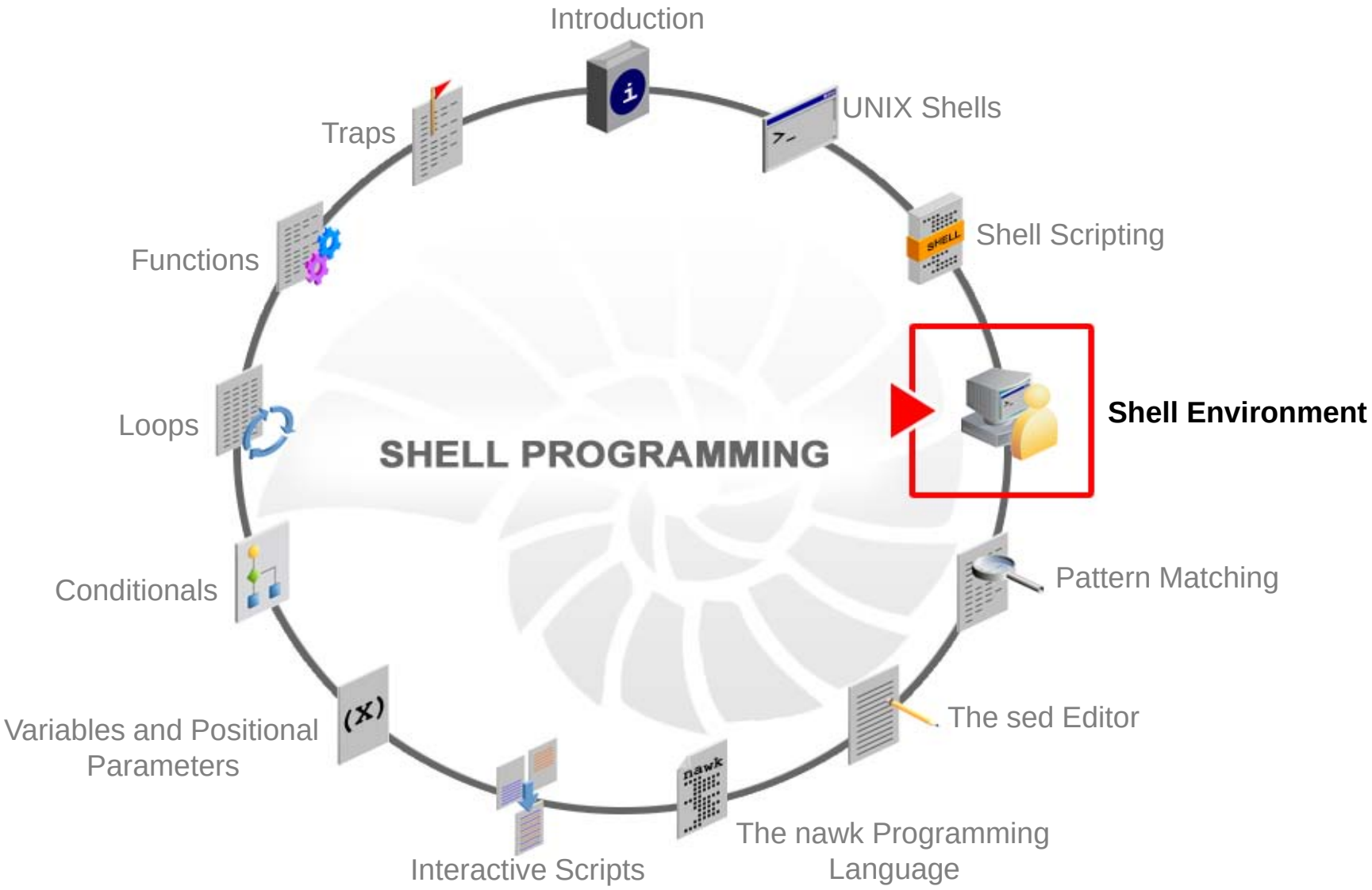
4

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.

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
```

Quiz

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

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

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
}
```

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

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.

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.

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

Quiz

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

a. True
b. False

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

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.

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.

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`

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
```

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
```

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
...
}
```

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
```

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

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 ))
```

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.		

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`

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.
```

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.

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

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