Shell Scripts-II

Session Objectives

- If .. fi construct
- Case .. esac construct
- For construct
- While construct
- Until construct
- Break and continue statements
- Set command
- Functions in shell script
- Signals

Programming language constructs

Shell offers standard programming structures like loops and branching, operations on variables, file creation and argument passing.

The if construct

The if construct is useful to perform a set of commands based on a condition being true. If the condition is not true, we can specify another set of commands to be executed.

The syntax is

```
if (condition)
then
commands
else
commands
fi
```

`fi` is used to indicate the end of the if construct.

Linux also provides the `if..elif` construct, the syntax is given below:

```
if (condition)
then
commands
elif (condition)
commands
else
commands
fi
```

An `if` construct can have multiple `elif` statements but can have only one `else construct` and one `fi` to terminate the construct.

If's companion – test

When If is used to evaluate expressions, the test statement is used as its control command. Test evaluates the condition placed on its right, and returns either a true or false exit status. This return value is used by if for taking decisions.

Test uses the following operators called relational operators to compare two integers within a shell script.

Relational operators

Operators	**Meaning**
`–eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`–lt`	less than

-ge	Greater than or equal to
-le	less than or equal to

Test does not display any output, but simply returns a value, which is assigned to $?. Take a look at the following code:

```
$ a=10
$ b=20
$ c=14
$ test $a –eq $b ; echo $?
1
$ test $a –lt $b ; echo $?
0
$ test $a –gt $c ; echo $?
1
```

Logical operators

A list of the logical operators that can be used in shell scripts is given in the table below:

Operator	**Significance**
!	Negates the following expression
-a	Indicates and operation
-o	Indicates or operation

Test can be used as a standalone feature and can also be used in the command line of the if condition.

Example

Write a shell script that takes two arguments first argument being the pattern, second argument the file in which the pattern has to be searched.

💻 HANDS-ON

```
# to check test
if test $# -ne 2
then
   echo “invalid number of arguments”
   exit 3
else
   if grep “$1” $2
   then
      echo “pattern found”
   else
      echo “pattern not found”
   fi
fi
```

In the above program there are 2 if constructs one nested within the other. Here test is used to check if the user has entered the right number of command line arguments. If this condition evaluates to true,the messsage invalid number of arguments is displayed and the

script is terminated. If the right number of arguments are specified, grep command is executed.

Example

Write a shell script to calculate the bonus to be given to the employees based on their basic salary.

💻HANDS-ON

```
# To calculate bonus
echo “Enter employee\’s basic salary”
read basic

if [ $basic -gt 5000 ]
then
hra=`expr $basic \* 4`
echo $hra

elif [ $basic –ge 4000 –a $basic –le 5000 ]
then
hra=`expr $basic \* 3`
echo $hra

else
hra=`expr $basic \* 2`
echo $hra

fi
```

The test keyword can also be replaced with [ ].

Example

💻HANDS-ON

```
Echo “what is your name”
Read name
If test $name = BILL
Then
Echo “We have met before”
Else
Echo “We have not met before”
Fi
```

So the command

test $name = BILL can be written as [$name = BILL]

Test on character strings

Test statement can be used in testing and comparing strings.

Test	Exit status
-n string	True if string is not null
-z string	True if string is null
s1= s2	True if string s1 = s2
s1 != s2	True if string s1 is not equal to s2

Test on file types

The test command has got various options to check for the file status

Test	Exit status
-f	to check for existence of the file and to check if it is an ordinary file
-d	to check for existence of the file and to check if it is a directory
-r	to check if file exists and if it is readable
-w	to check if file exists and if it is writable
-x	to check if file exists and if it is executable
-s	to check if file exists and if it is not empty

Example

Write a shell script that takes one argument and checks if it is a file or a directory and displays the contents.

💻HANDS-ON

```
ans=""
wish=""
if [ $# -lt 1 ]
then
     echo "invalid usage. Usage: $0 filename"
     exit
fi
if [ -f $1 ]
then
   echo "$1 is an ordinary file. Do you want to see the contents?"
   read ans
  if [ $ans = "Y" –o $ans = "y" ]
  then
   if [ -r $1 ]
    then
      cat $1
```

```
   else
     echo “$1 has no read permission”
   fi
  else
    break
  fi
elif [ -d $1 ]
then
  echo “$1 is a directory. Do you want to see the contents?”
  read wish
 if [ $wish = “Y” –o $wish = “y” ]
   then
     cd $1
     ls –l
 else
   break
 fi
else
   echo “$1 does not exist”
   exit
fi
```

The case..esac construct

The case..esac construct is used in shell scripts to perform a set of instructions depending on the value of a variable.

The syntax is

case value in
choice1) commands;;
choice2) commands;;
…
esac

Example

Write a shell script that displays a menu to the user and based on the user’s choice performs a set of operations:

💻HANDS-ON

```
# usage of the case..esac construct
echo “Menu”
echo “1. Date”
echo “2. Current directory”
echo “3. Users currently logged in”
echo “Please enter your choice”
read choice
```

```
case $choice in
1) date;;
2) pwd;;
3) who;;
*) echo “invalid choice”;;
esac
```

The *) will take over if the user fails to enter any one of the above choices.

The for construct

The for construct takes a list of values as input and executes the loop for every value in the loop.

The syntax is

```
for variable in <list of values>
do
commands
done
```

Example

Write a shell script to calculate the square of numbers from 1 to 5.

💻HANDS-ON

```
# to find square of numbers from 1 to 5
for k in 1 2 3 4 5
do
echo “ the number is $k”
echo “ the square of the number is `expr $k \* $k`”
done
```

There are variations in for loop. It can be used for listing from variables,wild-card expressions,positional parameters,command substitution.

List from variables

```
filestr=”file1 file2 file3”
for var in $filestr
do
  wc $var
done
```

The above program iterates through the values of the variable filestr.

List from wild-card expressions

```
for file in *
   do
     cat $file
   done
```

The above shell script takes filenames from the current directory as values and list the contents of the files.

List from positional parameters

```
for i in $*
   do
     cat $i
   done
```

The above shell script takes filenames from shell script arguments as values and list the contents of the files.

List from command substitution

```
for i in `cat file1`
   do
     echo $i
   done
```

The while loop

The while loop is used to execute commands repeatedly as long as the condition remains true.

The syntax is

```
while condition
do
commands
done
```

Example

Write a shell script to reverse a string given by the user

💻HANDS-ON

```
# To reverse the given string
echo “enter the string to be reversed”
read str
s=`expr length $str`
while [ $s –gt 0 ]
```

```
do
temp=$temp`echo $str |cut –c $s`
s=`expr $s – 1`
done
echo “reversed string is $temp”
```

The while true loop

The while true loop runs indefinitely unless an interrupt character is pressed or an exit statement is encountered.

Consider the following example,

💻HANDS-ON

```
# to illustrate while true loop
ans=””
while true
do
echo “do you wish to continue?”
read ans
if [ $ans = “Y”  -o $ans = “y” ]
then
echo “Welcome”
else
exit
fi
done
```

The until construct

The until loop is similar to the while loop except that it continues as long as the control command fails.
The syntax is

```
until condition
do
commands
done
```

Example

Write a shell script to guess a number between 1 and 30

💻**HANDS-ON**

```
# To guess a number between 1 and 30
num=24
count=0
echo “Guess the number that I am thinking of. The number is between
1 and 30”
read guess
until [ $num –eq $guess ]
do
if [ $guess –gt $num ]
then
echo “too high. Try again”
else
echo “too low. Try again”
fi

count=`expr $count + 1`
read guess
done
echo “You have taken $count chances to guess the number”
```

The until false loop

The until false loop is complementary to the while true loop. As long as the condition remains false, the execution of commands continues.

💻**HANDS-ON**

```
# illustrate usage of until false loop
until false
do
ps –f
sleep 5
echo “Linux at your service”
echo “press ctrl+c to exit”
done
```

The above example displays the current process information, wait for five seconds and again displays the process information. This loop continues until the user presses Ctrl+c to exit from the program.

The break and continue statements

The break statement helps in the termination of a loop. The example given below demonstrates the usage of the break statement

💻HANDS-ON

```
# Program to demonstrate break statement
while true
do
echo “enter your choice”
echo “enter q to quit”
echo “Menu”
echo “a.Today’s date”
echo “b.Users logged in”
read choice
case $choice in
a) date;;
b) who;;
q) break;;
*) echo “not a valid choice”
esac
done
```

The continue statement is used when we want to skip the remaining commands in the loop and start from the beginning of the loop again. The difference between the break and continue commands is that the break command breaks out of the loop but the continue command continues execution from the beginning. The example given below demonstrates the continue statement.

💻HANDS-ON

```
# Program to demonstrate continue statement
str=””
echo “do you want to give a value for str?”
read str
while [ $str = “y” –o $str = “Y” ]
do
echo “enter a name”
read name
echo $name >>names
wish=””
echo “do you want to continue?”
read wish
if [ $wish = “y” –o $wish = “Y” ]
then
continue
else
echo “the contents of the file names is : `cat names`”
exit
fi
done
```

The set command

The set command is used to find out the existing values of environment variables. The second use of set command is to assign values to the positional parameters. For example, set java oracle sets $1 to java and $2 to oracle.

The –v option

The set command when used with the –v option will echo each command inside the shell script before executing it. So this can be used for debugging the shell programs. It can also be invoked along with “sh filename”.

💻HANDS-ON

```
# To demonstrate usage of –v option
echo “Enter your name”
read name
if (who | grep $name > /dev/null)
then
echo “$name has logged in”
else
echo “$name has not logged in”
fi
```

```
$ sh -v exam
# To demonstrate usage of -v option
echo "Enter your name"
Enter your name
read name
meera
if (who | grep $name > /dev/null)
then
echo "$name has logged in"
else
echo "$name has not logged in"
fi
meera has logged in
```

The –x option

The –x option is similar to the –v option except that it shows the substitution that the shell makes inside the shell scripts. A + sign at the beginning of the line shows the output of the –x option.

So executing the previous example with the –x option produces the following output.

```
$ sh -x exam
+ echo 'Enter your name'
Enter your name
+ read name
meera
+ who
```

```
+ grep meera
+ echo 'meera has logged in'
meera has logged in
$_
```

Functions in shell scripts

The Bourne shell has functions. Functions are very useful in shell scripts. They allow more flexibility. A function has the form:

```
function ( )
{
command;
}
```

where the space after {, and the semicolon (;) are both required. The semicolon can be dispensed with if a <newline> precedes the }. Additional spaces and <newline>'s are allowed.

Consider the following function. The function merely echoes the name which the user has entered.

💻HANDS-ON

```
func2()
{
echo $name
}
echo "Enter the name"
read name
func2
```

Signals

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

Because signals can arrive at any time during the execution of a script, they add an extra level of complexity to shell scripts. Scripts must account for this fact and include extra code that can determine how to respond appropriately to a signal regardless of what the script was doing when the signal was received.

In this chapter we will look at the following topics:

- The different types of signals encountered in shell programming

- How to deliver signals using the kill command
- Handling signals
- How to use signals within your script

How Are Signal Represented?

Each type of event is represented by a separate signal. Each signal is only a small positive integer. The signals most commonly encountered in shell script programming are given in Table 19.1.

Important Signals for Shell Scripts

Name	Value	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Interrupt from keyboard
SIGQUIT	3	Quit from keyboard
SIGKILL	9	Kill signal
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Termination signal

In addition to the signals listed in the above table , you might occasionally see a reference to signal 0, which is more of a shell convention than a real signal. When a shell script exits either by using the exit command or by executing the last command in the script, the shell in which the script was running sends itself a signal 0 to indicate that it should terminate.

Getting a List of Signals

All the signals understood by your system are listed in the C language header file `signal.h`. The location of this file varies between UNIX flavors. Some common locations are

- Solaris and HPUX: /usr/include/sys/signal.h
- Linux: /usr/include/asm/signal.h

Some vendors provide a man page for this file which you can view with one of the following commands:

- In Linux: man 7 signal
- In Solaris: man -s 5 signal
- In HP-UX: man 5 signal

Another way that your system can understand a list of signals is to use the -l option of the kill command. For example on a Solaris system or Linux, the output is:

```
$ kill -l   -→This is not one but the alphabet 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	31. SIGXFSZ	32. SIGWAITING
33. SIGLWP	34. SIGFREEZE	35. SIGTHAW	36. SIGCANCEL
37. SIGLOST			

The actual list of signals varies between Solaris, HP-UX, and Linux.

Default Actions

Every signal, including those listed in the table above, has a *default action* associated with it. The default action for a signal is the action that a script or program performs when it receives a signal. Some of the possible default actions are

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called core containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

The default action for the signals that you should be concerned about is to terminate the process.

Delivering Signals

There are several methods of delivering signals to a program or script. One of the most common is for a user to type CONTROL-C or the INTERRUPT key while a script is executing. In this case a SIGINT is sent to the script and it terminates.

The other common method for delivering signals is to use the kill command as follows:

```
kill -signal pid
```

Here *signal* is either the number or name of the signal to deliver and *pid* is the process ID that the signal should be sent to.

TERM

By default the kill command sends a TERM or terminates a signal to the program running with the specified pid. A PID is the process ID given by UNIX to a program while it is executing. Thus the commands

```
kill pid
kill -s SIGTERM pid
```

are equivalent.

Now look at a few examples of using the kill command to deliver other signals.

HUP

The following command

```
$ kill -s SIGHUP 1001
```

sends the HUP or hang-up signal to the program that is running with process ID 1001. You can also use the numeric value of the signal as follows:

```
$ kill -1 1001
```

This command also sends the hang-up signal to the program that is running with process ID 1001. Although the default action for this signal calls for the process to terminate, many UNIX programs use the HUP signal as an indication that they should reinitialize themselves. For this reason, you should use a different signal if you are trying to terminate or kill a process.

QUIT and INT

If the default kill command cannot terminate a process, you can try to send the process either a QUIT or an INT (interrupt) signal as follows:

```
$ kill -s SIGQUIT 1001
```
or
```
$ kill -s SIGINT 1001
```

One of these signals should terminate a process, either by asking it to quit (the QUIT signal) or by asking it to interrupt its processing (the INT signal).

kill

Some programs and shell scripts have special functions called *signal handlers* that can ignore or discard these signals. To terminate such a program, use the kill signal:

```
$ kill -9 1001
```

Here you are sending the kill signal to the program running with process ID 1001. In this case you are using the numeric value of the signal, instead of the name of the signal. By convention, the numeric value of the kill signal, 9, is always used for it.

The kill signal has the special property that it cannot be caught, thus any process receiving this signal terminates immediately "with extreme prejudice." This means that a process cannot cleanly exit and might leave data it was using in a corrupted state. You should only use this signal to terminate a process when all the other signals fail to do so.

Dealing with Signals

A program or script can handle a signal in three different ways:

- Do nothing and let the default action occur. This is the simplest method for a script to deal with a signal.
- Ignore the signal and continue executing. This method is not the same as doing nothing, because ignoring a signal requires the script to have some code that explicitly ignores signals.
- Catch the signal and perform some signal-specific commands. In this method the script has a special routine that executes when a signal is received. This is the most complex and powerful method of dealing with signals.

The first method for dealing with a signal requires no additional code in your shell script. This is the default behavior for all shell scripts that do not explicitly handle signals. All the scripts that you have looked at so far handle signals using this method.
In this section you will look at the second and third methods of dealing with signals.

The trap Command

The trap command sets and unsets the actions taken when a signal is received. Its syntax is
`trap` *`name signals`*

Here *name* is a list of commands or the name of a shell function to execute when a signal in the list of specified *signals* is received. If *name* is not given, trap resets the action for the given signals to be the default action.

There are three common uses for trap in shell scripts:

- Clean up temporary files
- Always ignore signals
- Ignore signals only during critical operations

You will look at a fourth use, setting up a timer, later in this chapter.

Cleaning Up Temporary Files

Most shell scripts that create temporary files use a trap command similar to the following:

```
trap "rm -f $TMPF; exit 2" 1 2 3 15
```

Here you remove the file stored in $TMPF and then exit with a return code of 2 indicating that exit was abnormal, when either a HUP, INT, QUIT, or TERM signal is received. Usually when a script exits normally its exit code is 0. If anything abnormal happens, the exit code should be nonzero. Sometimes when more complicated clean up is required, a shell function can be used. In order to make the uu script, you would add something similar to the following at the beginning of the script:

```
CleanUp() {
    if [ -f "$OUTFILE" ]
       then
        printf "Cleaning Up... ";
        rm -f "$OUTFILE" 2> /dev/null ;
        echo "Done." ;
    fi
}
trap CleanUp 1 2 3 15
```

Here the function CleanUp is invoked whenever the script receives a HUP, INT, QUIT, or TERM signal. This function removes the output file of the script, if that file exists. By cleaning up when a signal is received, partially encoded files are not left around to confuse users.

NOTE
The main reason to use functions to handle signals is that it is nicer to have a shell function invoked when a signal is received rather than write in the appropriate code inline. Also, the commands that should be executed when a signal is received might be different depending on which point in the script the signal was received. In many cases it is difficult to capture that logic in a few commands, thus is it necessary to use a shell function as the signal handling routine.

In the previous examples, a single signal handler has been used for all signals. This is not required, and frequently different signals have different handlers. As an example, the following trap commands are completely valid:

```
trap CleanUp 2 15
trap Init 1
```

Here the script calls a clean up routine when an INT or TERM signal is received and calls its initialization routine when a SIGHUP is received. Declarations such as these are common in scripts that run as daemons.

Ignoring Signals

Sometimes there is no intelligent or easy way to clean up if a signal is received. In these cases, it is better to ignore signals than to deal with them. There are two methods of ignoring signals:

```
trap '' signals
trap : signals
```

The signals argument for the trap command is a list of signals to ignore. The only difference between these two forms is that the first form passes a special argument, '' or null, to the trap command and the second uses the : command. The form to use is largely based on programmer style because both forms produce the same result. If you simply wanted the the shell script to ignore all signals, instead of cleaning up when it received a signal, you could add the following to the beginning of the script:

```
trap '' 1 2 3 15
```

Ignoring Signals During Critical Operations

When this command is given in a script, the script ignores all signals until it exits. From a programmer's perspective, this might be a good idea, but from a user's perspective, it is not. A better idea is to have only the critical sections of a script ignore traps. This is achieved by unsetting the signal handler when a section of critical code has finished executing.

As an illustration, say you have a shell script with a shell function called DoImportantStuff(). This routine should not be interrupted. In order to ensure this, you can install the signal handler before the function is called and reset it after the call finishes:

```
trap '' 1 2 3 15
DoImportantStuff
trap 1 2 3 15
```

The second call to trap has only signal arguments. This causes trap to reset the handler for each of the signals to the default. By doing this, you enable the user to still terminate the script and ensure that critical operations are performed without interruption.

REVIEW QUESTIONS

1. The _____ is used in place of the test command.
2. The while loop executes as long as the condition is _____.
3. The _____ loop is used to perform the same set of operations on a list of values.
4. The _____statement is used when we want to skip the remaining commands in the loop and start from the beginning of the loop again.
5. Signals are _____
6. Trap command is used to _____
7. Trap : 1 2 3 means _____
8. To call a function when a signal is received, the command is _____