat numtest.sh
    #!/bin/bash
    # Script name: numtest.sh
    num1=5
    num2=6
    if (( $num1 > $num2 ))
    then
            print "num1 is larger"
    else
            print "num2 is larger"
    fi

$ ./numtest.sh
    num2 is larger
```


The example script in the slide captures the command-line arguments in special variables: `$#`, `$1`, `$2`, `$3`, and so on. The `$#` variable captures the number of command-line arguments.

Positional Parameters: Example

```
$ cat argtest.sh
    #!/bin/bash
    # Script name: argtest.sh
    if (( $1 > $2 ))
    then
            print "num1 is larger"
    else
            print "num2 is larger"
    fi

$ ./argtest.sh 21 11
    num1 is larger

$ ./argtest.sh 1 11
    num2 is larger
```


This example in the slide accepts command-line arguments.

Positional Parameters: Example

- The script should verify that the type of input and the number of values input are correct.
- If they are incorrect, then the script can print an error message in the form of a `USAGE` message.
- Example:

```
if (( $# != 2 ))
then
print "USAGE: $0 arg1 arg2 "
exit
fi
```


When you write a script and set up user interaction, generally the script expects a certain type of user input. For example, if the script prints a request for an age, it expects you to enter a numeric value. If you enter a name instead, the script might attempt an arithmetic operation on the variable and get an error.

In the example in the slide, the script expects two positional parameters to be entered when the script is run. If the two parameters are not provided, the script prints out the `USAGE` message.

Including the `USAGE` message is a good programming practice. It is the conventional way to notify that the script was not executed correctly. Most operating system commands provide this type of information when you try to execute a command with an undefined option, such as `ls -z`.

```
$ ls -z
ls: illegal option -- z
usage: ls -1RaAdCxmnlhogrtuvVcpFbqisfHLeE@ [files]
```

Boolean AND, OR, and NOT Operators

- Boolean operators define the relationships between words or groups of words.
- Following are the Boolean operators:
  - AND operator is &&
  - OR operator is ||
  - NOT operator is !

The operators && and || can occur both outside and inside ((...)) or [[...]].

The Boolean NOT is often used in testing file objects. For example, to test whether a file is not readable as opposed to readable or to test whether an object is not a regular file, use the following statements:

```
if [[ ! -r $var ]] # if $var is not readable ...
if [[ ! -f $var ]] # if $var is not a regular file...
```

Boolean AND, OR, and NOT Operators: Example

```
$ cat leap2.sh
#!/bin/bash
# Script name: leap2.sh

# Make sure the user enters the year on the command line
# when they execute the script.
if [ $# -ne 1 ]
then
fi

year=$1
echo "You need to enter the year."
exit 1
if (( ( year % 400 ) == 0 )) ||
        (( ( year % 4 ) == 0 && ( year % 100 ) != 0 ))
then
        print "$year is a leap year!"
else
        print "$year is not a leap year."
fi
```


This example shows the || operator outside ((...)) and the && operator inside ((...)). The following are the outputs of the example in the slide.

```
$ ./leap2.sh 2000
2000 is a leap year!
$ ./leap2.sh 2003
2003 is not a leap year.
```

Agenda

- Describing the various forms of `if` statements and their role in testing conditions
- Explaining the use of `if` statements through examples
- Describing the role of the `case` statement in choosing from alternatives

The `case` Statement

The `case` statement helps choose from several alternatives.

```
case value in
pattern1)
        statement1
        ...
        statementn
        ;;
pattern2)
        statement1
        ...
        statementn
        ;;
*)
        statement1
        ...
        statementn
        ;;
esac
```

ORACLE

The `if/then/else` construct makes it easy to write programs that choose between two alternatives. Sometimes, however, a program needs to choose one of several alternatives. You could do this by using the `if/then/elif/else` construct, but in many cases it is more convenient to use the `case` statement. The syntax for the `case` statement is shown in the slide.

In the syntax, *value* refers to any value, but it is usually the value of some variable. The patterns used in the `case` statement can be any pattern, including shell metacharacters.

Include two semicolon terminators (that is, `;;`) after the last statement in the group for each pattern. This prohibits the shell from *falling through to* the statements for the next pattern and executing them. Think of `;;` as saying “Break out of the `case` statement now and go to the statement that follows the `esac` statement.”

The last pattern in a `case` statement is often the `*` metacharacter. If the value of the variable being tested does not match any of the other patterns, it always matches the `*` metacharacter. This is often called the *default pattern match*, and it is analogous to the `else` statement in the `if` constructs. The patterns used in the `case` statement can be any pattern, including shell metacharacters.

The `case` Statement: Example

```
$ cat case.sh
#!/bin/bash
# Script name: case.sh
mth=$(date +%m)
case $mth in
02)
        print "February usually has 28 days."
        print "If it is a leap year, it has 29 days."
        ;;
04|06|09|11)
        print "The current month has 30 days."
         ;;
*)
        print "The current month has 31 days."
        ;;
esac
```


The first two patterns in the example within the `case` statement are the string patterns that the variable `mth` is compared against.

The last pattern in the `case` statement is the * metacharacter. If `mth` does not match the first two patterns, it always matches this one.

The `case` Statement: Example

```
$ date
Thursday, March 6, 2014 01:09:38 PM IST

$ ./case.sh
The current month has 31 days.
$
```

A more familiar example is a boot script, such as the `/etc/init.d/volmgt` script. In many boot scripts, the first argument to the script is checked in a `case` statement to determine whether the daemon should be started or stopped. In the following example, the `case` not only searches for the patterns `start` and `stop`, but it also uses the default * metacharacter, which it executes when something is typed other than `start` or `stop`.

```
$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# ident "@(#)volmgt     1.9     06/01/20 SMI"
case "$1" in
'start')
        if [ -f /etc/vold.conf -a -f /usr/sbin/vold -a \
            "${_INIT_ZONENAME:=`/sbin/zonename`}" = "global" ];
   then
                echo 'volume management starting.'
                svcadm enable svc:/system/filesystem/volfs:default
        fi
        ;;
'stop')
        svcadm disable svc:/system/filesystem/volfs:default
        ;;
*)
        echo "Usage: $0 { start | stop }"
        exit 1
        ;;
esac
exit 0
```

Replacing Complex `if` Statements with `case` Statements

```
$ cat snoopy2.sh
#!/bin/bash
# Script name: snoopy2.sh
name=snoopy
if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
        echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
        echo "The doctor is in."
elif [[ "$name" == "schroeder" ]]
then
        echo "In concert."
else
        echo "Not a Snoopy character."
fi
```

ORACLE

When you have many alternatives to check a variable against, the syntax for an `if/then/elif/else` can get very confusing. Many shell programmers use the `case` statement in these situations.

The `snoopy` example displayed in the slide uses the `if/then/elif/else` statement.

Replacing Complex `if` Statements with `case` Statements

```
$ cat snoopy3.sh
#!/bin/bash
# Script name: snoopy3.sh
name=lucy
case $name in
"snoopy")
        echo "It was a dark and stormy night."
        ;;
"charlie")
        echo "You're a good man Charlie Brown."
        ;;
"lucy")
        echo "The doctor is in."
        ;;
"schroeder")
        echo "In concert."
        ;;
*)
echo "Not a Snoopy character."
;;
esac
```


The snoopy example written in the `if/then/elif/else` construct can be rewritten with an easier-to-read `case` statement as displayed in the slide.

The following example is compressed. It performs the same operations as the previous example; however, it places several statements on the same line and does not have blank lines.

```
#!/bin/bash
case $a in
"lucy") echo "The doctor is in." ;;
*) echo "Not a Snoopy character" ;;
esac
```

Summary

In this lesson, you should have learned how to:

- Describe the various forms of `if` statements and their role in testing conditions
- Explain the use of `if` statements through examples
- Describe the role of the `case` statement in choosing from alternatives

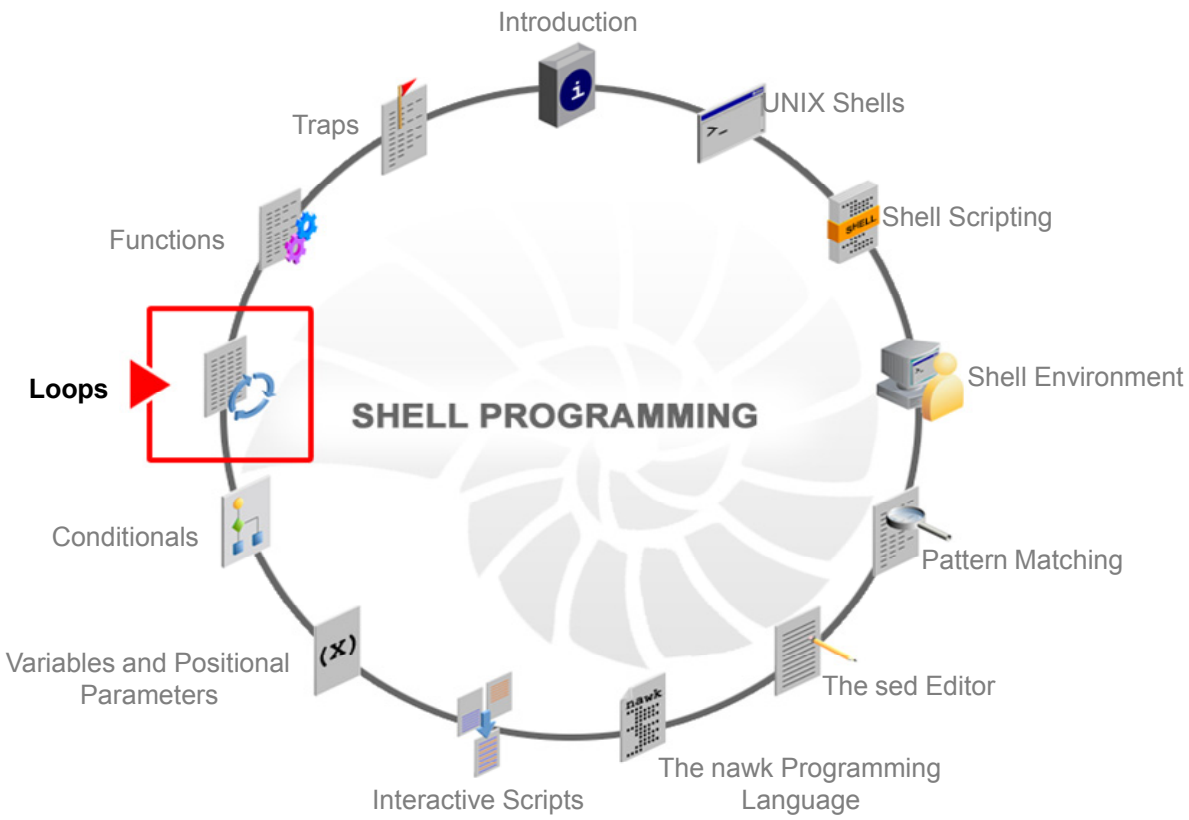
Practice 10 Overview: Conditionals

This practice covers the following topics:

- Using Conditionals
  - You answer questions about the usage of conditionals and you create a script that will determine whether the pathname is a directory or a file.
- Modifying the `adduser` Script by Using Conditionals
  - You modify the existing `adduser` script to add more functionality with the use of conditionals to create a new user.

11

Loops



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
```


In the syntax for the `for` loop shown in the slide:

- *`var`* is any variable name
- *`argument_list`* can be any list of words, strings, or numbers, and it can be literal or generated by using a shell command or shell command-line metacharacter
- The statements are any operating system commands, a user program, a shell script, or a shell statement that returns or produces a list

The value of the variable *`var`* is set to the first word in the list the first time through the loop. The second time through the loop, its value is set to the second word in the list, and so on. The loop terminates when *`var`* has taken on each of the values from the argument list, in turn, and there are no remaining arguments.

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
```

ORACLE

An explicit list is the simplest form of an argument list. The loop is executed once for each list element.

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.
```


When the arguments are in a variable and the statement is executed, the variable contents are substituted. If the variable is empty, the loop exits.

```
for var in $var_sub
```

In the example shown in the slide, the text entered at the prompt becomes the value of the variable `INPUT`. This variable represents the argument list of the `for` construct. Therefore, the text in `INPUT` is broken into words or tokens based on white space.

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.
```


In the example in the slide, the text entered on the command line becomes the argument list for the `for` construct. Therefore, the command line text is broken into words or tokens based on white space.

```
for var in (the_command-line_arguments)
```

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
```


The example in the slide uses the output of the `cat` command as the argument list.

If the file containing the list were in the following format, what is the result of the `for` loop?

```
$ cat fruit3
Apple Orange Banana Peach Kiwi

$ cat ex3.sh
#!/bin/bash
# Script name: ex3.sh
for var in $(cat fruit3)
do
        print  "$var"
done
```

In the above example, the results do not change. The `IFS` variable is any white space, so both a carriage return and a space or tab separate each field in the file.

If the file containing the list were in the following format, what would you need to do to use each item as a separate argument?

```
$ cat fruit4
apple:orange:banana:peach:kiwi

$ cat ex6.sh
#!/bin/bash
# Script name: ex6.sh
for var in $(cat fruit4)
do
        print  "$var"
done
$ ./ex6.sh
apple:orange:banana:peach:kiwi
```

In the above example, the `IFS` variable must be set to include a colon.

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
```


The `for.sh` example script uses the output of an `ls` command to create the argument list in the `for` statement.

```
$ cat for.sh
#!/bin/bash
# Script name: for.sh
print "Subdirectories in $(pwd):"
for fname in $(ls)                  # using command substitution to
do                                  # generate an argument list
        if [[ -d $fname ]]
        then
                print $fname
        fi
done
```

```
$ cd /usr/share
$ /opt/ora/examples/les11/for.sh
Subdirectories in /usr/share:
A2ps
Aclocal
Alacarte
Anthy
Application-registry
…
…
Zenity
zsh

$ cd
$ /opt/ora/examples/les11/for.sh
Subdirectories in /home/oracle:
Desktop
Documents
Download
Public
```

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*
```


Metacharacters are frequently used when specifying file names as the argument list. For example:

```
for var in a* ab??? s*[m-z]
```

If you use file-name metacharacters to generate some or all of a list of words and there are no files that match the pattern given, the shell does not expand the pattern. That is, if no file names match the metacharacter pattern given in the list in a `for` statement, the metacharacter pattern is taken literally as a value in the list.

Consider the following example:

```
for x in z*
do
        print "Working on file $x."
        <other statements>
done
```

The previous example is intended to work on files in the current directory that begin with the letter `z`; however, if the current directory does not contain any files beginning with the letter `z`, then there is only one element in the list for the `for loop: z*` construct. The statements in the `for` loop are executed only once, and the `print` statement prints:

```
Working on file z*.
```

The statements following the `print` statement result in an error message from the shell if they are statements that use the file `z*` as an argument. No such file exists in the current directory, so the statements fail.

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

Answer: c

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.

When a `while` statement is executed, the *`control_command`* is evaluated. If the *`control_command`* succeeds, all the statements between `do` and `done` are executed, and then the controlling command is again executed.

As long as the *`control_command`* succeeds, the loop body continues to execute.

As soon as the *`control_command`* fails, the statement following the `done` statement is executed.

The `do` statement is a separate statement and must appear on a line by itself (not on the line containing the `while` *`control_command`* statement) unless it is preceded by a semicolon. The same is true of the `done` statement. It must exist on a line of its own or be preceded by a semicolon if it exists at the end of the last statement line in the loop.

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 ))
```


Just as in the `if` statement, the controlling command of a `while` loop is often a `((...))` or `[[ ... ]]` or `[ ... ]` command. Frequently, a variable within the test changes value in the `while` loop so that the loop eventually terminates.

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 two examples show how you can alter numeric values in the loop body to cause the `while` loop to terminate.

There are two commands in the shell that always return the same exit status. These commands are `true` and `false`. The `true` command always causes the test to succeed (with zero errors), and the `false` command always causes the test to fail (with some number of errors).

The following example causes an infinite loop:

```
while true
```

To debug your script by forcing the condition to be false, use the following:

```
while false
```

Note: The null statement (:) always evaluates to true; therefore, the statements `while :` and `while ((1))` are interchangeable.

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.
$
```


In the example shown in the slide, the variable `num` is initialized to `1`, and a `while` loop is entered.

This `while` loop uses `num` instead of `$num` because the `((...))` command automatically does variable expansion.

Using `num` forces integer arithmetic, whereas using `$num` performs arithmetic on strings. Integer arithmetic is faster.

Within the `while` loop, the current value of `num` is printed to `stdout`. The variable `num` is incremented and the condition in the `while` statement is checked again. If the value of the variable `num` did not change within the `while` loop, the program would be in an *infinite* loop (a loop that never ends).

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."
```


You can place the `read` statement in the `while` condition. Each time a user inputs something from a keyboard, a loop executes. When a user presses Control-D, this tells the `read` statement that input is complete, the condition becomes false, and execution jumps to after the `done` statement. The output of the example in the slide is as below:

```
$ ./readinput.sh
Enter a string: OK
Keyboard input is: OK

Enter a string: This is fun
Keyboard input is: This is fun

Enter a string: I'm finished.
Keyboard input is: I'm finished.

Enter a string:^d 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
```


Instead of the input coming from the keyboard or `stdin`, you can also specify a file from which to read the input. In the example in the slide, the redirection is performed at the end of the `while` block:

```
done < phonelist
```

By redirecting on `stdin`, the file is opened only once. Each time around the loop, `read` returns the next line from the file. Lines are read into the loop while there are lines to read in from the file. After the end of the file is reached, a true state no longer exists, and the loop terminates. After the loop terminates, the statement that follows the `done` statement executes.

```
$ ./internal_redir.sh
The phone number for Claude Rains is 214-555-5107
The phone number for Agnes Moorehead is 710-555-6538
The phone number for Rosalind Russel is 710-555-0482
The phone number for Loretta Young is 409-555-9327
The phone number for James Mason is 212-555-2189
```

Note: Input redirection occurs on the line containing the word `done`. A loop is a complete command structure and any input redirection must come after the complete command structure.

Quiz

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

a. True
b. False

Answer: a

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 is very similar to the `while` loop, except that the `until` loop executes as long as the command fails.

The *`control_command`* can be any command that exits with a success or failure status. The statements can be any utility commands, user programs, shell scripts, or shell statements. If the *`control_command`* fails, the body of the loop (all the statements between *`do`* and *`done`*) executes, and the *`control_command`* executes again. As long as the *`control_command`* continues to fail, the body of the loop continues to execute. As soon as the *`control_command`* succeeds, the statement following the *`done`* statement executes.

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 is useful if the number of times the loop is executed depends on input from the user and not some predetermined number.

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 output of the example in the slide is as below:

```
$ ./break.sh
Enter any number (0 to exit): 5
Square of 5 is 25.

Enter any number (0 to exit): -5
Square of -5 is 25.

Enter any number (0 to exit): 259
Square of 259 is 67081.

Enter any number (0 to exit): 0
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.sh` example script renames files in the current directory whose names contain a capital letter to lowercase letters.

First, the script prints the name of the file on which it is currently working. If this file name contains no capital letters, the script executes the `continue` statement, forcing the shell to go to the beginning of the `for` loop and get the next file name.

If the file name contains a capital letter, the original name is saved in the *`orig`* variable. The name is also copied into the new variable; however, the line `typeset -l new` at the beginning of the script causes all letters stored in the variable to be converted to lowercase during the assignment. When this is done, the script executes an `mv` command by using the *`orig`* and *`new`* variables as arguments.

The script then prints a message concerning the name change for the file and gets the next file to be worked on.

When the `for` loop is finished, the `Done` message prints to let the user know that the script finished.

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.
```

ORACLE

Running the `ls` command after executing the script shows the following output:

```
$ ls test.dir
a           data.file       scr1            sort.data
als         recreate_names  slalk           textfile
```

Quiz

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

a. `until`
b. `continue`
c. `exit`
d. `break`

Answer: 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 `select` Statement

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

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


The variables *var* and *list* in the syntax follow the same syntactic rules used in the `for` loop (although the operation of the `select` loop is substantially different). If *list* contains metacharacters, the metacharacters are expanded into file names.

The elements in *list* are the menu choices presented to the user. The shell automatically numbers each menu choice (starting with `1`). The user must enter the number corresponding to the user's choice for the input to be valid.

Although you can use any statements within the `select` loop, the `case` statement is most often used to match the choice selected by the user and to take certain actions depending on that choice.

The input the user provides is saved in the `REPLY` variable.

If the user does not enter any input except for pressing the Return key, then the `REPLY` variable is set to the null string, and the shell redisplays the menu choices.

The `select` loop is exited when the user types the end-of-file (EOF) character, which is Control-D on most Oracle systems.

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 syntax creates a loop. A user is repeatedly presented with menu choices followed by a prompt (the value of the PS3 variable). The select loop is exited when a user types the EOF character for the system as the response to the menu prompt.

Sometimes a script produces so much output—between the time the menu list first is displayed to when the menu prompt is displayed—that a user is no longer able to see or remember the menu choices. To redisplay menu choices, a user needs to press the Return key in response to the menu prompt.

The default value of the PS3 variable in the bash shell is #?. Therefore, if you fail to set the PS3 variable, the system uses the string #? as the prompt and then waits for user input. If this happens, the user might not know that the script is waiting for input.

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 script in the slide provides five fruit choices and then prompts for user choice.

A `case` statement is used in the `select` loop to take appropriate action based on user selection. The default case, the `*` pattern, matches any input by the user that is not one of the numbers of the choices on the menu.

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
$
```


In the script output, you can see how the `select` loop works with several valid and some invalid choices.

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)
```


Users can exit the `select` loop at any time by entering the EOF character on their system. However, unless users know this or you include a message explaining this in your menu prompt, users can get stuck in the `select` loop.

Include a string in your menu choices to help users exit the menu. The string could simply say `Quit Menu`. Then the action in the `case` statement for this menu choice would be the `break` statement. The `break` statement exits the `select` loop, and execution of the script moves to the first statement following the `done` statement in the `select/do/done` syntax. If there are no statements following the `done` statement, the script terminates.

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.

The shell allows any statement within the `select` loop to be another `select` loop. This enables you to create a menu that has one or more submenus.

Include a `break` statement in an action for one or more choices on a submenu to exit a submenu loop and return to an outer menu loop.

The `PS3` variable is also used for the prompt in a submenu. Therefore, be careful when moving from one menu to another to reset this variable to the prompt for the menu to which you are moving.

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
```


The `submenu.sh` example script in the slide first creates two variables, `main_prompt` and `dessert_prompt`, to hold the prompts for the main menu and the dessert submenu, respectively.

The `select` loop for the main menu is exited when the user chooses `Order Completed`, at which time the script prints the message `Enjoy your meal`.

When the user selects the main menu choice dessert, a submenu appears, asking the user to input the number for the dessert choice.

In the `case` statement for the `select` loop for the dessert menu, the `break` statement is used for each menu choice. (The default choice does not use the `break` statement because you want the user to re-enter a dessert choice.) This exits the dessert menu. Then the menu prompt resets to the value of the *`main_prompt`* variable, and control returns to the main (outer) `select` loop.

All `print` statements in the script use single quotation marks so that the `$` in the prices is printed.

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. `#`

Answer: c

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`.

ORACLE

Sometimes a script does not require a specific number of arguments from users. Users are allowed to give as many arguments to the script as they want.

In these situations, the arguments of the script are usually processed in a `while` loop with the condition being (`($#)`). This condition is true as long as the number of arguments is greater than zero.

Doing a `shift` reduces the argument number, eventually making the condition in the `while` loop false and terminating the loop.

All positional parameters are shifted to the left by the number of positions indicated by *`num`* (or by `1` if *`num`* is omitted).

The number of left-most parameters are reduced after the shift. For example, the statement: `shift 4` eliminates the first four positional parameters. The value of the fifth parameter becomes the value of `$1`, the value of the sixth parameter becomes the value of `$2`, and so on. It is an error for *`num`* to be larger than the number of positional parameters.

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.sh` example script in the slide contains a `USAGE` message showing that the script should be run with arguments, although the number of arguments given is a variable.

If the user does not provide arguments, the `USAGE` message is printed and the script exits; otherwise, the `while` loop is entered. Each time through the loop, the current value of the positional parameter, `$1`, is printed and the `shift` statement executes.

The `shift` statement reduces the number of arguments by one as it shifts the values of the positional parameters one position to the left. (`$1` is assigned the value of `$2`, `$2` is assigned the value of `$3`, and so on, whereas the value of `$1` is discarded.)

After each argument is printed, the condition of (`($#)`) is false, and the `while` loop terminates.

The statement `print $*` shows that nothing is printed after the execution of the `while` loop because the value of `$#` is `0`.

The values of the positional parameters are set to letters of the alphabet by using the `set` statement and printed by using the statement `print $*` (to show the new values of the positional parameters).

The last two statements in the script, `print` $* and `shift`, illustrate what occurs when a larger shift is made.

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`

Answer: c

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.

In the `getopts` syntax:

- *`options`* represents the valid single-letter characters the script expects as options from the command line. If any single-letter character in *`options`* is followed by a colon (:), then the shell expects that option to have an argument.
- `var` represents any variable name whose value is an option from the command line if the option is one of the valid single-letter characters specified in the `getopts` statement

The `getopts` statement does not always process all options on the command line. It quits processing options if an argument that is not an option or an argument to an option is placed between two options. For example, suppose `scriptname` is a script with valid options `x` and `y`, neither of which requires an argument. Then the command causes the script to process the `x` option but not the `y` option:

```
scriptname -x filex –y
```

Note: When using the `print $*` or `print $@` statements, you can get an error message if the first argument to the script is an option beginning with a `-` sign. Thus, it is safer to use

`print -- $* or print -- $@`

When using the `echo $*` or `echo $@` statements, you do not get an error message.

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
```


The variable name used in the `case` statement is the name of the variable specified in the `getopts` statement. In the example code on the screen:

- The variable name is `opt_char`.
- If `-x` is given as an option to the script, the value of `opt_char` is `x`.
- If `+x` is given as an option to the script, the value of `opt_char` is `+x`.

The `while` loop is exited when there are no more options to process.

Note: Do not use the `shift` statement on command-line arguments until all options to a script are processed. If you use the `shift` statement within the `while` loop that contains the `getopts` statement, the `getopts` statement does not process correctly.

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

ORACLE

Unless you program the getopts statement to look for invalid options and to handle an invalid option when it encounters one, the shell gives its own error message.

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
```


If the script uses the `case` statement to match the options from the command line, then the `\?` pattern is used to match the value of `opt_char` when an invalid option is encountered. The backslash is necessary to escape the meaning of the `?` metacharacter.

Executing the example script gives the following output:

- `$ ./getoptsex.sh -y -x -z`
- `You entered the y option`
- `You entered the x option`
- `z is not a valid option.`
- `usage: ./getoptsex.sh -x -y`

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.

When the script is executed, the options that require an argument must be followed by their arguments before another option can be given. For example, if `x` and `y` are valid options in a script, with `x` requiring an argument, then the following commands are all valid ways of invoking the script:

```
scriptname -x x_arg -y
scriptname -xx_arg -y
scriptname -yx x_arg
scriptname -yxx_arg
```

The argument to an option is placed in the `OPTARG` variable so that it can be accessed by the statements in the `case` statement for the corresponding option.

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
```

ORACLE

In the example in the slide, the `USAGE` variable holds the value of how the script is to be invoked. The `year` variable holds the value of the current year. The `getopts` statement within the `while` loop specifies that this script has two valid options: `d` and `m`. The `m` option is followed by a colon (`:`), so it is required to have an argument supplied to it on the command line. The beginning colon in `:dm:` tells the `getopts` statement to watch for invalid switches. (It also tells `getopts` to watch for options on the command line that require an argument but have no argument supplied to them.)

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
```


On executing the `getopts1.sh` script with various entries, you get the output shown in the slide.

In the first output example, the `-d` option is correctly processed. Then the invalid `-k` option is encountered. The `getopts` statement sets the value of `opt_char` to a `?`. Then the `\?)` case matches the value in `opt_char` and the corresponding statements are executed.

In the second output example, although the `-m` option is valid, but the option requires an argument to perform its action. Remember that the colon after the `m` in `getopts :dm:` states that it requires an argument following it on the command line. If there is no argument, the `m)` case is not matched and processed.

In the last output example, two arguments are given to the script: `-d` and `filex`. Observe that `filex` is not preceded with a `-` or a `+` sign and, therefore, it is not processed as an option to the script.

Note: If `filex` had been the first argument given to the script, *no* options would have been processed.

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 `getopts2.sh` example script is the `getopts1.sh` script with an added pattern (`:`) in the `case` statement. This added pattern is matched when an option that requires an argument is given on the command line but is not followed with its argument.

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]
```


In the example output, the `getopts2.sh` script is executed and given the `-m` option without an argument. Observe the output. Otherwise, the script runs the same way as the `getopts1.sh` script.

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

Answer: b

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

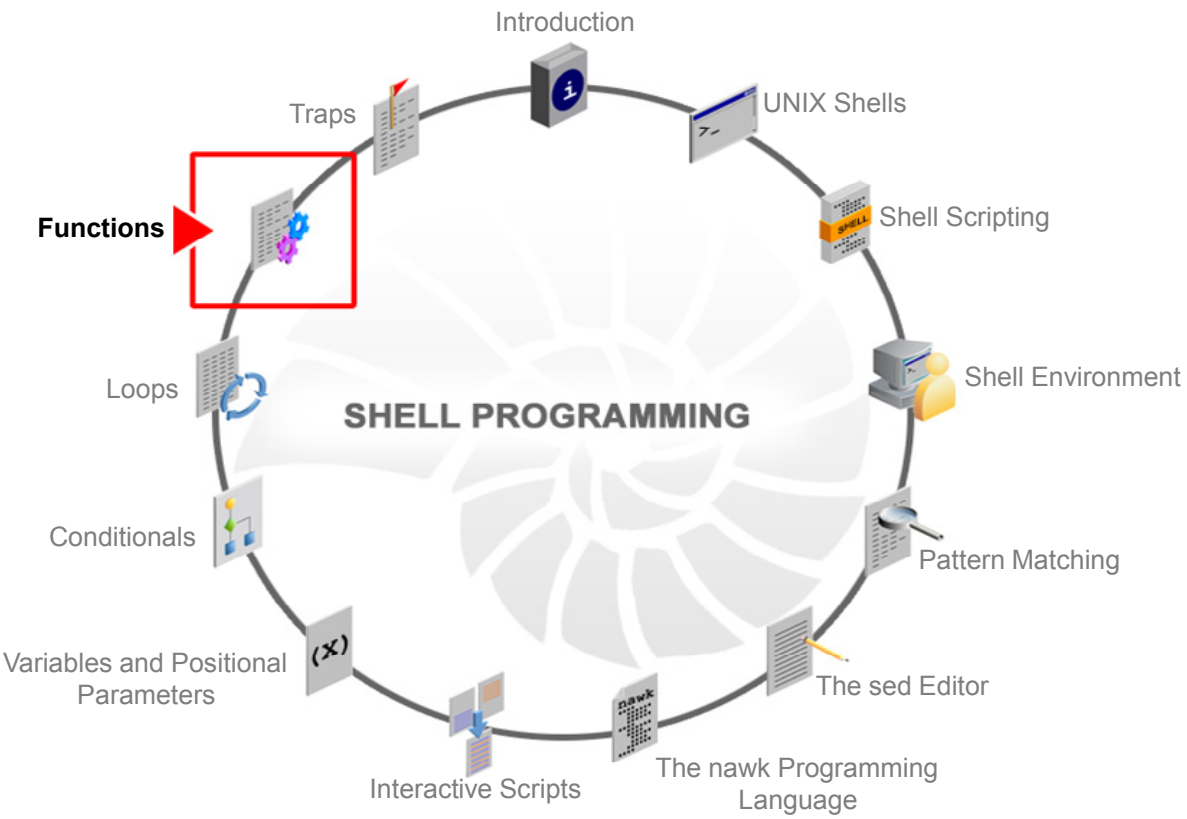
Practice 11 Overview: Loops

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.

12

Functions



ORACLE

Objectives

After completing this lesson, you should be able to:

- Create user-defined functions in a shell script
- Use the `typeset` and `unset` statements in a function
- Autoload a function file into a shell script

Agenda

- Creating user-defined functions in a shell script
- Using the `typeset` and `unset` statements in a function
- Autoloading a function file into a shell script

Functions in a Shell

- A function is a set of one or more statements that act as a complete routine.
- Each function must have a unique name within a shell or shell script.
- Syntax:

```
function function_name [ block_of_statement_lines]
```


A function is executed within the shell in which it has been declared; the function is executed as a subroutine of the shell. It is not executed as a subprocess of the current shell. After a function has been loaded into the current shell, it is retained.

Functions in a Shell: Example

```
#!/bin/bash

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

Positional Parameters and Functions

- Functions act like miniscripts, in that they can:
  - Accept parameters that are passed to them
  - Use local variables
  - Return values back to the calling shell command line
- Positional parameters passed to functions are not the same positional parameters that are passed to a script.
- The examples in the following slides illustrate that difference.

Positional Parameters and Functions

```
cat funparas.sh
#!/bin/bash
# Script name: funparas.sh
function hello
{
      print '$1 the function is: ' $1
}
print 'Input passed and stored in $1 is: ' $1

hello John   # execute the function hello

print
print 'After the function $1 is still ' $1

$ ./funparas.sh Susan
Input passed and stored in $1 is: Susan
$1 in the function is: John
After the function $1 is still Susan
```

Positional Parameters and Functions

```
$ cat ~/.bashrc

killit ()    # Bash shell syntax
{
  pkill -u $1
}


function rcgrep     # Bash Shell syntax
{
  grep $1 /etc/init.d/* |more
}


rgrep ()     # Bash shell syntax
{
  find $2 -type file -exec grep $1 {} \; | more
}
```


The example in the slide shows a customized `.bashrc` file that has three functions: `killit`, `rcgrep`, and `rgrep`. This following examples show the execution of each of the three functions.

Positional Parameters and Functions

```
$ killit oracle
Note: This will kill all the 'oracle' user processes.

$ rcgrep sed
/etc/init.d/pppd:      sed -e
's/^#.*//;s/\([^\\]\)#.*/\1/;s/[ ]*$//;s
/^[   ]*//' \
/etc/init.d/pppd:      sed -e
's/^#.*//;s/\([^\\]\)#.*/\1/;s/[ ]*$//;s
/^[   ]*//' \
/etc/init.d/README:scripts. The S* scripts should only be
used for cleanup during
```


The first example in the slide calls the `killit` function, which calls `pkill` to kill all processes owned by the user specified as the first argument to the function. It then prints a message that is logged to a file.

The second example executes a `grep` search through all the files in the `/etc/init.d` directory for the string passed in as the first argument to the `rcgrep` (recursive `grep`) function.

Positional Parameters and Functions

```
$ rgrep root /etc/default
# If CONSOLE is set, root can only login on that device.
# If the specified device is /dev/console, then root can
also log into
# Comment this line out to allow remote login by root.
# SUPATH sets the initial shell PATH variable for root
# to log all root logins at level LOG_NOTICE and multiple
failed login
# CONSOLE determines whether attempts to su to root should
be logged
# SUPATH sets the initial shell PATH variable for root
# root, LOG_INFO messages are generated for su's to other
users, and LOG_CRIT
```


The example in the slide performs a `rcgrep` search. Two arguments are passed to the `rcgrep` function: the first is the search string, and the second is the directory from which to begin the search. The `find` statement recursively finds all files under that directory, and passes each file name to the `grep` statement, which then searches for the search string within each file. Therefore, you can apply `grep` to the contents of the files, rather than to the file names.

Return Values

- The `return` statement terminates the function and passes a value back to the calling shell or script.
- The `return` statement returns any designated value between `0` (zero) and `255`.
- By default, the value passed by the `return` statement is the current value of the `?` exit status variable.

Note: A value can pass from a function back to the shell or script that called that function.

Quiz

Positional parameters passed to functions are not the same positional parameters that are passed to a script.

a. True
b. False

Answer: a

Agenda

- Creating user-defined functions in a shell script
- Using the `typeset` and `unset` statements in a function
- Autoloading a function file into a shell script

The `typeset` and `unset` Statements

- The following shows the syntax for using `typeset` and `unset` with functions:
  - `typeset -f`: Lists the known functions and their definitions
    - `functions` is an alias for `typeset -f`
  - `typeset +f`: Lists the known function names
  - `unset -f` *name*: Unsets the value of the function
- The examples in the following slides use `typeset` and `unset` with functions.

The `typeset` and `unset` Statements

```
$ typeset -f
function killit
{
pkill -u $1
print -n "Had to kill process for user: $1 "
print "on $(date +%D) at $(date +%T)"
# The previous print statement may be appended to a log
file.
}
function rcgrep
{
grep $1 /etc/init.d/* |more
}
function rgrep
{
find $2 -type file -exec grep $1 {} \; | more
}
```

The `typeset` and `unset` Statements

```
$ typeset +f
killit
rcgrep
Rgrep

$ alias | grep fun
functions='typeset -f'

$ unset -f rcgrep

$ typeset +f
killit
rgrep
```


The function in the slide echoes its purpose and the line it will execute.

Agenda

- Creating user-defined functions in a shell script
- Using the `typeset` and `unset` statements in a function
- Autoloading a function file into a shell script

Function File

- You can create functions within a shell script, or they can be external to the shell script, such as in a file.
- Only one function can be in each function file.
- A function file can be autoloaded into a shell script and used by that script.

You can create functions within a shell script, or they can be external to the shell script. Functions created within a shell script exist only within the shell interpreting that script. You can put functions in a file. When you create a function, it must contain the definition of only one function. The name of a function and function file must be the same. Make a function file an executable. A function file can be autoloaded into a shell script and used by that script. Only one function can be in each function file. For example:

```
$ cat holder
function holder
{
print
print -n "Type some text to continue: "
read var1
print "In function holder var1 is: $var1"
}
```

Autoloading a Function File

- To autoload a function file:
  - Declare the `FPATH` variable before any command lines attempt to invoke a function from one of the function files

```
$ FPATH=$HOME/function_dir ; export FPATH
```

  - The directories listed in the `FPATH` environment variable should contain only function files
- By using the `FPATH` variable, the functions can be autoloaded into a shell script and do not need to be declared in every script.

A function is treated in the same way as a bash shell built-in statement. When the function is invoked, it is invoked in the current shell and not as a subprocess.

Before a function is invoked, the shell must know that the function exists. Therefore, you must define functions at the start of the shell script before any command-line command attempts to invoke the function.

If a function was created as a function file, you must create or modify the `FPATH` variable to include the directory name that contains the function file. Declare the `FPATH` variable before any command-line command attempts to invoke a function from one of the function files.

Note: The `FPATH` variable is similar to the `PATH` variable. It can contain the names of one or more directories, each separated by colons. The files that exist in the `$HOME/function_dir` directory should all be function files. Ensure that no other type of files reside in that directory.

Autoloading a Function File

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

FPATH=./funcs
export FPATH

print "Calling holder..."
holder

print
print "After the function var1 is: $var1"

$ ./holdertest.sh
Calling holder...
Type some text to continue: shell scripts
In function holder var1 is: shell scripts
After the function var1 is: shell scripts
```


Note: By updating a function file, any scripts that use that function autoload the updated version of the function file when the scripts are next invoked.

Summary

In this lesson, you should have learned how to:

- Create user-defined functions in a shell script
- Use the `typeset` and `unset` statements in a function
- Autoload a function file into a shell script

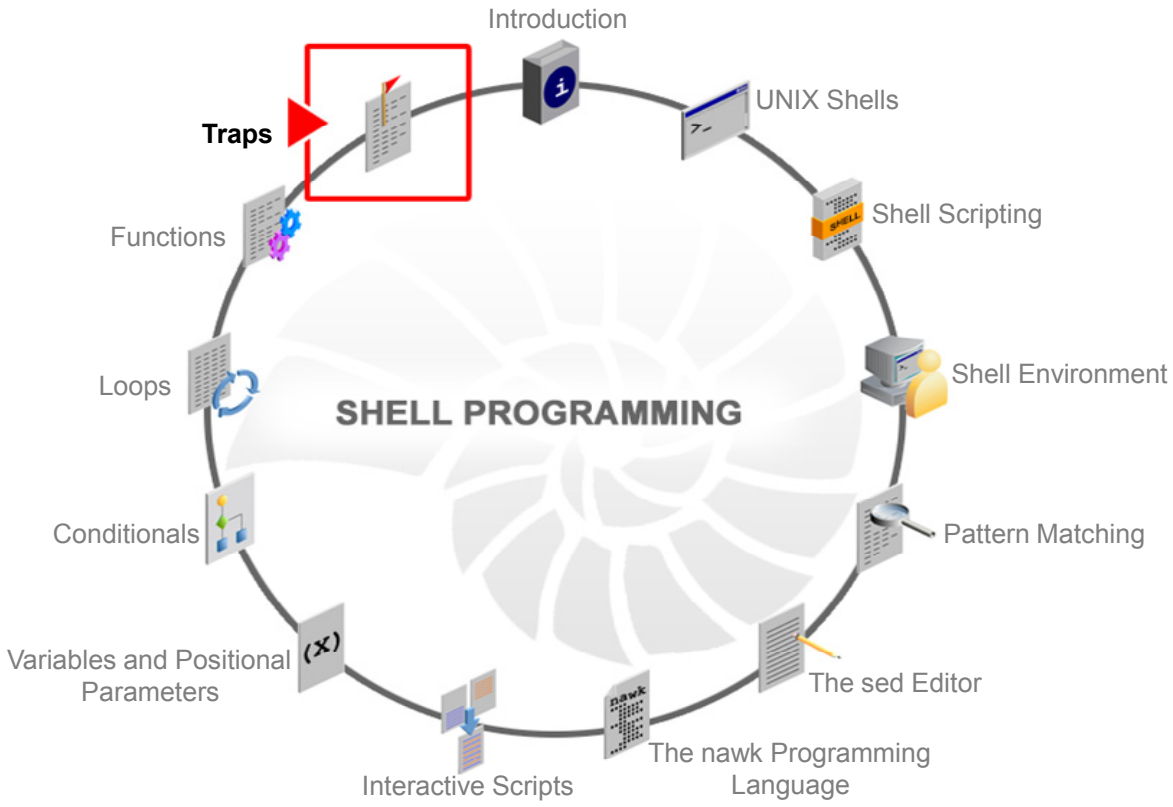
Practice 12 Overview: Functions

This practice covers the following topic:

- Using Functions

13

Traps



Objectives

After completing this lesson, you will be able to:

- Describe the role of shell signals in interprocess communication
- Catch signals and user errors with the `trap` statement

Agenda

- Describing the role of shell signals in interprocess communication
- Catching signals and user errors with the `trap` statement

Shell Signals

- A shell signal is a message sent from one process to another indicating abnormal event.
- You can also send your own signals through:
  - Keyboard sequences such as `Control-C`
  - The `kill` command at the shell level

Typically, a process sends a signal to one of its own subprocesses. You can obtain more information on signals from the `man` pages of `signal` and `kill`.

Signals Using Keyboard Sequence

The following are some signals that can be sent from the keyboard:

- Signal `2` (`INT`) by pressing `Control-C`
- Signal `3` (`QUIT`) by pressing `Control-\`
- Signal `23` (`STOP`) by pressing `Control-S`
- Signal `24` (`TSTP`) by pressing `Control-Z`
- Signal `25` (`CONT`) by pressing `Control-Q`

The `INT` and `QUIT` signals cause the currently running process associated with the device (console or window) to terminate.

The `TSTP` signal causes the process to stop or suspend execution. The job is put into the background; the process can be resumed by putting the job into the foreground.

Signals Using the `kill` Command

- You can send a signal to processes running on the system by using the `kill` command.

```
kill -signal pid
```

- For example, the `-9` option sends the `KILL` signal to the process.

```
kill -9 pid
kill -KILL pid
```

Note: When you do not specify a signal name or number, the `TERM` signal, signal `15`, is sent to the process.

Note: The `root` user can send any signal to any process. Other users can only send signals to processes they own.

Shell Signal Values

- The bash shell has 72 defined signals.
- Each signal has a name and a number associated with it.
- The following is a list of shell signal values generated by the `kill -l` command:

```
$ kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE    14) SIGALRM     15) SIGTERM     16) SIGUSR1
17) SIGUSR2    18) SIGCHLD     19) SIGPWR      20) SIGWINCH
21) SIGURG     22) SIGIO       23) SIGSTOP     24) SIGTSTP
25) SIGCONT    26) SIGTTIN     27) SIGTTOU     28) SIGVTALRM
29) SIGPROF    30) SIGXCPU
..... (output omitted)
```


The numeric signal values range from `0` (`EXIT`) through `72` (`RTMAX`). You can confirm the numeric value of a signal by executing the `kill` command with the `-l` option, followed by the signal name. For example:

```
$ kill -l EXIT
0
$ kill -l RTMAX
72
$ kill -l KILL
9
$ kill -l TSTP
24
```

Quiz

Which of the following signals is sent to a process when you do not specify a signal name or number with the `kill` command?

a. Signal 2 (`INT`)
b. Signal 3 (`QUIT`)
c. Signal 15 (`TERM`)
d. Signal 23 (`STOP`)
e. Signal 24 (`TSTP`)
f. Signal 25 (`CONT`)

Answer: c

Agenda

- Describing the role of shell signals in interprocess communication
- Catching signals and user errors with the `trap` statement

The `trap` Statement

- Most signals sent to a process executing a shell script cause the script to terminate.
- You can use the `trap` statement to avoid having the script terminate from specified signals.
- Syntax:

```
trap 'action' signal [ signal2 ... signalx ]
```

- Example:

```
trap 'echo "Control-C not available"' INT
```


The syntax for using the `trap` statement is:

```
trap 'action' signal [ signal2 ... signalx ]
```

Where:

- *action* is a statement or several statements separated by semicolons. If an action is not provided, then no action is performed, but the signal is still "trapped." Enclose the action within a pair of single quotation marks.
- *signal* is the name or number for the signal to be caught

The statements to be executed can be more than one line, as long as the closing quotation mark is followed by the signal value or values to be trapped. For example:

```
trap 'echo "Control-C not available"
echo "Core dumps not allowed"
sleep 1
continue' INT QUIT
```

The `trap` statement does not work if you attempt to trap the `KILL` or `STOP` signals. The shell does not let you catch these two signals, thereby ensuring that you can always terminate or stop a process. This means that shell scripts can still be terminated by using the following command:

```
kill -9 script_PID
kill -KILL script_PID
```

Also, the execution of the shell script can be suspended by using the `Control-S` character because both signals can never be trapped within a bash shell script.

The following statement tells the bash shell to restore the original actions for the signal:

```
trap - signal
```

Example

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

# Script name: trapsig.sh

trap 'print "Control-C cannot terminate this script."' INT
trap 'print "Control-\ cannot terminate this script."' QUIT
trap 'print "Control-Z cannot terminate this script."' TSTP

print "Enter any string (type 'dough' to exit)."
while (( 1 ))
do
        print -n "Rolling..."
        read string
        if [[ "$string" = "dough" ]]
        then
                break
        fi
done
        print "Exiting normally"
```


The `trapsig.sh` example script in the slide sets up a `trap` statement for the signals `INT` (`Control-C`), `QUIT` (`Control-\`), and `TSTP` (`Control-Z`). The script does not illustrate any practical application; it just shows how to install signal handling for these signals.

After the `trap` statement, a `while` loop is entered that prints the string `rolling...` and waits for user input. The `while` loop and the script terminate when the user types in the `dough` string. The script cannot be terminated by pressing `Control-C`, `Control-\`, or `Control-Z`. The example output shows the script being executed. The user presses the Return key after the first `rolling ...` prompt. The user then types a `d` in response to the second `rolling . . .` prompt and an `s` in response to the third.

The users presses `Control-C`, then `Control-\`, and then `Control-Z` in response to the next `rolling...` prompts, and this causes the appropriate `trap` statements to execute.

Finally, the user types the string `dough`, and the script terminates.

Example

```
$ ./trapsig.sh
Enter any string (type 'dough' to exit).
Rolling...
Rolling...d
Rolling...s
Rolling...Rolling...Rolling...Rolling...Rolling...4
Rolling...^c
Control-C cannot terminate this script.
Rolling...^\
Control-\ cannot terminate this script.
Rolling...^z
Control-Z cannot terminate this script.
Rolling...dough
Exiting normally
$
```


This is another example of using the `trap` statement. This example uses two scripts, `parent` and `child`. The `parent` script passes the signal to the `child` script and the messages are from the script that is called.

```
$ cat parent
# trap - SIGNAL - cancel redefinitions
# trap "cmd" SIGNAL - "cmd" is executed, signal is inherited
to child
# trap : SIGNAL - nothing is executed, signal is inherited
to child
# trap "" SIGNAL - nothing is executed, signal is not
inherited to child
```

```
#!/bin/ksh
#
# Script name: parent
trap "" 3 # ignore, no forward to children
trap : 2 # ignore, forward to children
./child
```

```
$ cat child
#
# Script name: child
trap 'print "Control-C cannot terminate this script."' INT
trap 'print "Control-\ cannot terminate this script."' QUIT
while true
do
echo "Type ^D to exit..."
read || break
done
```

```
$ ./parent
Type ^D to exit...
^\
Type ^D to exit...
^C
Control-C cannot terminate this script.
Type ^D to exit...
^\
Type ^D to exit...
$
```

Catching User Errors

- In addition to catching signals, you can use the `trap` statement to take specified actions when an error occurs in the execution of a script.
- The syntax for this type of `trap` statement is:

```
trap 'action' ERR
```

- The value of the `$?` variable in the script indicates when the `trap` statement is to be executed.
- It holds the `exit` (error) status of the previously executed command or statement.
- The `trap` statement is executed whenever `$?` becomes nonzero.

ORACLE

Catching User Errors

```
$ cat traperr1.sh
#!/bin/bash
# Script name: traperr1.sh
declare -i num
while [ 1 ]
do
        echo "Enter any number ( 0 to exit ):"
        read num
        if [ $num -eq 0 ]
        then
                echo "Exiting normally ..."
                exit 0
        else
                print "Square of $num is $(( num * num )). \n"
        fi
Done
$ ./traperr1.sh
Enter any number (-1 to exit): r
traperr1.sh[9]: r: bad number
Square of 2 is 4.
Enter any number (-1 to exit): -1
```


The `traperr1.sh` example script requires users to enter the integer `0` if they want to exit. If a user enters an integer that is not `0`, the script prints the square of the number. The script then requests another integer until the user quits the script by entering `0`.

The example output shows the script being executed. The user enters the letter `r`. This is not an integer. The user's input is read into the variable `num`, which is declared to only hold integer data types. Without a `trap` statement, the shell prints an error message and exits the script. You can avoid this problem by using a `trap` statement, which is shown in the next section.

The ERR Signal

- The standard error messages are printed to the screen.
- You can however redirect the error messages, such that, if an error occurs, the user sees just the message that you set up with the `trap` statement.

The ERR Signal

```
$ cat trapsig2.sh
#!/bin/bash
# Script name: trapsig2. sh
declare -i num
exec 2> /dev/null
trap 'print "You did not enter an integer.\n"' ERR
while (( 1 ))
do
    print -n "Enter any number ( -1 to exit ): "
    read num
    status=$?
    if (( num == -1 ))
    then
            break
    elif (( $status == 0 ))
    then
            print "Square of $num is $(( num * num )). \n"
    fi
done
print "Exiting normally"
```


The `trapsig2.sh` example script is the `traperr1.sh` script rewritten with a `trap` statement, a test of the `exit` status, and an `exec` statement to redirect standard error messages to `/dev/null`.

The value of `$?` is saved in the `status` variable immediately after the `read` statement is executed so that you can check later whether the `read` statement successfully read an integer.

The `if` statement checks to see whether the user entered `-1`. If so, the script breaks out of the `while` loop and terminates. If the user did not enter `-1` and the value of status is `0` (indicating that the `read` statement read an integer), the square of the integer entered by the user is printed and the user is again prompted for a number. If the user does not enter an integer, the value of `$?` is nonzero and it is trapped. The `print` statement within the `trap` statement is then executed. Control then returns to the `while` loop, prompting the user for another number.

The ERR Signal

```
$ ./trapsig2.sh
Enter any number ( -1 to exit ): 3
Square of 3 is 9.

Enter any number ( -1 to exit ): r
You did not enter an integer.

Enter any number ( -1 to exit ): 8
Square of 8 is 64.

Enter any number ( -1 to exit ): -1
Exiting normally
```


The example output shows the `trapsig2.sh` script being executed, with the user typing in an integer, then the letter `r`, another integer, and finally `-1` to exit the script. The messages output by the script for the corresponding input from the user is what is expected, based on the preceding description.

Declaring the `trap` Statement

- To trap a signal any time during execution, define the `trap` statement at the start of the script.
- To trap a signal only when certain command lines are executed, turn on the `trap` statement before the lines, and then turn off the `trap` statement after the lines.
- If a loop is being used, a `trap` statement can include the `continue` command to make the loop start again from its beginning.
- You can also trap the `EXIT` signal so that certain commands are executed only when the shell script is being terminated with no errors.

For example, if a shell script created temporary files, you could ensure that these are removed by using a trap of the `EXIT` signal value.

```
trap 'rm -f /tmp/tfile* ; exit 0' EXIT
```

Note: The preceding example includes an `exit` statement within the list of statements to be executed when the trap occurs. This is necessary to exit the script.

Declaring the `trap` Statement

```
#ident "@(#)profile 1.18 98/10/03 SMI "/* SVr4.0 1.3 */
# The profile that all logins get before using their own
.profile.
trap "" 2 3 # trap INT (Control-C) and QUIT (Control-\)
# and give no feedback
export LOGNAME PATH
if [ "$TERM" = "" ]
then
        if /bin/i386
        then
                TERM=sun-color
        else
                TERM=sun
        fi
        export TERM
fi
```


The example in the slide is a copy of the `/etc/profile` file with some added comments to explain the various `trap` statements.

Declaring the `trap` Statement

```
# Login and -su shells get /etc/profile services.
# -rsh is given its environment in its .profile.
case "$0" in
-sh | -ksh | -jsh)
if [ ! -f .hushlogin ]
then
/usr/sbin/quota
# Allow the user to break the Message-Of-The-Day only.
# The user does this by using Control-C (INT).
# Note: QUIT (Control-\) is still trapped (disabled).
trap "trap '' 2" 2
/bin/cat -s /etc/motd
trap "" 2 # trap Control-C (INT) and give no feedback.
```

Declaring the `trap` Statement

```
/bin/mail -E
case $? in
0)
echo "You have new mail."
;;
2)
echo "You have mail."
;;
esac
fi
esac
umask 022
trap 2 3 # Allow the user to terminate with Control-C (INT) or
# Control-\(QUIT)
```

Quiz

The `trap` statement does not work if you attempt to trap the `KILL` or `STOP` signals.

a. True
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Describe the role of shell signals in interprocess communication
- Catch signals and user errors with the `trap` statement

Practice 13 Overview: Traps

This practice covers the following topic:

- Using Traps