# Advanced Shell Scripting

Senthil kumar Kaliyamurthy

© 2009 Wipro Ltd – Internal & Restricted

# Module Outline


Lesson 1 Shell Scripting Overview

Lesson 2 Environment

Lesson 3 Flow Control and Loops

Lesson 4 Shell Variables and Parameters

Lesson 5 Shell Functions

Lesson 6 Filters

Lesson 7 SED and AWK

Lesson 8 Debugging



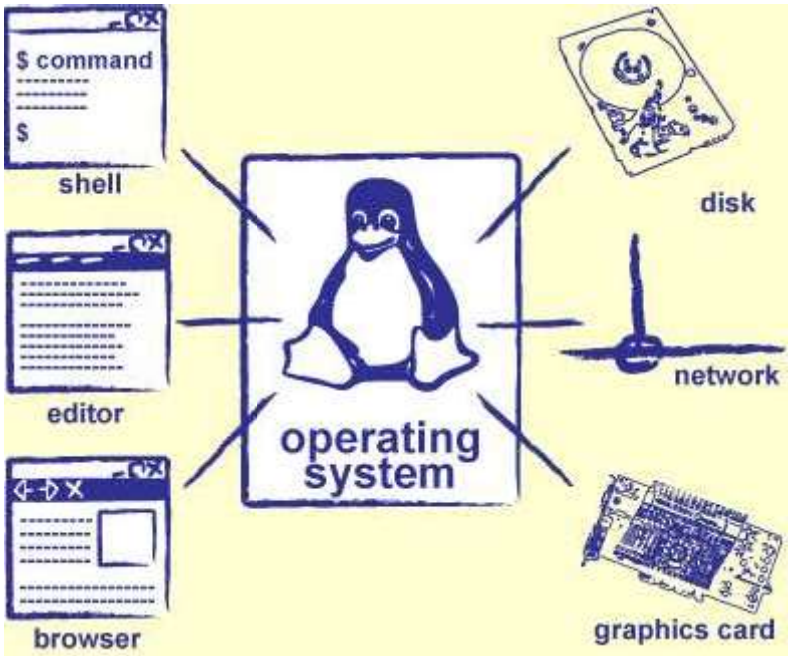

© 2009 Wipro Ltd – Internal & Restricted

# Lesson 1

Shell Scripting Overview


© 2009 Wipro Ltd – Internal & Restricted

# Shell Scripting Overview

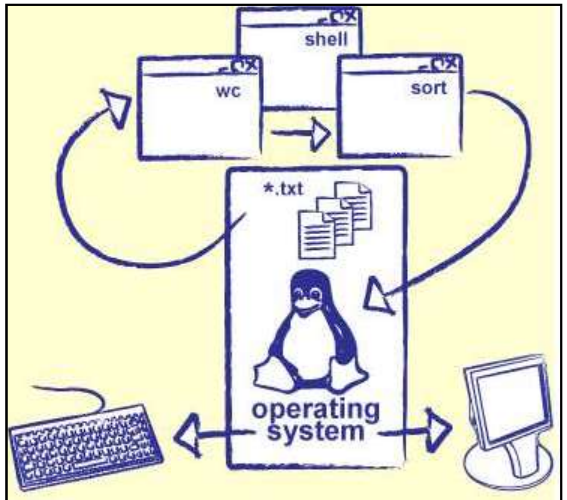



Shell is not the operating system…

Shell script is a set of UNIX commands along with flow control statements to yield desired output.


© 2009 Wipro Ltd – Internal & Restricted

# Shell Scripting Overview




Shell manage a user's interactions with the operating system by

**1)** Reading commands from keyboard
**2)** Executing those commands … or run another programs
**3)** Display the output

**Unix file Attributes**

This hyphen indicates that we're listing the permissions of a file, rather than a directory, or anything else.

-rwxr--r-x

OWNER'S PERMISSIONS
In this case, the owner can read, write and execute the file.

GROUP'S PERMISSIONS
In this case, the file's group can only read the file.

EVERYONE'S PERMISSIONS
In this case, everyone else can only read and execute the file.

**How "-rwxr-x--x" is equivalent to "751"**

r = 4, w = 2, x = 1

| | owner | group | other |
|---|---|---|---|
| | rwx | r-x | --x |
| read | 4 r | 4 r | 0 - |
| write | 2 w | 0 - | 0 - |
| execute | 1 x | 1 x | 1 x |
| Sum --> | 7 | 5 | 1 |


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 2

Environment


© 2009 Wipro Ltd – Internal & Restricted

# Environment

**The Login Process**

**1)** User enters his user name and then password.
**2)** Operating system authenticates the user id and password
**3)** A "shell" is created for you based on your entry in the "/etc/passwd" file
**4)** User is placed in HOME directory
**5)** Startup information is read from the file named "/etc/profile". This file is known as the system login file. When every user logs in, they read the information in this file.
**6)** Additional information is read from the file named ".profile" that is located in your "home" directory. This file is known as your personal login file.

**Specifying the Script's Interpreter**

#!/bin/bash

**The PATH Environment Variable**

$ echo $PATH
$ /usr/kerberos/bin:/home/poc1/OraHome1/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/rahul/bin

**Sub-shells**

Invoking a shell script by entering only the script's name on the command line creates a subshell in which the script runs. It is important to understand that the sub shell has a separate environment from the login/parent shell. Alternatively, prefacing the script name with a "." (dot) and a space causes the script to run within the context of the login session, as if you had typed each line of the script in at the command prompt.


© 2009 Wipro Ltd – Internal & Restricted

# Files Associated with Bash

For Login Shells:

- **/etc/profile**
  - Contains system wide settings of environmental variables and programs at the time of login.
- **~/.bash_profile**
  - Personalized settings to overrides the default setting. The rest two are ignored. Executed only once at startup. Not when switched from one shell to another and back to bash (unlike .bashrc)
- **~/.bash_login**
  - Executed if .bash_profile is not found and ignores .profile
- **~/.profile**
  - Executed if both the above are not present. The settings in this overrides the /etc/profile settings

For Non-Login Shell: (Any shell started without logging in)

- **~/.bashrc**
  - Executed immediately when an interactive shell starts
  - /etc/profile is only executed once at the startup of the session


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 3

Flow Control and Loops


© 2009 Wipro Ltd – Internal & Restricted

# Flow control & Loops

# **First form**

if *condition* ; then

*commands*

fi

# **Second form**

if *condition* ; then

*commands*

else

*commands*

fi

# **Third form**

if *condition* ; then

*commands*

elif *condition*

*commands*

fi

The CASE statement can be used to match (using shell wildcards such as * and ?) a given string against a number of possibilities.

case ... in

...) do something here;;

esac

for var in ....; do

....

done

while ...; do

....

Done

The **while-loop** will run while the expression that we test for is true. The keyword "break" can be used to leave the loop at any point in time. With the keyword "continue" the loop continues with the next iteration and skips the rest of the loop body

The **select** statement is a bash specific extension and is very good for interactive use. The user can select a choice from a list of different values:

select var in ... ; do

break

done

.... now $var can be used ....


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 4

## Shell Variables and Parameters


© 2009 Wipro Ltd – Internal & Restricted

# Shell Variables

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- ***Variable Names:***
- The name of a variable can contain only letters ( a to z or A to Z), numbers ( 0 to 9) or the underscore character ( _).

- By convention, Unix Shell variables would have their names in UPPERCASE.

- The following examples are valid variable names:

  _KUMAR

  TOKEN_A

  VAR_1

  VAR_2


© 2009 Wipro Ltd – Internal & Restricted

# Shell Variables

- Following are the examples of invalid variable names:

  2_VAR
  -VARIABLE
  VAR1-VAR2
  VAR_A!

- The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.
- **Defining Variables:**
- Variables are defined as follows::
- variable_name=variable_value
- For example: NAME="Sachin Kumar"
- Above example defines the variable NAME and assigns it the value "Sachin Kumar". Variables of this type are called scalar variables.
- A scalar variable can hold only one value at a time.


© 2009 Wipro Ltd – Internal & Restricted

# Shell Variables

The shell enables you to store any value you want in a variable.

For example:

VAR1="Sachin Kumar"

VAR2=100

- **Accessing Values:**

To access the value stored in a variable, prefix its name with the dollar sign ( $):

For example, following script would access the value of defined variable NAME and would print it on STDOUT:

```
#!/bin/sh
NAME="Sachin Kumar"
echo $NAME
```

This would produce following value: ➔ Sachin Kumar


© 2009 Wipro Ltd – Internal & Restricted

# Shell Variables

**Read-only Variables:**

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME:

```
#!/bin/sh
NAME="Sachin Kumar"
readonly NAME
NAME="Qadiri"
```

**This would produce following result:**

/bin/sh: NAME: This variable is read only.


© 2009 Wipro Ltd – Internal & Restricted

# Shell Variables

- **Variable Types:**
- When a shell is running, three main types of variables are present:

- **Local Variables:** A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.

- **Environment Variables:** An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

- **Shell Variables:** A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

- Previous tutorial warned about using certain nonalphanumeric characters in your variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

- For example, the $ character represents the process ID number, or PID, of the current shell:

- **[unix_user]$echo $$**

- Above command would write PID of the current shell:

- 29949

- The following table shows a number of special variables that you can use in your shell scripts:


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

- **Variable: Description**
- **$0**:The filename of the current script.
- **$n**:These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
- **$#**:The number of arguments supplied to a script.
- **$***:All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
- **$@**:All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
- **$?**:The exit status of the last command executed.
- **$$**:The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
- **$!**:The process number of the last background command.


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

- ***Command-Line Arguments:***

- The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

- Following script uses various special variables related to command line:

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Paramers : $#"
```


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

Here is a sample run for the above script:

```
[unix_user]$./test.sh Sachin Kumar
File Name : ./test.sh
First Parameter : Sachin
First Parameter : Kumar
Quoted Values: Sachin Kumar
Quoted Values: Sachin Kumar
Total Number of Paramers : 2
```

- **Special Parameters $* and $@:**
- There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "“.


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

- Both the parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.
- We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters:

```
#!/bin/sh
for TOKEN in $*
do
   echo $TOKEN
done
```

There is one sample run for this script:

```
[unix_user]$./test.sh Sachin
Kumar 10 Years Old

Sachin
Kumar
10
Years
Old
```


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

- **Exit Status:**
- The **$?** variable represents the exit status of the previous command.

- Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

- Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

- Following is the example of successful command:


© 2009 Wipro Ltd – Internal & Restricted

# Special Variables

```
[unix_user]$./test.sh Sachin Kumar
File Name : ./test.sh
First Parameter : Sachin
First Parameter : Kumar
Quoted Values: Sachin Kumar
Quoted Values: Sachin Kumar
Total Number of Paramers : 2
[unix_user]$echo $?
0
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Arrays

- A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.
- Shell supports a different type of variable called an array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.
- All the naming rules discussed for Shell Variables would be applicable while naming arrays.

- **Defining Array Values:**
- The difference between an array variable and a scalar variable can be explained as follows.
- Say that you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows:

```
NAME01="Sachin"
NAME02="Qadir"
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Arrays

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

**array_name[index]=value**

Here array_name is the name of the array, index is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands:

NAME[0]="Sachin"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"


© 2009 Wipro Ltd – Internal & Restricted

# Shell Arrays

If you are using ksh shell the here is the syntax of array initialization:

**set -A array_name value1 value2 ... valuen**

If you are using bash shell the here is the syntax of array initialization:

**array_name=(value1 ... valuen)**

- **Accessing Array Values:**
- After you have set any array variable, you access it as follows:
- ${array_name[index]} Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is the simplest example:


© 2009 Wipro Ltd – Internal & Restricted

# Shell Arrays

```
#!/bin/sh
NAME[0]="Sachin"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

This would produce following result:

```
[unix_user]$./test.sh
First Index: Sachin
Second Index: Qadir
```

You can access all the items in an array in one of the following ways:
${array_name[*]}
${array_name[@]}


© 2009 Wipro Ltd – Internal & Restricted

# Shell Arrays

Here array_name is the name of the array you are interested in.

Following is the simplest example:

```
#!/bin/sh
NAME[0]="Sachin"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

**This would produce following result:**

```
[unix_user]$./test.sh
First Method: Sachin Qadir Mahnaz Ayan Daisy
Second Method: Sachin Qadir Mahnaz Ayan Daisy
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Substitutions

- **What is Substitution?**
- The shell performs substitution when it encounters an expression that contains one or more special characters.
- ***Example:***
- Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line:

```
#!/bin/sh
a=10
echo -e "Value of a is $a \n"
```

**This would produce following result.** Here -e option enables interpretation of backslash escapes.

```
Value of a is 10
```

Here is the result without -e option:

```
Value of a is 10\n
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Substitutions

Here are following escape sequences which can be used in echo command:

| Escape | Description |
|---|---|
| \\ | backslash |
| \a | alert (BEL) |
| \b | backspace |
| \c | suppress trailing newline |
| \f | form feed |
| \n | new line |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |

You can use -E option to disable interpretation of backslash escapes (default).

You can use -n option to disable insertion of new line.


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Command Substitution

- Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

***Syntax:***

- The command substitution is performed when a command is given as:
- `` `command` ``
- When performing command substitution make sure that you are using the backquote, not the single quote character.

***Example:***

- Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution:


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Command Substitution

```
#!/bin/sh
DATE=`date`
echo "Date is $DATE”
USERS=`who | wc -l`
echo "Logged in user are $USERS”
UP=`date ; uptime`
echo "Uptime is $UP"
```

**This will produce following result:**

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13,
   0.07, 0.15
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Variable Substitution

- Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.
- Here is the following table for all the possible substitutions:

- **Form** **Description**

- **${var}** Substitue the value of var.
- **${var:-word}** If var is null or unset, word is substituted for var. The value of var does not change.
- **${var:=word}** If var is null or unset, var is set to the value of word.
- **${var:?message}** If var is null or unset, message is printed to standard error. This checks that variables are set correctly.
- **${var:+word}** If var is set, word is substituted for var. The value of var does not change.


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Variable Substitution

**Example:**

Following is the example to show various states of the above substitution:

```
#!/bin/sh
echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"
echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"
unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"
var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"
echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Variable Substitution

This would produce following result:

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

- There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

- There are following operators which we are going to discuss:

- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

- The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.
- Here is simple example to add two numbers:


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

- This would produce following result:

- `Total value : 4`

- There are following points to note down:
- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expresssion should be enclosed between ``, called inverted commas.


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

- **Arithmetic Operators:**
- There are following arithmatic operators supported by Bourne Shell.
- Assume variable a holds 10 and variable b holds 20 then:

| Operator | Description | Example |
|---|---|---|
| + | Addition - Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| * | Multiplication - Multiplies values on either side of the operator | `expr $a * $b` will give 200 |
| / | Division - Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = | Assignment - Assign right operand in left operand | a=$b would assign value of b into a |
| == | Equvality - Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != | Not Equvality - Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

- It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a == $b ] is correct where as [$a==$b] is incorrect.
- All the arithmetical calculations are done using long integers.

- **Relational Operators:**

- Bourne Shell supports following relational operators which are specific to numberic values. These operators would not work for string values unless their value is numerics.

- For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

- Assume variable a holds 10 and variable b holds 20 then:


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

| Operator | Description | Example |
|---|---|---|
| -eq | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is not true. |
| -ne | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a -ne $b ] is true. |
| -gt | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is not true. |
| -lt | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | [ $a -lt $b ] is true. |
| -ge | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | [ $a -ge $b ] is not true. |
| -le | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | [ $a -le $b ] is true. |


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a <= $b ] is correct where as [$a <= $b] is incorrect.

- **Boolean Operators:**

There are following boolean operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then:

| Operator | Description | Example |
|---|---|---|
| ! | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| -o | This is logical OR. If one of the operands is true then condition would be true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| -a | This is logical AND. If both the operands are true then condition would be true otherwise it would be false. | [ $a -lt 20 -a $b -gt 100 ] is false. |


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

- **String Operators:**
- There are following string operators supported by Bourne Shell.
- Assume variable a holds "abc" and variable b holds "efg" then:

| Operator | Description | Example |
|---|---|---|
| = | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a = $b ] is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a != $b ] is true. |
| -z | Checks if the given string operand size is zero. If it is zero length then it returns true. | [ -z $a ] is not true. |
| -n | Checks if the given string operand size is non-zero. If it is non-zero length then it returns true. | [ -z $a ] is not false. |
| str | Check if str is not the empty string. If it is empty then it returns false. | [ $a ] is not false. |


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

- **File Test Operators:**

There are following operators to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

| Operator | Description | Example |
|---|---|---|
| -b file | Checks if file is a block special file if yes then condition becomes true. | [ -b $file ] is false. |
| -c file | Checks if file is a character special file if yes then condition becomes true. | [ -b $file ] is false. |
| -d file | Check if file is a directory if yes then condition becomes true. | [ -d $file ] is not true. |
| -f file | Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true. | [ -f $file ] is true. |
| -s file | Check if file has size greater than 0 if yes then condition becomes true. | [ -s $file ] is true. |


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Basic Operators

| Operator | Description | Example |
|---|---|---|
| -g file | Checks if file has its set group ID (SGID) bit set if yes then condition becomes true. | [ -g $file ] is false. |
| -k file | Checks if file has its sticky bit set if yes then condition becomes true. | [ -k $file ] is false. |
| -p file | Checks if file is a named pipe if yes then condition becomes true. | [ -p $file ] is false. |
| -t file | Checks if file descriptor is open and associated with a terminal if yes then condition becomes true. | [ -t $file ] is false. |
| -u file | Checks if file has its set user id (SUID) bit set if yes then condition becomes true. | [ -u $file ] is false. |
| -r file | Checks if file is readable if yes then condition becomes true. | [ -r $file ] is true. |
| -e file | Check if file exists. Is true even if file is a directory but exists. | [ -e $file ] is true. |


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 5

Shell Functions


© 2009 Wipro Ltd – Internal & Restricted

# Unix - Shell Functions

- Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.
- Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.
- Shell functions are similar to subroutines, procedures, and functions in other programming languages.

- **Creating Functions:**
- To declare a function, simply use the following syntax:

```
function_name () {
   list of commands
}
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Functions

- The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.
- ***Example:***
- Following is the simple example of using function:

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World"
}
# Invoke your function
Hello
```

**When you would execute above script it would produce following result:**

```
[unix_user]$./test.sh
Hello World
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Functions

- **Pass Parameters to a Function:**
- You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.
- Following is an example where we pass two parameters *Sachin* and *Kumar* and then we capture and print these parameters in the function.

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World $1 $2"
}

# Invoke your function
Hello Sachin Kumar
```

This would produce following result:

```
[unix_user]$./test.sh
Hello World Sachin Kumar
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Functions

**Example:**

Following function returns a value 1:

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World $1 $2"
   return 10
}

# Invoke your function
Hello Sachin Kumar

# Capture value returned by last command
ret=$?
echo "Return value is $ret"
```

This would produce following result:

```
[unix_user]$./test.sh
Hello World Sachin Kumar
Return value is 10
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Functions

- **Nested Functions:**
- One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.
- Following simple example demonstrates a nesting of two functions:

```
#!/bin/sh
# Calling one function from another
number_one () {
   echo "This is the first function speaking..."
   number_two
}
number_two () {
   echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...
This is now the second function speaking...
```


© 2009 Wipro Ltd – Internal & Restricted

# Shell Functions

- **Function Call from Prompt:**
- You can put definitions for commonly used functions inside your *.profile* so that they'll be available whenever you log in and you can use them at command prompt.
- Alternatively, you can group the definitions in a file, say *test.sh*, and then execute the file in the current shell by typing:
- `[unix_user]$. test.sh`
- This has the effect of causing any functions defined inside test.sh to be read in and defined to the current shell as follows:

- `[unix_user]$ number_one`
- `This is the first function speaking...`
- `This is now the second function speaking...`

- `[unix_user]$`
- To remove the definition of a function from the shell, you use the unset command with the -f option. This is the same command you use to remove the definition of a variable to the shell.
- `[unix_user]$unset -f function_name`


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 6

Filters


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- **Filters**
- Shell scripts are often called on to manipulate and reformat the output from commands that they execute.
- Sometimes this task is as simple as displaying only part of the output by filtering out certain lines. In most instances, the processing required is much more sophisticated.
- several commands that are used heavily as text filters in shell scripts. These commands include
- **head**
- **tail**
- **grep**
- **sort**
- **uniq**
- **tr**
- I will also show you how to combine these commands to filter text in extremely powerful ways.


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- **The head Command**

- The basic syntax for the head command is
- head [-n *lines*] *files*
- Here *files* is the list of the files you want the head command to process. Without the -n *lines* option, the head command shows the first 10 lines of its standard input. This option shows the specified number of *lines* instead.
- Although this command is useful for viewing the tops of large README files, its real power happens in daily applications. Consider the following problem. I need to generate a list of the five most recently accessed
- To retrieve a list of the five most recently accessed files, I can pipe the output of the ls command into a head command:

```
ls -1ut /home/ranga/public_html | head -5
```

This produces the following list:

```
index.html
RCS
humor
misc
downloads
```


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- **The tail Command**
- The basic syntax for the tail command is similar to that of the head command:
- `tail [-n lines] files`
- Here *files* is the list of the files the tail command should process. Without the -n *lines* option, the tail
- command shows the last 10 lines of its standard input. With this option it shows the specified number of *lines* instead.

```
ls -1rt /var/spool/mail | tail -5
```

- On my system, I get this list:

```
senthil
joseph
kanwal
sridhar
vinodh
```


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- **The follow Option**
- An extremely useful feature of the tail command is the -f ( *f* as in *follow*) option:
- **`tail -f file`**
- Specifying the -f option enables you to examine the specified *file* while programs are writing to it.
- Often I have to look at the log files generated by programs that I am debugging, but I don't want to wait for the program to finish, so I can start the program and then use tail -f for the log file.
- Some Web administrators use a command such as the following to watch the HTTP requests made for their system:

- **`$ tail -f /var/log/httpd/access_log`**

- You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.
- To make a pipe, put a vertical bar (|) on the command line between two commands.
- When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a *filter*.


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- ***The grep Command:***
- The grep program searches a file or files for lines that have a certain pattern. The syntax is:
- [unix_user]$grep pattern file(s) The name "grep" derives from the ed (a UNIX line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."
- A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.
- The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

```
[unix_user]$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Filters

There are various options which you can use along with grep command:

| Option | Description |
|---|---|
| -v | Print all lines that do not match pattern. |
| -n | Print the matched line and its line number. |
| -l | Print only the names of files with matching lines (letter "l") |
| -c | Print only the count of matching lines. |
| -i | Match either upper- or lowercase. |

Next, let's use a regular expression that tells grep to find lines with "carol", followed by zero or more other characters abbreviated in a regular expression as ".*"), then followed by "Aug".

Here we are using -i option to have case insensitive search:

```
[unix_user]$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Filters

- The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting:

| Option | Description |
|---|---|
| -n | Sort numerically (example: 10 will sort after 2), ignore blanks and tabs. |
| -r | Reverse the order of sort. |
| -f | Sort upper- and lowercase together. |
| +x | Ignore first x fields when sorting. |

- More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by order of size.
- The following pipe consists of the commands **ls, grep,** and **sort**:

```
[unix_user]$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
[unix_user]$
```


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 7

Sed and awk


© 2009 Wipro Ltd – Internal & Restricted

# Sed

- Sed is a non-interactive line editor. It receives text input, whether from stdin or from a file, performs certain operations on specified lines of the input, one line at a time, then outputs the result to stdout or to a file. Within a shell script, sed is usually one of several tool components in a pipe.

- Sed determines which lines of its input that it will operate on from the *address range* passed to it. Specify this address range either by line number or by a pattern to match. For example, *3d* signals sed to delete line 3 of the input, and */windows/d* tells sed that you want every line of the input containing a match to "windows" deleted.

- Of all the operations in the sed toolkit, we will focus primarily on the three most commonly used ones. These are **p**rinting (to stdout), **d**eletion, and **s**ubstitution.


© 2009 Wipro Ltd – Internal & Restricted

# Sed

**Basic sed operators**

| Operator | Name | Effect |
|---|---|---|
| [address-range]/p | print | Print [specified address range] |
| [address-range]/d | delete | Delete [specified address range] |
| s/pattern1/pattern2/ | substitute | Substitute pattern2 for first instance of pattern1 in a line |
| [address-range]/s/pattern1/pattern2/ | substitute | Substitute pattern2 for first instance of pattern1 in a line, over *address-range* |
| [address-range]/y/pattern1/pattern2/ | transform | replace any character in pattern1 with the corresponding character in pattern2, over *address-range* (equivalent of **tr**) |
| g | global | Operate on *every* pattern match within each matched line of input |

Unless the g (*global*) operator is appended to a *substitute* command, the substitution operates only on the first instance of a pattern match within each line.


© 2009 Wipro Ltd – Internal & Restricted

# Sed

- From the command line and in a shell script, a sed operation may require quoting and certain options.

```
sed -e '/^$/d' $filename
# The -e option causes the next string to be interpreted as an editing
   instruction.
#  (If passing only a single instruction to "sed", the "-e" is optional.)
#  The "strong" quotes ('') protect the RE characters in the instruction
#+ from reinterpretation as special characters by the body of the script.
# (This reserves RE expansion of the instruction for sed.)
# Operates on the text contained in file $filename.
```

In certain cases, a **sed** editing command will not work with single quotes.

```
filename=file1.txt
pattern=BEGIN
  sed "/^$pattern/d" "$filename"  # Works as specified.
# sed '/^$pattern/d' "$filename"    has unexpected results.
#        In this instance, with strong quoting (' ... '),
#+      "$pattern" will not expand to "BEGIN".
```


© 2009 Wipro Ltd – Internal & Restricted

# Sed

- Sed uses the -e option to specify that the following string is an instruction or set of instructions. If there is only a single instruction contained in the string, then this option may be omitted.

```
sed -n '/xzy/p' $filename
# The -n option tells sed to print only those lines matching the
   pattern.
# Otherwise all input lines would print.
# The -e option not necessary here since there is only a single
   editing instruction.
```


© 2009 Wipro Ltd – Internal & Restricted

# Sed

**Examples of sed operators**

| Notation | Effect |
|---|---|
| 8d | Delete 8th line of input. |
| /^$/d | Delete all blank lines. |
| 1,/^$/d | Delete from beginning of input up to, and including first blank line. |
| /Jones/p | Print only lines containing "Jones" (with -n option). |
| s/Windows/Linux/ | Substitute "Linux" for first instance of "Windows" found in each input line. |
| s/BSOD/stability/g | Substitute "stability" for every instance of "BSOD" found in each input line. |
| s/ *$// | Delete all spaces at the end of every line. |
| s/00*/0/g | Compress all consecutive sequences of zeroes into a single zero. |
| /GUI/d | Delete all lines containing "GUI". |
| s/GUI//g | Delete all instances of "GUI", leaving the remainder of each line intact. |


© 2009 Wipro Ltd – Internal & Restricted

# awk

- The awk command is a programming language that enables you to search through files and modify records with these files based on patterns. The name *awk* comes from the last names of its creators Alfred *A*ho, Peter *W*einberger, and Brian *K*ernighan. It has been a part of UNIX since 1978, when it was added to UNIX Version 7.
- Currently three main versions are available:
- The original awk
- A newer version nawk
- The POSIX/GNU version gawk
- **Basic Syntax**
- The basic syntax of an awk command is
- awk '*script*' *files*
- Here *files* is a list of one or more files, and *script* is one or more commands of the form:
- /*pattern*/ { *actions* }
- Here *pattern* is a regular expression, and *actions* is one or more of the commands that are covered . If *pattern* is omitted, awk performs the specified *actions* for each input line.


© 2009 Wipro Ltd – Internal & Restricted

# awk

- Look at the simplest task in awk, displaying all the input lines from a file. In this case you use a modified version of the file fruit_prices.txt :

```
$ awk '{ print ; }' fruit_prices.txt
Fruit Price/lbs Quantity
Banana $0.89 100
Peach $0.79 65
Kiwi $1.50 22
Pineapple $1.29 35
Apple $0.99 78
```

- Here you use the awk command print to print each line of the input. When the print command is given without arguments, it prints the input line exactly as it was read.
- Notice that there is a semicolon ( ;) after the print command. This semicolon is required to let awk know that the command has concluded. Strictly speaking, some older versions of awk do not require this, but it is good practice to include it anyway.


© 2009 Wipro Ltd – Internal & Restricted

# awk

- **Field Editing**
- One of the nicest features available in awk is that it automatically divides input lines into *fields*. A field is a set of characters that are separated by one or more field separator characters. The default field separator characters are tab and space.
- When a line is read, awk places the fields that it has parsed into the variable 1 for the first field, 2 for the second field, and so on. To access a field, use the field operator, $. Thus, the first field is $1.
- **Caution -** The use of the $ in awk is slightly different than in the shell. The $ is required only when accessing the value of a field variable; it is not required when accessing the values of other variables.
- As an example of using fields, you can print only the name of a fruit and its quantity using the following awk command
- `$ awk '{ print $1 $3 ; }' fruit_prices.txt`
- Here you use awk to print two fields from every input line:
- The first field, which contains the fruit name
- The third field, which contains the quantity


© 2009 Wipro Ltd – Internal & Restricted

# awk

```
FruitQuantity
Banana100
Peach65
Kiwi22
Pineapple35
Apple78
```

- Notice that in the output there is no separation between the fields. This is the default behavior of the print command. To print a space between each field you need to use the , operator as follows:

- `$ awk '{ print $1 , $3 ; }' fruit_prices.txt`

```
Fruit Quantity
Banana 100
Peach 65
Kiwi 22
Pineapple 35
Apple 78
```


© 2009 Wipro Ltd – Internal & Restricted

# awk

- You can format the output by using the awk printf command instead of the print command as follows:

```
$ awk '{ printf "%-15s %s\n" , $1 , $3 ; }' fruit_prices.txt
Fruit Quantity
Banana 100
Peach 65
Kiwi 22
Pineapple 35
Apple 78
```

- **Taking Pattern-Specific Actions**

- Say that you want to highlight those fruits that cost more than a dollar by putting a * at the end of the line for those fruits. This means that you need to perform different actions depending on the pattern that was matched for the price.


© 2009 Wipro Ltd – Internal & Restricted

# awk

- Start with the following script:

```
#!/bin/sh
awk '
/ *\$[1-9][0-9]*\.[0-9][0-9] */ { print $1,$2,$3,"*"; }
/ *\$0\.[0-9][0-9] */ { print ; }
' fruit_prices.txt
```

- Here you have two patterns: The first one looks for fruit priced higher than a dollar, and the second one looks for fruit priced lower than a dollar. When a fruit priced higher than a dollar is encountered, the three fields are output with a * at the end of the line. For all other fruit, the line is printed exactly as it was read.
- The output looks like the following:

```
Banana $0.89 100
Peach $0.79 65
Kiwi $1.50 22 *
Pineapple $1.29 35 *
Apple $0.99 78
```


© 2009 Wipro Ltd – Internal & Restricted

# awk

- **Using STDIN as Input**
- Recall that the basic form of an awk command is
- awk '*script*' *files*
- If *files*, the list of files, is omitted, awk reads its input from STDIN. This enables us to use it to filter the output of other commands. For example, the command

  $ ls -l
- produces output formatted similar to the following:

```
total 64
-rw-r--r-- 1 ranga users 635 Nov 29 11:10 awkfruit.sh
-rw-r--r-- 1 ranga users 115 Nov 28 14:07 fruit_prices.txt
-rw-r--r-- 1 ranga users 80 Nov 27 13:53 fruit_prices.txt.7880
lrwxrwxrwx 1 ranga users 8 Nov 27 19:01 nash -> nash.txt
-rw-r--r-- 1 ranga users 62 Nov 27 16:06 nash.txt
-rw-r--r-- 1 ranga users 11 Nov 29 10:38 nums.txt
lrwxrwxrwx 1 ranga users 8 Nov 27 19:01 urls -> urls.txt
-rw-r--r-- 1 ranga users 180 Nov 27 12:34 urls.txt
```


© 2009 Wipro Ltd – Internal & Restricted

# awk

- You can use awk to manipulate the output of the ls -l command so that only the name of a file and its size are printed. Here, the name of the file is in field 9, and the size is in field 5. The following command prints the name of each file along with its size:

```
$ /bin/ls -l | awk '$1 !~ /total/ { printf "%-32s %s\n",$9,$5 ; }'
```

- The output looks like the following:

```
awkfruit.sh 635
fruit_prices.txt 115
fruit_prices.txt.7880 80
nash 8
nash.txt 62
nums.txt 11
urls 8
urls.txt 180
```


© 2009 Wipro Ltd – Internal & Restricted

# Lesson 8

Debugging


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- **Debugging**

- For larger shell scripts, especially the kind that change system configurations, trying to deduce the source of a problem from a script's output is insufficient.
- Often, by the time you get the output, it is too late--the script will have made incorrect modifications or changes. Another common scenario is adding features to a large script that someone else developed.
- In such cases you need to make sure your changes don't affect the rest of the script. Fortunately, the shell provides several built-in commands for enabling different modes of debugging support.
- In this chapter you learn how to enable debugging, and then you will look at how to use the following debugging modes:

- **Syntax checking**
- **Shell tracing**


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- **Enabling Debugging**
- By now, you are quite familiar with the basic syntax for executing a shell script:
- `$ script arg1 arg2 ... argN`
- Here *script* is the name of the script, and *arg1* through *argN* are the arguments to the script. A frequently used alternative method to execute a shell script is
- `$ /bin/sh script arg1 arg2 ... argN`
- Here you explicitly specify the shell, in this case /bin/sh, that you used to execute the script. The advantage of this method is that you can enable a debugging mode by supplying arguments to the shell.
- Using this method of script invocation, the basic syntax for enabling a debugging mode is
- `$ /bin/sh option script arg1 arg2 ... argN`
- A third way of enabling debugging is to change the first line of the *script*. Usually, the first line of a script is


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

```
#!/bin/sh
```

- UNIX uses this line to determine the shell you can use to execute a script. This indicates that the shell /bin/sh should be used to execute the script. Modify this line to specify a debugging option as follows:

  ```
  #!/bin/sh option
  ```

- Here *option* is one of the debugging options listed in Table
- Because the previously mentioned methods for enabling debugging modes take effect when a script is invoked, they are sometimes referred to as *invocation activated* debugging modes.

- **Debugging Options for Shell Scripts**
- **TABLE 1**
- **Option Description**
- -n Reads all commands, but does not execute them.
- -v Displays all lines as they are read.
- -x Displays all commands and their arguments as they execute. This option is often referred to as the *shell tracing* option.


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- **Using the set command**
- In one of the invocation activated debugging modes, the default behavior is for that debugging mode to take effect at the first line of your script and remain in effect until the last line. Sometimes you just need to debug a particular function or section of your script. In these cases, enabling debugging for the entire script is overkill.
- The debugging output is quite extensive, and it is often hard to sort out the real errors from the noise. Address this problem with the set command to enable the debugging modes.
- By using the set command, you can enable and disable debugging at any point in your shell script.
- **Enabling Debugging Using set**
- The basic syntax follows:
- `set option`
- Here *option* is one of the options given in Table.
- You can use the set command anywhere in your shell script, and many scripts use it to change the debugging flags as part of the normal execution of the script.
- Because these debugging modes are activated only when the shell script programmer uses the set command, they are sometimes referred to as "programmer activated" modes.


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- Consider the following excerpt from a shell script

```
#!/bin/sh

set -x
if [ -z "$1" ] ; then
echo "ERROR: Insufficient Args."
exit 1
fi
```

-
- Here the shell programmer is requesting that shell tracing (the -x option) be activated with the command from line 2:
- set -x
- Because this command occurs before the if statement (lines 3 through 6), shell tracing will be active while the if statement executes.
- Unless it is explicitly disabled later in the script, shell tracing remains in effect until the script exits.


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- **Disabling Debugging Using set**
- In addition to enabling debugging modes, you can use the set command to disable debugging modes as follows: **set +*option***
- Here *option* is the letter corresponding to one of the options given in Table 1. For example, the command: **$ set +x**
- disables the shell tracing debugging mode.
- All the debugging modes that are enabled for a script can be deactivated using the following command:
- $ set -
- **Enabling Debugging for a Single Function**
- One of the most common uses of the set command is to enable a particular debugging mode before a function executes and then disable debugging when the function finishes.
- For example, if you have a function called BuggyFunction() and you only want to enable the shell tracing debugging mode while that function executes, use the following command:
- **set -x ; BuggyFunction; set +x ;**
- Here the debugging mode is enabled just before the function is called and is disabled when the function completes. This method is favored over explicitly using the set command inside a function to enable debugging because it enables the implementation of the function to remain unchanged.


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

**Syntax Checking**

- When dealing with any shell script, check the syntax of the script before trying to execute it. This enables you to fix most problems.

- To enable syntax checking, use the -n option as follows:

- /bin/sh -n *script arg1 arg2 ... argN*

- Here *script* is the name of a script and *arg1* through *argN* are the arguments for that script. This command generates output only if errors occur in the specified *script*.
- Check the syntax of the following script

```
#!/bin/sh

 YN=y
 if [ $YN = "yes" ]
 echo "yes"
 fi
```

- Can you spot the error?


© 2009 Wipro Ltd – Internal & Restricted

# Debugging

- If this script is stored in the file buggy1.sh, check its syntax as follows:
- $ /bin/sh -n ./buggy1.sh
- The output looks like the following:

- ./buggy1.sh: syntax error at line 7: 'fi' unexpected

- This tells you that when the shell tried to read line 7, it found that the fi statement on line 6 was unexpected. By now you have probably figured out that the reason the shell was trying read line 7 is that the if statement on line 4
- if [ $YN = "y" ]
- is not terminated with a then statement. This line should read as follows:
- if [ $YN = "y" ] ; then
- Making this change, you find that the syntax of the script is okay because the command
- $ /bin/sh -n buggy1.sh
- produces no output.


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Shell Tracing

- **Shell Tracing**
- There are many instances when syntax checking gives your script a clean bill of health, but bugs are still lurking in your script. Running syntax checking on a shell script is similar to running a spelling checker on a text document--it might find most of the misspellings, but it can't fix problems like spelling *read* as *red*.
- For text documents, you need to proofread them in order to find and fix all misspellings. To find and fix these types of problems in shell scripts, you need to use shell tracing.
- In shell tracing mode the shell prints each command in the exact form in which it executes. For this reason, you sometimes see the shell tracing mode referred to as the *execution tracing mode*.
- The shell tracing or execution tracing mode is enabled by using the -x option ( *x* as in *execution*). For a complete script, it is enabled as follows:
- $ /bin/sh -x *script arg1 arg2 ... argN*
- As was mentioned before, it can also be enabled using the set command:
- set -x
- To get an idea of what the output of shell tracing looks like, try the following command:
- $ set -x ; ls *.sh ; set +x
- The output is similar to the following:
- + ls buggy.sh buggy1.sh buggy2.sh buggy3.sh buggy4.sh
- buggy.sh buggy1.sh buggy2.sh buggy3.sh buggy4.sh
- + set +x


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Shell Tracing

- In the output, the lines preceded by the plus ( +) character are the commands that the shell executes. The other lines are the output from those commands.

- As you can see from the output, the shell prints the exact ls command it executes. This is extremely useful in debugging because it enables you to determine whether all the substitutions were performed correctly.

- **Finding Syntax Bugs Using Shell Tracing**

- example script: buggy3.sh.

```
#!/bin/sh
Failed() {
if [ $1 -ne 0 ] ; then
echo "Failed. Exiting." ; exit 1 ;
fi
echo "Done."
}
YesNo() {
echo "$1 (y/n)? \c"
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Shell Tracing

```
read RESPONSE
case $RESPONSE in
[yY]|[Yy][Ee][Ss]) RESPONSE=y ;;
[nN]|[Nn][Oo]) RESPONSE=n ;;
esac
}
YesNo "Make backup"
if [ $RESPONSE = "y" ] ; then
echo "Deleting old backups, please wait... \c"
rm -fr backup > /dev/null 2>&1
Failed $?
echo "Making new backups, please wait... \c"
cp -r docs backup
Failed
fi
```

- There are at least three syntax bugs in this script and at least one logical oversight. See if you can find them.
- Assuming that the script is called buggy3.sh, first check its syntax as follows:
- $ /bin/sh -n ./buggy3.sh


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Shell Tracing

- Because there is no output, execute it:
- $ /bin/sh ./buggy3.sh
- The script first prompts you as follows:
- Make backup (y/n)?
- Answering y to this prompt produces output similar to the following:
- Deleting old backups, please wait... Done.
- Making new backups, please wait... buggy3.sh: test: argument expected
- On Linux systems, the output might vary slightly. In any case, an error message is generated. Because this doesn't state which line of the script the error occurs on, you need to track it down manually. From the output you know that the old backup was deleted successfully; therefore, the error is probably in the following part of the script:
- echo "Making new backups, please wait... \c"
- cp -r docs backup
- Failed
- Enable shell tracing for this section as follows:
- set -x
- echo "Making new backups, please wait... \c"
- cp -r docs backup
- Failed
- set +x


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Using Debugging Hooks

- To activate debugging, use the following:
- **Debug on**
- To deactivate debugging, use either of the following:
- **Debug**
- **Debug off**
- Actually, any argument passed to this function other than on or ON deactivates debugging.
- As an example of using this function, modify the functions in the script buggy3.sh to have debugging

```
#!/bin/sh
Debug() {
if [ "$DEBUG" = "true" ] ; then
if [ "$1" = "on" -o "$1" = "ON" ]
; then
set -x
else
set +x
fi
fi
}
```

```
Failed() {
Debug on
if [ "$1" -ne 0 ] ; then
echo "Failed. Exiting." ; exit 1 ;
fi
echo "Done."
Debug off
}
```


© 2009 Wipro Ltd – Internal & Restricted

# Unix – Using Debugging Hooks

```
YesNo() {
Debug on
echo "$1 (y/n)? \c"
read RESPONSE
case "$RESPONSE" in
[yY]|[Yy][Ee][Ss]) RESPONSE=y ;;
*) RESPONSE=n ;;
esac
Debug off
}
YesNo "Make backup"
if [ "$RESPONSE" = "y" ] ; then
echo "Deleting old backups,
please wait... \c"
rm -r backup > /dev/null 2>&1
Failed $?
echo "Making new backups, please
wait... \c"
cp -r docs backup
Failed $?
fi
```

The output will be normal if the script executes in either of the following methods:

```
$ /bin/sh ./buggy3.sh
$ ./buggy3.sh
```

```
The output includes shell tracing
if the same script executes in
either of the following methods:

$ DEBUG=true /bin/sh ./buggy3.sh
$ DEBUG=true ./buggy3.sh
```


© 2009 Wipro Ltd – Internal & Restricted

# References

**Publications**

Unix Shell Programming - Yashavant Kanetkar

Teach Yourself Shell Programming in 24 Hours - Sams

**Courses**

<List additional courses in your curriculum which learners can take>

**URL's**

For further reading, you can visit

**Training Programs**

<List additional training programs related to the curriculum which learners can attend>


© 2009 Wipro Ltd – Internal & Restricted

# Quiz


© 2009 Wipro Ltd – Internal & Restricted

# Question 1-Shell scripting overview

1. Startup information is read from the file named called?
   a) /etc/profile
   b) /etc/passwd
   c) ./profile
   d) Both a and b


© 2009 Wipro Ltd – Internal & Restricted

# Question-2 Environment

2. What is the shell variable you used to find PATH in shell?
   a) $PATH
   b) $HOME
   c) $OS
   d) $SHELL


© 2009 Wipro Ltd – Internal & Restricted

# Question-3 Flow control & Loops

3. What is the command used to end case statement?

a) esac

b) end case

c) ecase

d) Both a and b


© 2009 Wipro Ltd – Internal & Restricted

# Question-4 Shell Variables

4. What is the shell variable used to find the exit status?

a) $$

b) $?

c) $@

d) $0


© 2009 Wipro Ltd – Internal & Restricted

# Question-5 Shell functions

5. To remove the definition of a function from the shell, you use the unset command with below format?

a) unset

b) unset -f

c) unset -v

d) Unset -fv


© 2009 Wipro Ltd – Internal & Restricted

# Question-6 Filters

6. Choose the correct options with grep command to print all lines that do not match pattern?

a) -v

b) -n

c) -c

d) -i


© 2009 Wipro Ltd – Internal & Restricted

# Question-7 Sed and awk

5. What is the sed operator used to substitute during replacement action?

a) e

b) s

c) d

d) p


© 2009 Wipro Ltd – Internal & Restricted

# Question-8 Debugging

8. What is the command used for to deactivate the debugging in shell script?
   a) Debug
   b) Debug on
   c) Debug off
   d) Both a and c


© 2009 Wipro Ltd – Internal & Restricted

# References


© 2009 Wipro Ltd – Internal & Restricted

# References

**Publications**

Unix Shell Programming - Yashavant Kanetkar

Teach Yourself Shell Programming in 24 Hours - Sams

**Courses**

<List additional courses in your curriculum which learners can take>

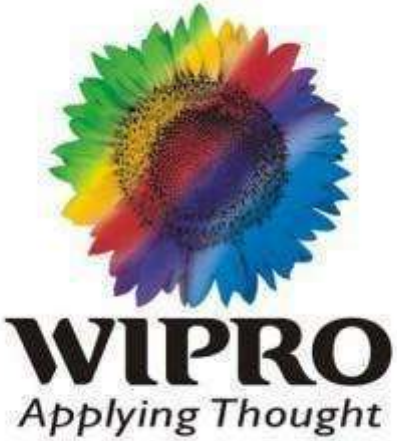
**URL's**

<List additional URLs related to the curriculum which learners can attend>

**Training Programs**

<List additional training programs related to the curriculum which learners can attend>


© 2009 Wipro Ltd – Internal & Restricted




# Thank You

© 2009 Wipro Ltd – Internal & Restricted