UNIX

Lesson 08: Shell Programming

Lesson Objectives

- At the end of the session you will be able to understand:
  - Shell variable
  - Environment variables
  - Shell script commands
  - Command substitution
  - Command line argument
  - Conditional statements
  - Calling PL/SQL block(procedure, function) from UNIX
  - Iterative statements

8.1: Shell Variables
Introduction

- System Variables
  - Set during:
    - Boot
    - Login
- .profile:
  - Script executed at login.
  - Alters operating environment of a user.
- $set
  - Displays a list of system variables.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

System Variables

There are several variables set by the system - some during booting and some after logging in. These are called the system variables, and they determine the environment one is working in. The user can also alter their values. The set statement can be used to display list of system variables.

```
$ set
HOME=/usr1/deshpavn
HUSHLOGIN=FALSE
HZ=100
IFS=
LOGNAME=deshpavn
MAIL=/usr/spool/mail/deshpavn
MAILCHECK=600
MF_ADM=adm.cat@Unix
MSG_MAIL=1
MS_PROFILE=1
OPTIND=1
PATH=/bin:/usr/bin:/usr1/deshpavn/bin:.
PS1=$
PS2=>
SHELL=/bin/sh
TERM=ansi
TZ=IST-5:30
```

8.2: Environmental Variables
Standard shell variables

- Shell Variables

  - PATH   : Contains the search path string.
  - HOME   : Specifies full path names for user login
                 directory.
  - TERM   : Holds terminal specification information
  - LOGNAME   : Holds the user login name.
  - PS1   : Stores the primary prompt string.
  - PS2   : Specifies the secondary prompt string.

Output of set command
Significance of some of these variables is explained below:
PATH Variable: Determines the list of directories (in order of precedence) that need
to be scanned while you look for an executable command.
Path can be modified as:
   $ PATH=$PATH:/usr/user1/progs
This causes the /usr/user1/progs path to get added to the existing PATH list.
HOME Variable: This controls the login or Home directory for the user.
IFS Variable: It contains a string of characters that can be used as separators on
command line.
PS1 and PS2 Variables: These determine the primary and secondary prompt.

8.2: Environmental Variables
Scripts executed automatically

- .profile script
  - shell script that gets executed by the shell when the user logs on
  - Used by Bourne shell
- .cshrc ,.login
  - Used by C Shell users
  - *.login* and is read when the user logs in.
  - *.cshrc* and is read whenever a new C shell is created
- .logout script
  - .logout file can also be created for commands to be executed when you log out.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

.profile script
The *.profile* script is a shell script that gets executed by the shell when the user logs on. It contains settings for the operating environment of the user, and it remains in effect throughout the login session. Using this file, it is possible to customize operating environment.

.cshrc ,.login and .logout script
For the Bourne shell, the system reads the *.profile* file and executes the commands found there. C Shell users, however, have two files to read and execute. One is called *.login* and is read when the user logs in. The second is called *.cshrc* and is read whenever a new C shell is created, including the login shell. A .logout file can also be created for commands to be executed when you log out.

8.3: Shell script Commands
Example

- Simple Shell Script: Accept Name & Display Message
   hello.sh

   ```
   echo "Good Morning!"
            echo "Enter your name?"
            read name
            echo "HELLO $name How are you?
   ```

- To execute the shell script
   $sh hello.sh
- To debug the shell script use –x option
   $sh –x hello.sh

In above program, the *read* command accepts input from the user and stores it in name variable.

To display the variable value, you need to precede the variable name with a $ sign:

   echo "HELLO **$name** How are you?

8.4: Arithmetic Operations
Details

> echo "Enter first Number"
> read no1
> echo "Enter second Number"
> read no2
> res=`expr $no1 + $no2`
> echo "The result is $res"

- In the above example, instead of *expr* we can use *let.*
  - Syntax:
    - let *expressions*   or   ((*expressions*))
  - In above script res=`expr $no1 + $no2` can be replaced by
                 let res=no1+n02

The above program accepts two numbers and displays their sum as a result. Instead of the *expr* command, we can use the *let* command.

Example:
Add *one* to variable i. Using expr statement:
- i=`expr $i + 1`

Add *one* to variable i. Using let statement:
- let i=i+1   If no spaces in expression
- let "i = i + 1"   enclose expression in "… " if expression includes spaces
- ((i = i + 1))

Expr is generally used but let is more user-friendly. It is used in Bash and Korn shell

8.5: Command Substitution
Details

- Command is enclosed in backquotes (`).
- Shell executes the command first.
  - Enclosed command text is replaced by the command output.
- Display output of the date command using echo:

  > $echo The date today is `date`
  >   The date today is Fri 27 00:12:55 EST 1990

- Issue echo and date commands sequentially:

  > $echo The date today is; date

$echo The date today is `date`
In this command date is a command which is enclosed in backquotes and hence will get replaced by its output and then echo command will display message

$echo The date today is; date
In above command echo and date commands are separated by ; hence will get executed sequentially.

8.5: Command Substitution

Example

- Following instructions print pwd as a string:

var=pwd
        echo $var
            Output: pwd

- Following instructions execute PWD shell command and display the present working directory:

var=`pwd`
       echo $var
        Output: /usr/deshpavan

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

In the first example pwd is a string which is assigned to var variable. Hence o/p of echo $var will be pwd

But in second example 'pwd' string is assigned to var variable
Hence echo $var command will is
   echo`pwd'
Since pwd is enclosed in backquotes it will get replaced by present working directory. echo will display name of current working directory.

8.6: Command Line Arguments
Details

- Specify arguments along with the name of the shell program on the command line called as command line argument.
- Arguments are assigned to special variables $1, $2 etc called as positional parameters.
- *special parameters*
  - $0 – Gives the name of the executed command
  - $* - Gives the complete set of positional parameters
  - $# - Gives the number of arguments
  - $$ - Gives the PID of the current shell
  - $! – Gives the PID of the last background job
  - $? – Gives the exit status of the last command
  - $@ - Similar to $*, but generally used with strings in looping constructs

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as *command line arguments*.
Parameters Related to Command Line Arguments
When you specify argument along with the name of the shell procedure, they are assigned into parameters $1, $2 etc. They are called as positional parameters.
There are also some other *special parameters* you can use. Some of them are:
- $0 – Gives the name of the executed command
- $* - Gives the complete set of positional parameters
- $# - Gives the number of arguments
- $$ - Gives the PID of the current shell
- $! – Gives the PID of the last background job
- $? – Gives the exit status of the last command
- $@ - Similar to $*, but generally used with strings in looping constructs

8.6: Command Line Arguments

Details

- Arguments are assigned to special variables (positional parameters).
  - $1 - First parameter , $2 - Second parameter,….
  - Example:

```
echo Program: $0
        echo Number of arguments are $#
        echo arguments are $*
grep "$1" $2
echo "\n End of Script"
```

  - Run script:

```
$ scr1.sh "Unix" books.lst          --$1 is UNIX  , $2 –books.lst
```

In above example
$ scr1.sh "Unix" books.lst   - The output only has lines with UNIX as substring from book.lst file .
Program: scr1.sh
Number of arguments are 2.
Arguments are Unix books.lst.
1001|Learning Unix |Computers |01/01/1998| 575
1004|Unix Device Drivers |Computers |09/08/1995| 650
1007|Unix Shell Programming |Computers |03/02/1993| 536
 End of Script.

8.7: Conditional Execution
Details

- Logical Operators && and ||:
  - **&&** operator delimits two commands. Second command is executed only if the first *succeeds*.
  - **||** operator delimits two commands. Second command is executed only if the first *fails*.
  - Example:

> $grep `director` emp.lst && echo "pattern found"
> $grep `manager` emp.lst || echo "pattern not found"

Conditional Execution using && and ||
The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command. In case of &&, the second command executes only if the first has succeeded. Similarly, || will ensure that the second command is executed only if the first has failed.

The following command displays "Found!" only if the XML pattern is found in the *books.lst* file at least once.

$ grep "XML" books.lst && echo "Found!"
1003|XML Unleashed |Computers |20/02/2000| 398
1006|XML Applications |Fiction |09/08/2000| 630
Found!

The following command displays "Not Found …". If *grep* does not find the "WAP" pattern in the *books.lst* file.
$ grep "WAP" books.lst || echo "Not Found..."
"Not Found..."

8.8: if Statement Format
Details

- Syntax

(i) if <condition is true>

   then

    <execute commands>

   else

    <execute commands>

fi

(ii) if <condition is true>

   then

    <execute commands>

fi

    – Example

```
if grep "^$1" /etc/passwd 2>/dev/null
   then
           echo "pattern found"
   else
           echo "pattern not found"
fi
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

In UNIX **/dev/null** or **the null device** is a special file that discards all data written to it.

The null device is typically used to dispose the unwanted output stream of a process.

In given example, if *grep* returns any error and you wish to discard error messages, use /dev/null device.

8.9: if Statement Format
if Statement

- Syntax:

  (iii) if <condition is true>

      then

        <execute commands>

      elif <condition is true>

      then

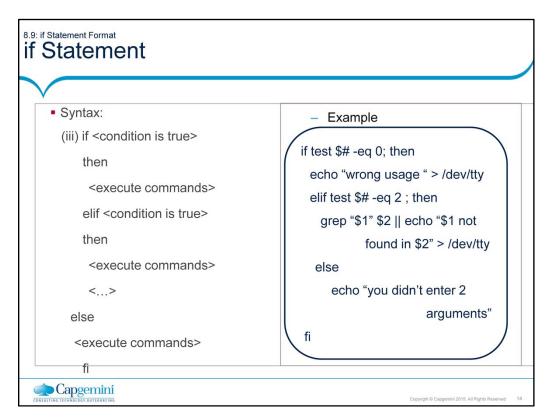   <execute commands>

        <...>

    else

     <execute commands>

     fi

— Example

```
if test $# -eq 0; then
    echo "wrong usage " > /dev/tty
    elif test $# -eq 2 ; then
        grep "$1" $2 || echo "$1 not
                found in $2" > /dev/tty
    else
        echo "you didn't enter 2
                        arguments"
fi
```

In the example, test command is use to specify condition
The shell scripts checks for *two* command line arguments. If the number of arguments is *zero*, then the output is:
Wrong Usage

If it is *two*, then the first argument is used as a pattern and the second one is used as the file name to search in the *grep* command.

If the pattern is found, then the output of the *grep* command is displayed. Otherwise, the output of the echo command is displayed.

If the number of arguments are not *two,* then the output is as follows:
"you didn't enter 2 arguments".

8.9: test Statement
Relational Operator for numbers

- Specify condition either using *test* or *[condition]*
  - Example: test $1 –eq $2   same as [$1 –eq $2]
- Relational Operator for Numbers:
  - eq: Equal to
  - ne: Not equal to
  - gt: Greater than
  - gc: Greater than or equal to
  - lt: Less than
  - lc: Less than or equal to

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.9: test Statement

Relational Operator for strings and logical operators

- String operators used by *test:*
  - -n str   True, if str not a null string
  - -z str   True, if str is a null string
  - S1 = S2   True, if S1 = S2
  - S1 != S2   True, if S1 $\neq$ S2
  - str   True, if str is assigned and not null

- Logical Operators
  - -a   .AND.
  - -o   .OR.
  - !   Not

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

9.9: test Statement
File related operators

- File related operators used by test command
  - -f <file>   True, if file exists and it is regular file
  - -d<file>   True, if file exist and it is directory file
  - -r <file>   True, if file exist and it is readable file
  - -w <file>   True, if file exist and it is writable file
  - -x <file>   True, if file exist and it is executable file
  - -s <file>   True, if file exist and it's size > 0
  - -e <file>   True, if file exist

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.9: test Statement
Example

- Check whether user has entered a filename or not:
  - Example:

```
echo "Enter File Name:\c "
read fn
if [ -z "$fn" ]
then
    echo "You have not entered file name"
fi
```

In the given example **–z** checks whether **$fn** is empty or not. If users do not enter the file name, then the output is as follows:

"You have not entered file name".

8.9: test Statement
Example

- Example:

> if test $x -eq $y
>
> ≡ if [$x -eq $y]

- Example:

> If [! -f fname]
>
> then
>
> echo "file does not exists"
>
> fi

if test $x -eq $y
≡ if [$x -eq $y]

In above command both the conditions are the same. You can use the "["
bracket to check the condition in place of the *test* command.

test $x –eq $y returns true if the values of variables x and y are equal. You can
write the same condition as [$x –eq $y]. Here, instead of *test* command we use
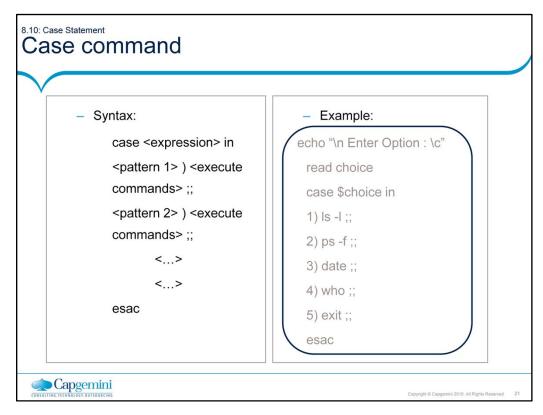"[" (square bracket).

If [! -f fname]

You can also write this condition as:
 test !-f fname

8.9: test Statement
Example

```
                    echo "Enter the source file name : \c"

                    read source

                    #check for the existence of the source file

                    if test –s "$source"   #file exists & size is > 0

                    then

                        if test ! –r  "$source"

                        then

                                echo "Source file is not readable"

                                exit

                        fi

                    else

                        echo "Source file not present"

                        exit

                    fi
```

The above example checks whether a given source file exists and displays appropriate messages.

8.10: Case Statement
Case command

Syntax:	Example:
– Syntax: case <expression> in <pattern 1>) <execute commands> ;; <pattern 2>) <execute commands> ;; <...> <...> esac	– Example: echo "\n Enter Option : \c" read choice case $choice in 1) ls -l ;; 2) ps -f ;; 3) date ;; 4) who ;; 5) exit ;; esac

In a *case* statement you can also use commands enclosed in *backquotes*. The
given example executes command `**date | cut -d " " –f1**` which returns only
the day part. The output is used to execute the appropriate case.
Example:

```
case `date | cut -d " " –f1`  in
Mon ) <commands> ;;
Tue ) <commands> ;;
                    :
esac
```

Example:
#display the options to the user
```
            echo "1. Date and time          2. Directory listing"
            echo "3. Users information       4. Current directory"
            echo "Enter choice (1,2,3,4) :\c"
read choice
case $choice in
1)      date;;
2)      ls –l;;
3)      who;;
4)      pwd;;
*) echo wrong choice;;
esac
#end of script
```

8.10: Case Statement
Example

```
echo "do you wish to continue?"
read ans
  Case "$ans" in
  [yY] [eE] [sS]) ;;
          [nN] [oO]) exit ;;
                      *) "invalid option" ;;
esac
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

In the above example, the first case matches with "yes" or "YES". Similarly, the second case matches with "no" or "NO".

8.11: While loop Statement
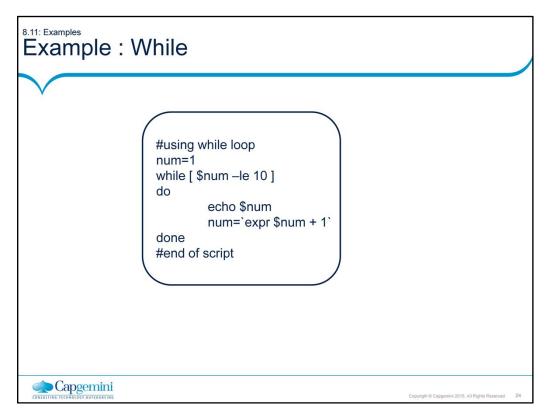## Syntax and Example

– Syntax:

while <condition is true>

do

      <execute statements>

done

e.g.

```
while [ $x -gt 3 ]
        do
                ps -a
                sleep 5
        done
```

```
while true
        do
                ps -a
                sleep 5
        done
```

Example: Script to edit, compile and execute a program.

```
while true
Do
  cc $1
case $? In
o) echo "Compilation Successful"
   echo "Executing a.out"
   a.out ; exit ;;
*) echo "Compilation Error"
   echo "Press <Enter> to edit"
   read pause
   vi $1 ;;
Esac
done
```

8.11: Examples
Example : While

```
#using while loop
num=1
while [ $num –le 10 ]
do
          echo $num
          num=`expr $num + 1`
done
#end of script
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

In the above example, the loop executes till the condition is true. This is till the value of the variable num is < 10.

8.12: Break & Continue Statement
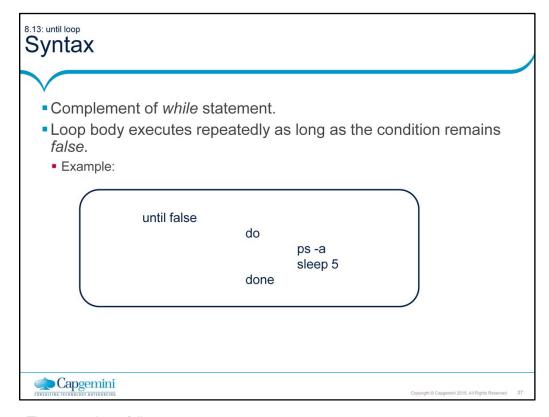
break and continue statement

- Continue:
  - Suspends statement execution following it.
  - Switches control to the top of loop for the next iteration.

- Break:
  - Causes control to break out of the loop.

8.12: Break & Continue Statement
Example

```
while echo "designation : \c"
                    do
                    read desig
                    case "$desig" in
                        [0-9]) if grep "^$desig" emp.lst >/dev/null
                                    then
```

In above example, the *while* loop is an unending loop as **echo " designation : \c "** statement (which is put as a condition in the while loop) always returns an exit status of success (condition becomes true).

Hence, it is more efficient if you write the following as a single statement:
while true
 echo "designation : \c"

In the above program if you enter a designation as a two digit number, it matches with case [0-9][0-9]. If the designation found in the file break statement is executed, control comes out of the loop and the program halts.
Otherwise, the default case is executed. Continue statement transfers the control at the beginning of the loop.

8.13: until loop
Syntax

- Complement of *while* statement.
- Loop body executes repeatedly as long as the condition remains *false.*
  - Example:

```
until false
              do
                      ps -a
                      sleep 5
              done
```

The syntax is as follows:
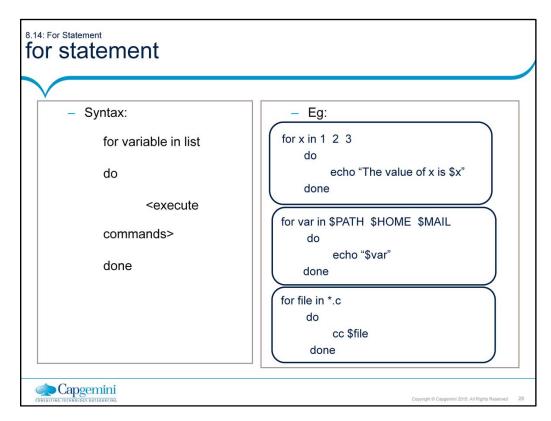**until condition**
do
  commands
Done

This loop is a complement of the *while* loop. In the *while* loop statements are repeated till the condition is *true.* But in an *until* loop, statements inside loop are repeated till the condition is *false.* As soon as the condition becomes true, the iteration stops.

In the above example given until loop is infinite loop.

8.14: For Statement
for statement

– Syntax:

```
for variable in list

do

        <execute

commands>

done
```

– Eg:

```
for x in 1  2  3
    do
        echo "The value of x is $x"
    done
```

```
for var in $PATH  $HOME  $MAIL
    do
        echo "$var"
    done
```

```
for file in *.c
    do
        cc $file
    done
```

Example 1:
In this example, *for* loop executes *three* times because three numbers are there in the list . In every iteration *x* is assigned 1, 2 and 3 respectively.
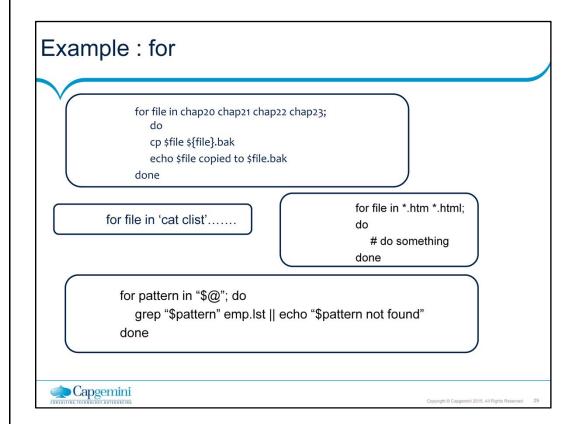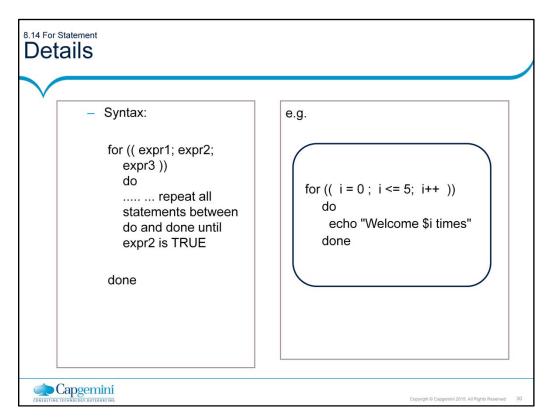
Example 2:
In this example also, *for* loop executes 3 times. In each iteration, *var* takes values from system variables in the list $PATH, $HOME and $MAIL respectively.

Example 3:
In this example, the for loop iterates equal to the number of files with extension c in the current working directory. This is because *.c is replaced with a list of all files with extension *c* in the current working directory.
Some more examples are:

```
for i in 1 2 3 4 5 6 7 8 9 0
do
                echo $i
done
```

Example : for

```
for file in chap20 chap21 chap22 chap23;
    do
    cp $file ${file}.bak
    echo $file copied to $file.bak
done
```

```
for file in 'cat clist'…….
```

```
for file in *.htm *.html;
do
    # do something
done
```

```
for pattern in "$@"; do
    grep "$pattern" emp.lst || echo "$pattern not found"
done
```

8.14 For Statement
Details

- Syntax:

```
for (( expr1; expr2;
   expr3 ))
   do
   ..... ... repeat all
   statements between
   do and done until
   expr2 is TRUE

   done
```

e.g.

```
for ((  i = 0 ;  i <= 5;  i++  ))
   do
    echo "Welcome $i times"
   done
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

In above example, syntax before the first iteration, *expr1* is evaluated. This is usually used to initialize variables for the loop. All statements between *do* and *done* are executed repeatedly until the value of *expr2* is true.

After each iteration of the loop, *expr3* is evaluated. This is usually used to increment a loop counter.

The output of the given example is:
Welcome 0 times
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times

8.15: Examples
Example : Until

```
#script to create a employee file
ans="y"
until [ $ans = "N" –o $ans = "n" ]
do
        echo "Enter the name :\c"
        read name
        echo "Enter the grade :\c"
        read grade
        echo "Enter the basic :\c"
        read basic
        echo $name: $grade : $basic >>emp
echo "Want to continue (Y/N) :\c"
read ans
done
#end of script
```

In above example the loop executes till the condition is false. This is as soon as the user enters "N" or "n" for *ans*, the condition is true and the loop stops iteration.

Some more examples of shell script are:
Script to accept five numbers and display their sum:
> **echo the parameters passed are : $1, $2, $3, $4, $5**
> **echo the name of script is   : $0**
> **echo the number of parameters passed are : $#**
> **#calculate the sum**
> **sum=`expr $1 + $2 + $3 + $4 + $5`**
> **echo the sum is $sum**
#end of script

Invoke this script as follows:
$sh disp_sum 10 12 13 14 15
The above command is to be followed by 5 different number as shown.

8.16: Shell functions

Functions in Shell Script

- Use shell functions to modularize the script.

- These are also called as script module

- Normally defined at the beginning of the script.

- Syntax (Function Definition):

  functionname(){

      commands

      }

- Example: Function to create a directory and change directories:

- Use mkcd mydir to call the function. mydir is used as $1 in the function.

```
mkcd()
{
mkdir $1       --$1 is the argument we pass while calling function
cd $1
}
```

You can also call the shell function *script module* as it makes a whole script section available under a single name. Normally, shell functions are defined at the beginning of the script. Or several functions can be stored in a file and read whenever they are needed. Files are stored in the *bin* directory. Function name can be any combination from the regular character string.
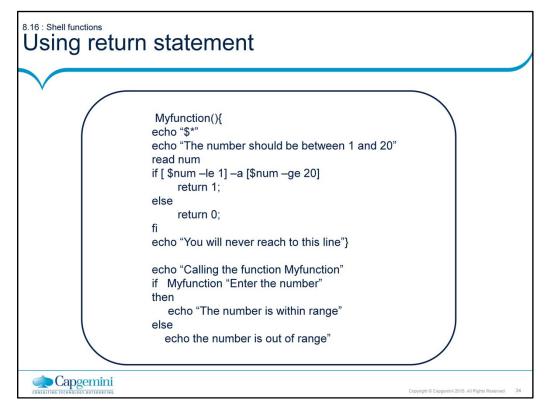
8.16 : Shell functions

Using return statement

- Used to come out of a function from within.
  - If called *without* an argument, function return value is the same as *exit* status of the last command executed within the function
  - If called *with* an argument it returns the argument specified.
  - Example:

```
functret()
{
command1
if .......
then
      return 1
else
      return 0
fi
Command2
}
```

8.16 : Shell functions
Using return statement

```
 Myfunction(){
echo "$*"
echo "The number should be between 1 and 20"
read num
if [ $num –le 1] –a [$num –ge 20]
      return 1;
else
      return 0;
fi
echo "You will never reach to this line"}

echo "Calling the function Myfunction"
if   Myfunction "Enter the number"
then
    echo "The number is within range"
else
    echo the number is out of range"
```

In the above example, *Myfunction* is called in the *if* statement with message "enter the number". This message is passed as three arguments.

In *Myfunction,* the first line is echo $*.
Hence, it display message "Enter the number".
Read num accepts the number.
If the number is between 1 and 20, the function returns 1, otherwise it returns 0.

If *Myfunction* returns 1, then the output is:
                    The number is within range.

Otherwise the output should be as follows:
                    The number is out of range.

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

Why are we using UNIX over WINDOWS for database

- VAST OPEN SOURCE MANAGEMENT UTILITIES
- FLEXIBILITY
- FAST PERFORMANCE

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX
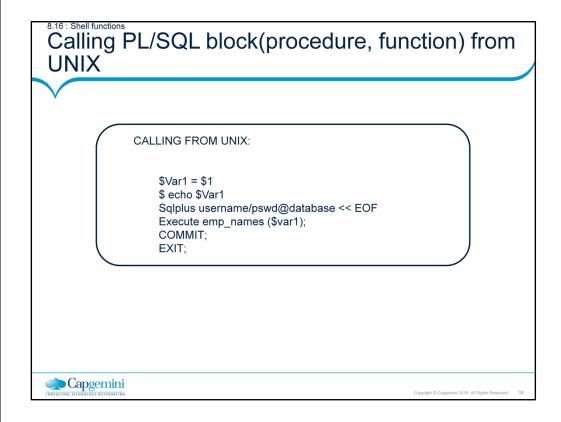
Steps for invoking stored procedure and function in UNIX

- On the UNIX command prompt type the following command
  Sqlplus   –s –l username/pswd@db

- If output variable present in the sub program then
  Var "variablename" datatype;

- Execute proc_name(input_variable);

- EOF

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

```
CREATE OR REPLACE PROCEDURE emp_names (deptno IN NUMBER)
 Is
     Ename VARCHAR2(10);
     CURSOR cur (deptno NUMBER) IS
     SELECT last_name from EMPLOYEES
     WHERE dept_id = deptno;
Begin
OPEN cur(deptno);
Loop
FETCH cur into ename;
EXIT WHEN cur%notfound;
Dbms_output.Put_line(ename);
End loop;
CLOSE cur;
EXCEPTION
When no_data_found then
 Dbms_output.Put_line (sqlerrm);
 End;
```

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

CALLING FROM UNIX:

```
$Var1 = $1
$ echo $Var1
Sqlplus username/pswd@database << EOF
Execute emp_names ($var1);
COMMIT;
EXIT;
```
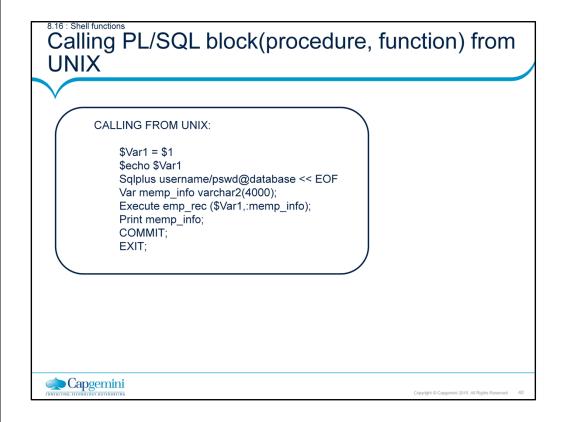
8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX
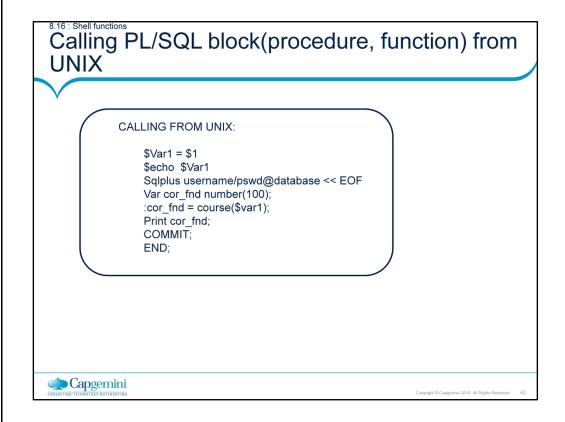
```
CREATE OR REPLACE PROCEDURE emp_rec (
 emp.no  IN  Employees.Employee_id%type,
 Emp_info    OUT  EMPLOYEES%ROWTYPE)
Is
Begin
 SELECT * INTO emp_info
 From Employees
 WHERE EMPLOYEE_ID = emp_no;
Exception
 When others then
  Raise_application_error(-20001,'an error was encountered –
  '||SQLCODE||' -ERROR- '||SQLERRM);
End;
```

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

CALLING FROM UNIX:

```
$Var1 = $1
$echo $Var1
Sqlplus username/pswd@database << EOF
Var memp_info varchar2(4000);
Execute emp_rec ($Var1,:memp_info);
Print memp_info;
COMMIT;
EXIT;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

```
CREATE OR REPLACE function course(name IN varchar2 )
  RETURN number
   Is
   cnum number;
   cursor cur  IS
   SELECT course_number
   FROM courses_tbl
   WHERE course_name = name;
BEGIN
   Open cur;
   Fetch cur into cnum;
   if cur%notfound then
   cnum := 7777;
   End if;
   Close cur;
   Return cnum;
End;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Shell functions

Calling PL/SQL block(procedure, function) from UNIX

CALLING FROM UNIX:

```
$Var1 = $1
$echo  $Var1
Sqlplus username/pswd@database << EOF
Var cor_fnd number(100);
:cor_fnd = course($var1);
Print cor_fnd;
COMMIT;
END;
```
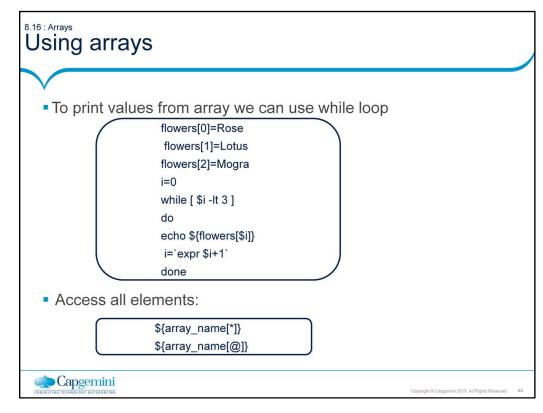
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Arrays

Using arrays

- Contains a collection of values accessible by individuals or groups
  - Subscript of array element indicates their position in the array.
    - arrayname[subscript]
- First element is stored at subscript 0.
  - Assign a value in *flowers* array at the first position.
    - Flowers[0]=Rose
- Assign values in an array with a single command:
  - $ set -A Flowers Rose Lotus
- Access individual array elements
  - ${arrayname[subscript]}

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

8.16 : Arrays
Using arrays

- To print values from array we can use while loop

```
flowers[0]=Rose
 flowers[1]=Lotus
flowers[2]=Mogra
i=0
while [ $i -lt 3 ]
do
echo ${flowers[$i]}
 i=`expr $i+1`
done
```

- Access all elements:

```
${array_name[*]}
${array_name[@]}
```

You can display all elements from the array using * or @ symbol:
Num[0]="Zero"
Num[1]="One"
Num[2]="Two"
Num[3]="Three"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"

Summary

- .profile:
  - Script executed during login time.
- *Command enclosed in backquotes (`):*
  - Shell executes the command first
  - Enclosed command text is replaced by the command the output.
- Test:
  - Command used to check the condition in an if statement.
- Different loop statements in Unix are:
  - For
  - While
  - Until

Summary

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Review Questions

- Complete The Following
  - ------- command can be replaced by test command.
  - ------ condition checks whether two strings are equal or not.
  - ------ loop terminates as soon as condition becomes true.

- TRUE OR FALSE
  - PS1 stores primary cursor string: