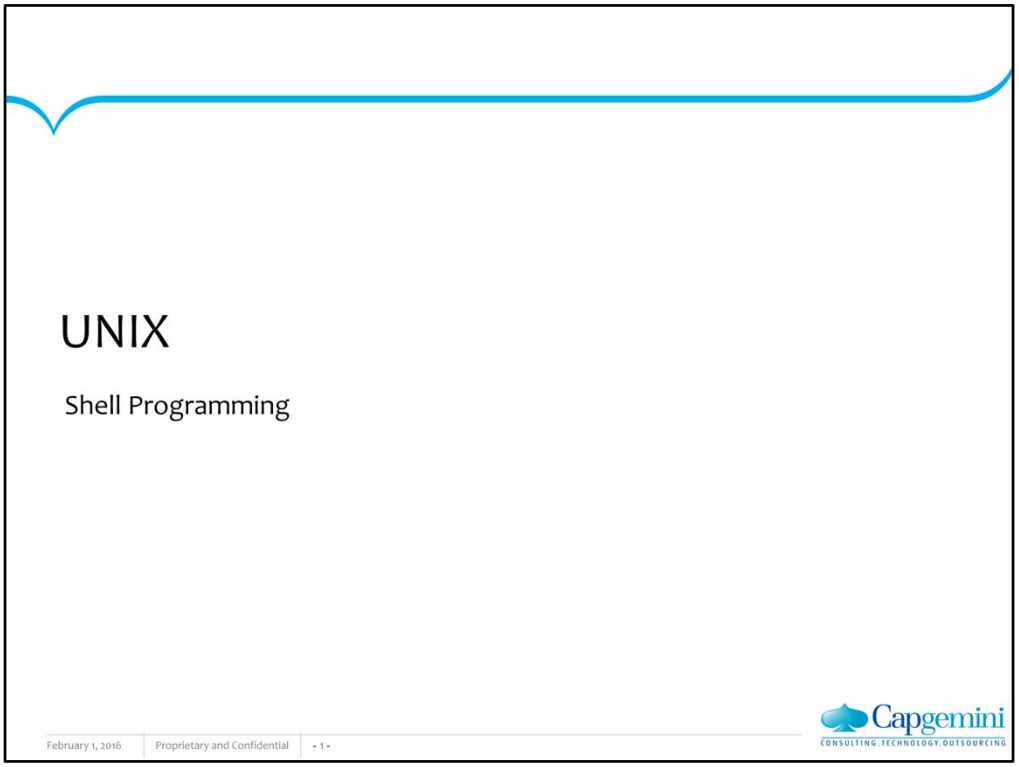
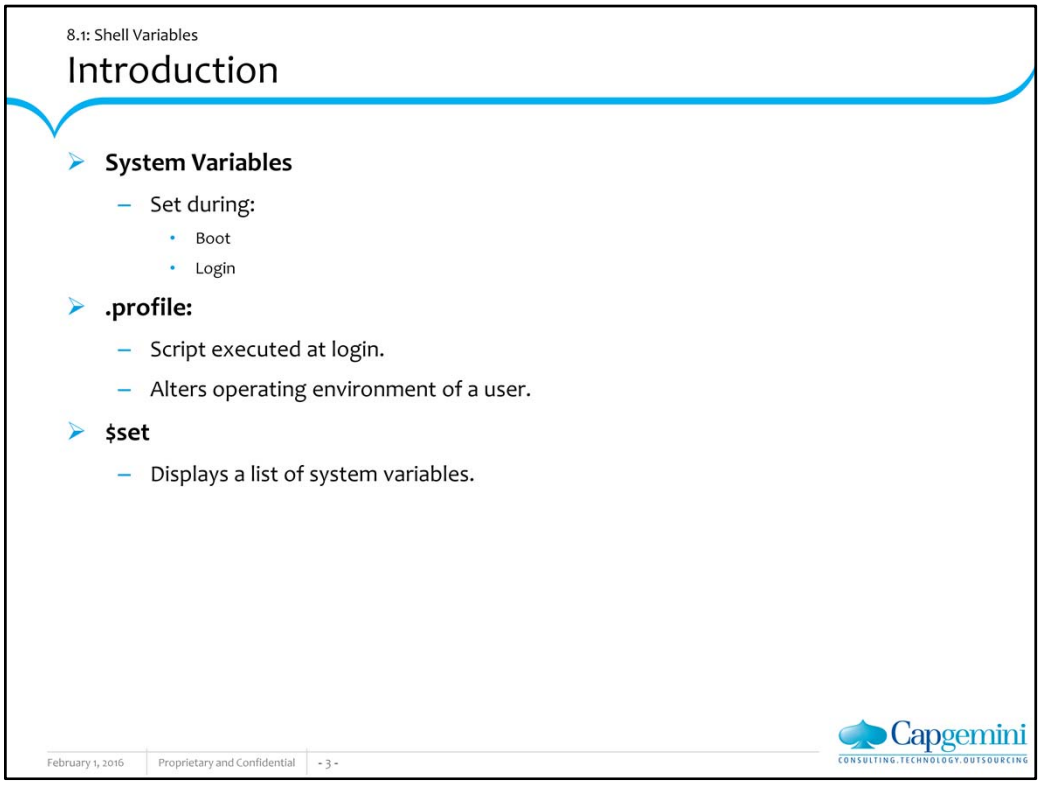

UNIX
Shell Programming
February 1, 2016
Proprietary and Confidential
- 1 -
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

## Lesson Objectives

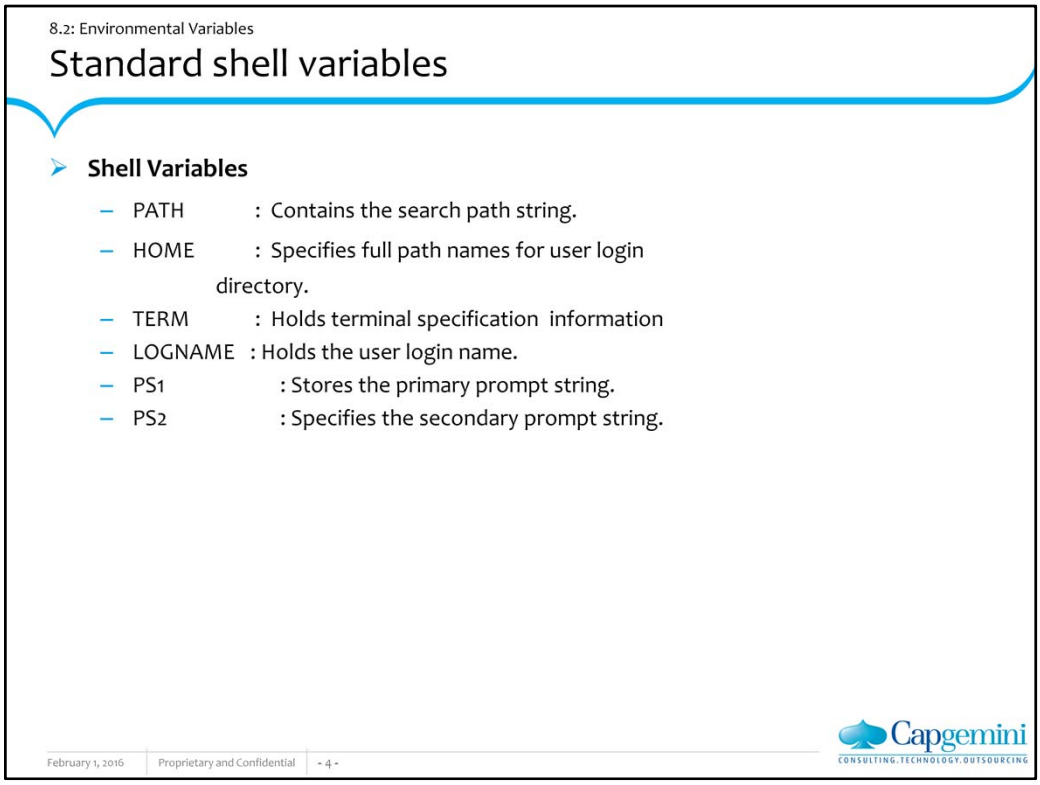
- **At the end of the session you will be able to understand:**
  - Shell variable
  - Environment variables
  - Shell script commands
  - Command substitution
  - Command line argument
  - Conditional statements
  - Iterative statements




**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
```




**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:

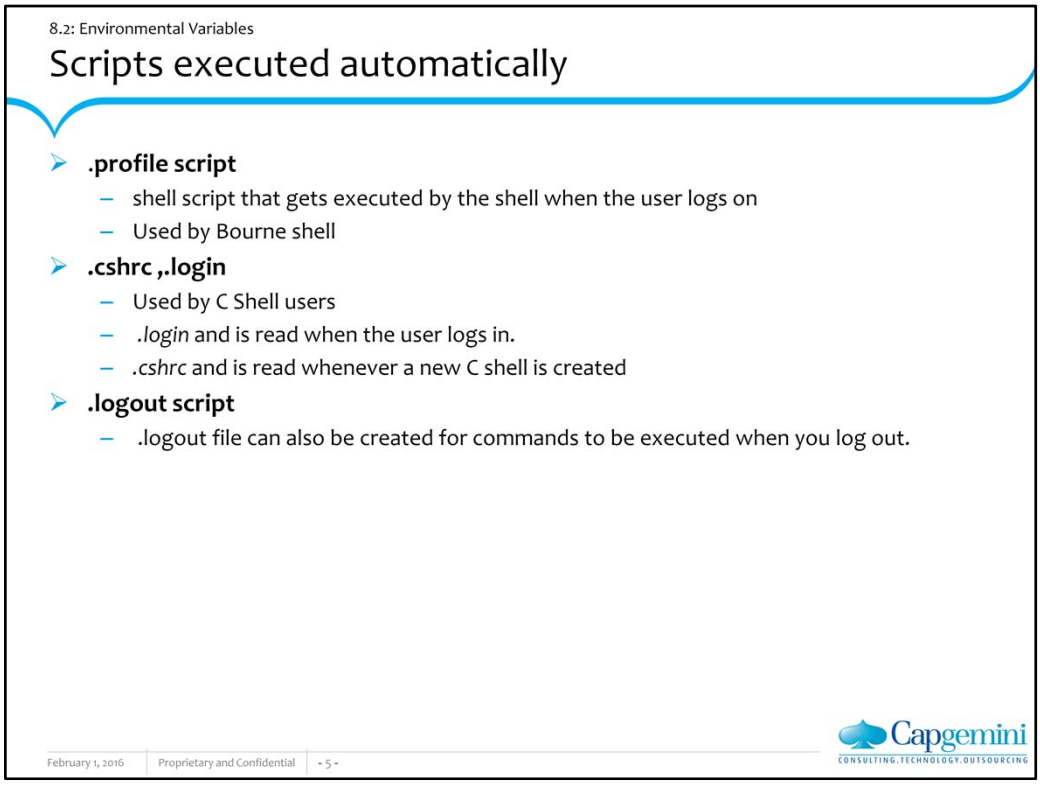
```
$ PATH=$PATH:/usr/user1/progs
```

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.

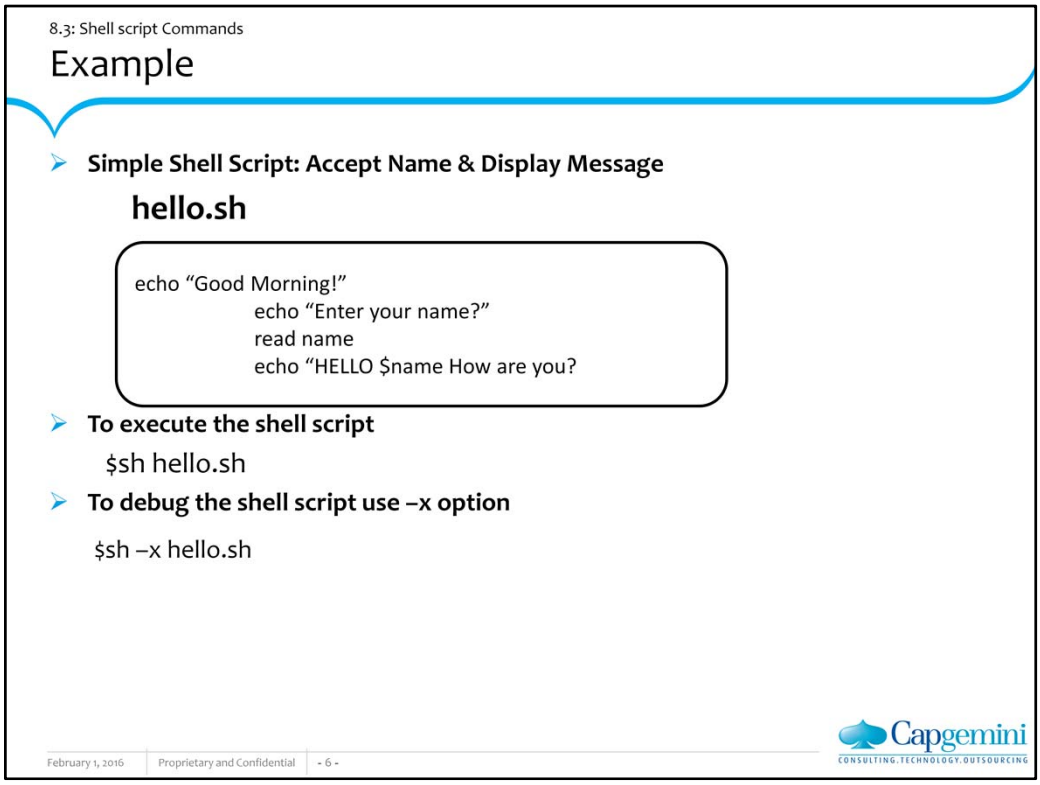



**.profile script**

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**

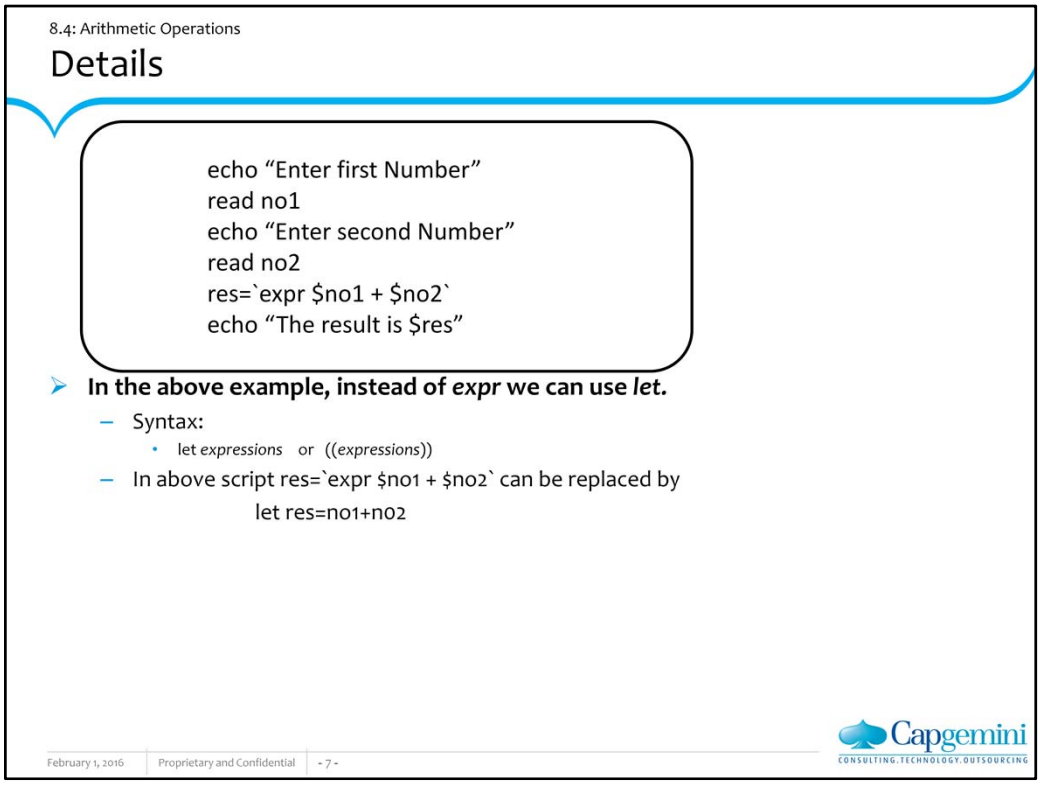
For the Bourne shell, the system reads the *.profile* file and executes the commands found there. C Shell users, however, have two files to read and execute. One is called *.login* and is read when the user logs in. The second is called *.cshrc* and is read whenever a new C shell is created, including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




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?

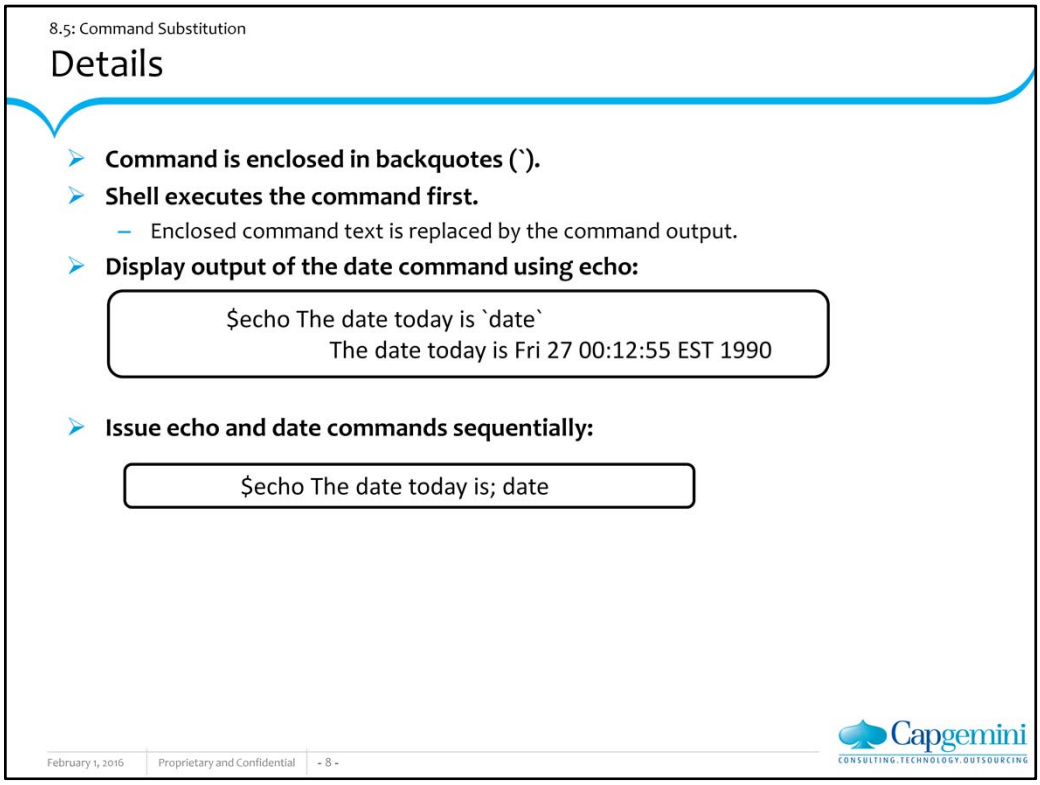



The above program accepts two numbers and displays their sum as a result. Instead of the *expr* command, we can use the *let* command.

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

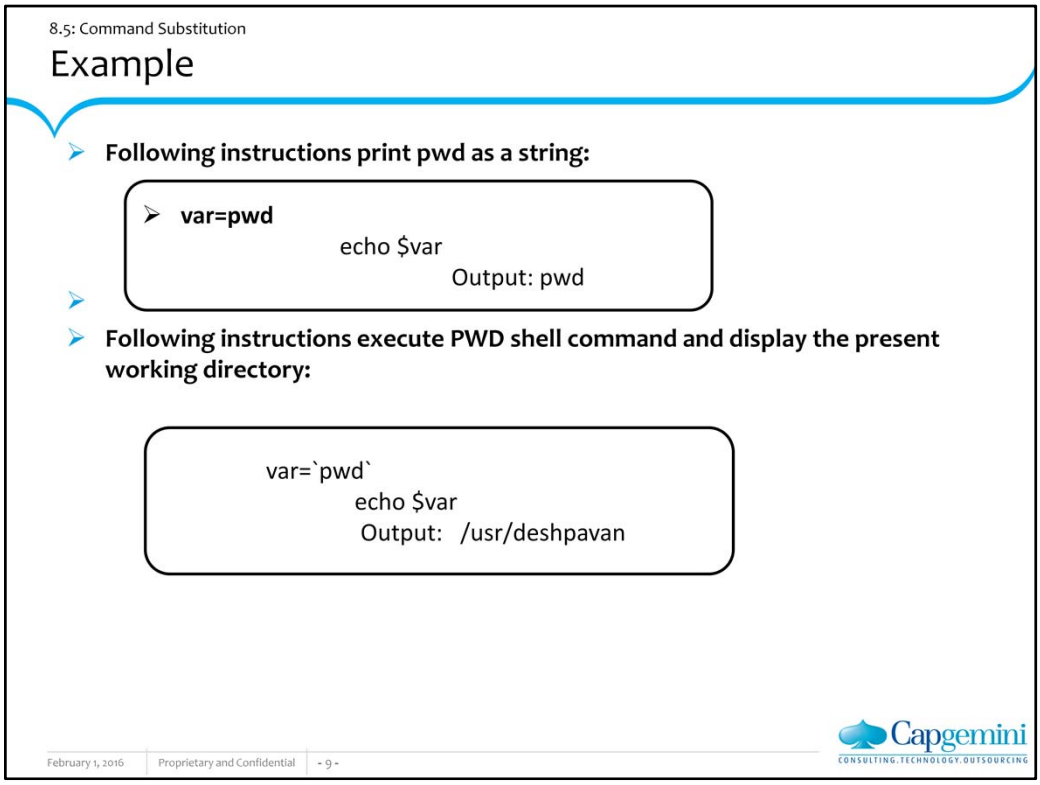
Add *one* to variable i. Using let statement:
- let i=i+1 If no spaces in expression
- let "i = i + 1" enclose expression in "… " if expression includes spaces
- (( i = i + 1 ))

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



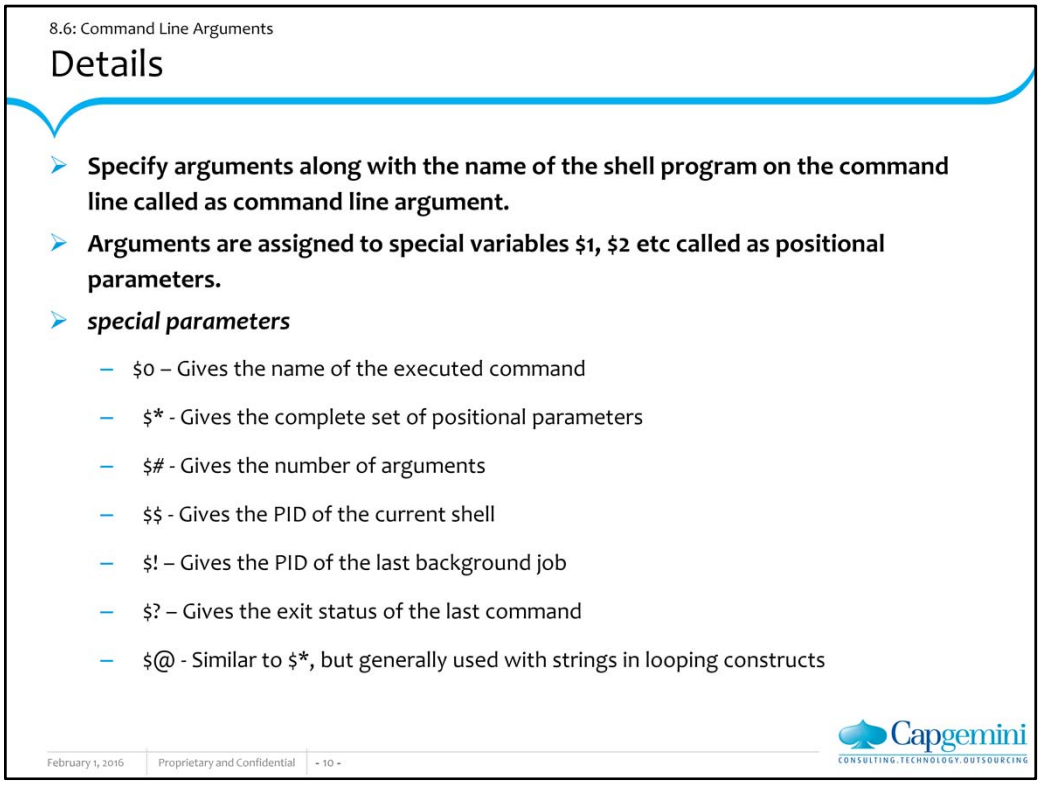

**$echo The date today is `date`**
In this command date is a command which is enclosed in backquotes and hence will get replaced by its output and then echo command will display message

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




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.

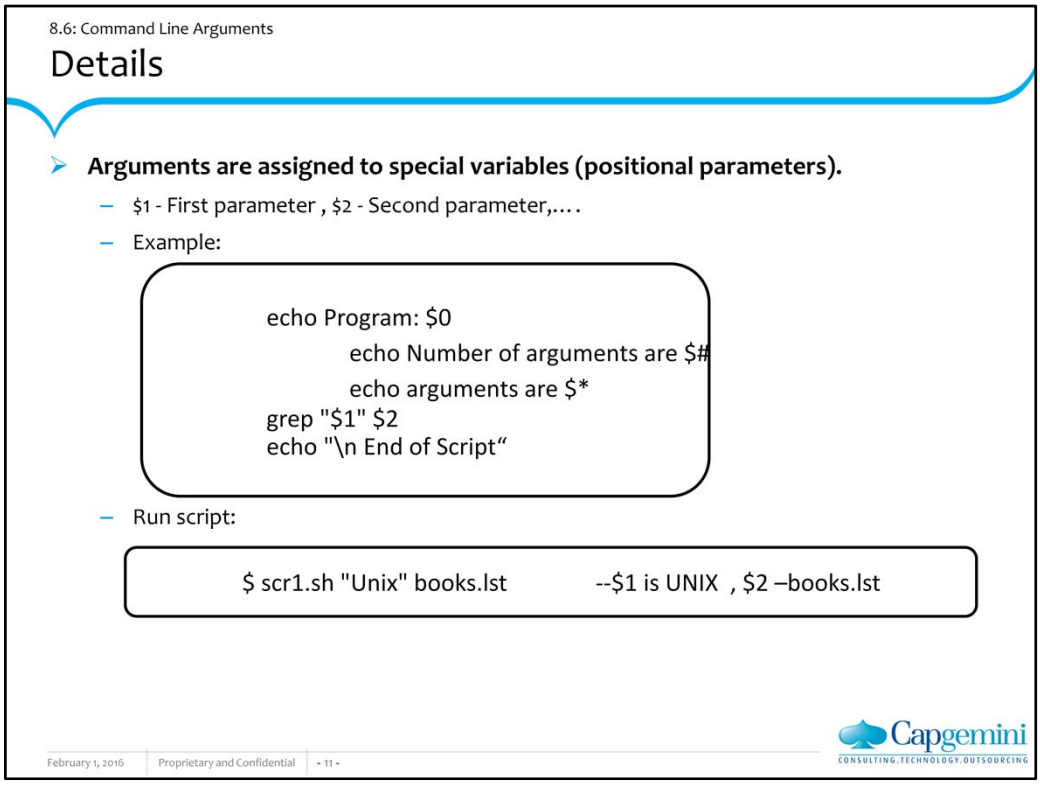



You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as *command line arguments*.

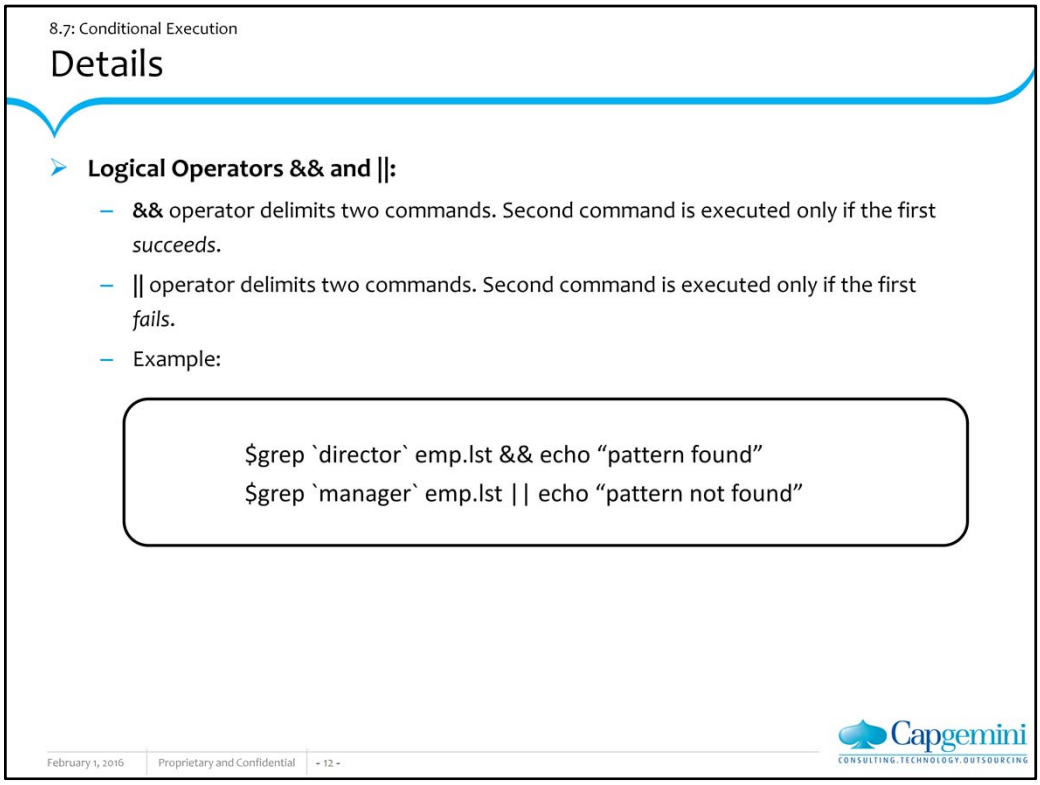
**Parameters Related to Command Line Arguments**

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




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.




**Conditional Execution using && and ||**

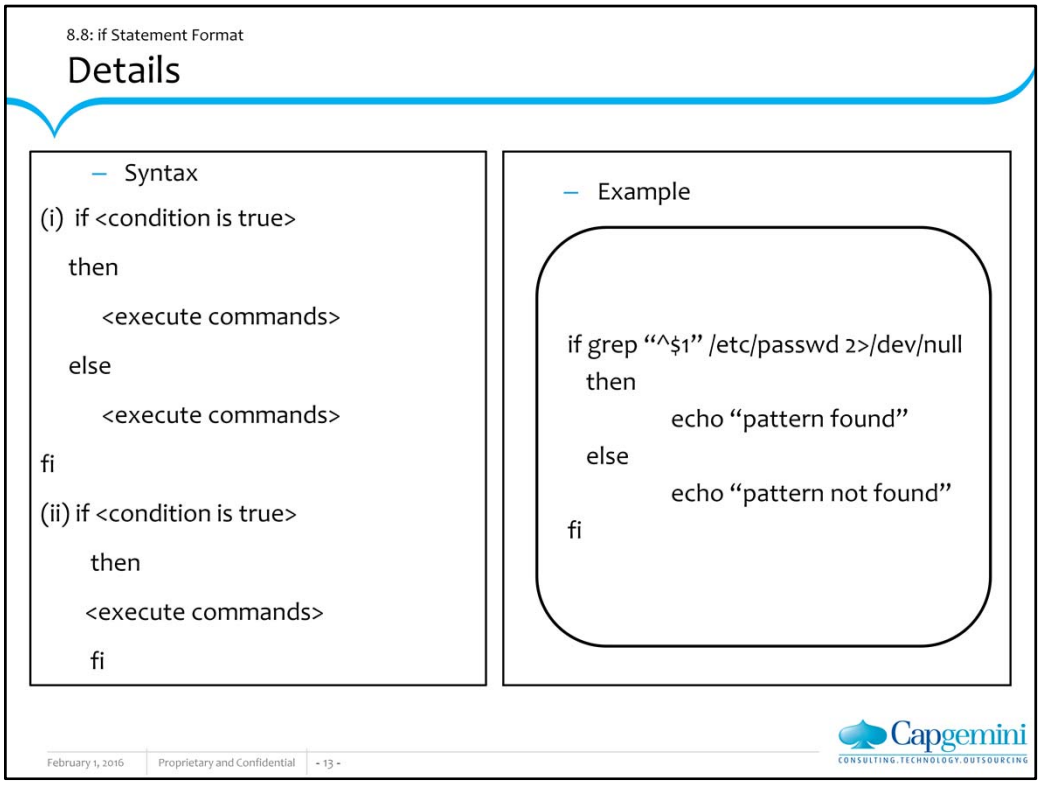
The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command. In case of &&, the second command executes only if the first has succeeded. Similarly, || will ensure that the second command is executed only if the first has failed.

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.

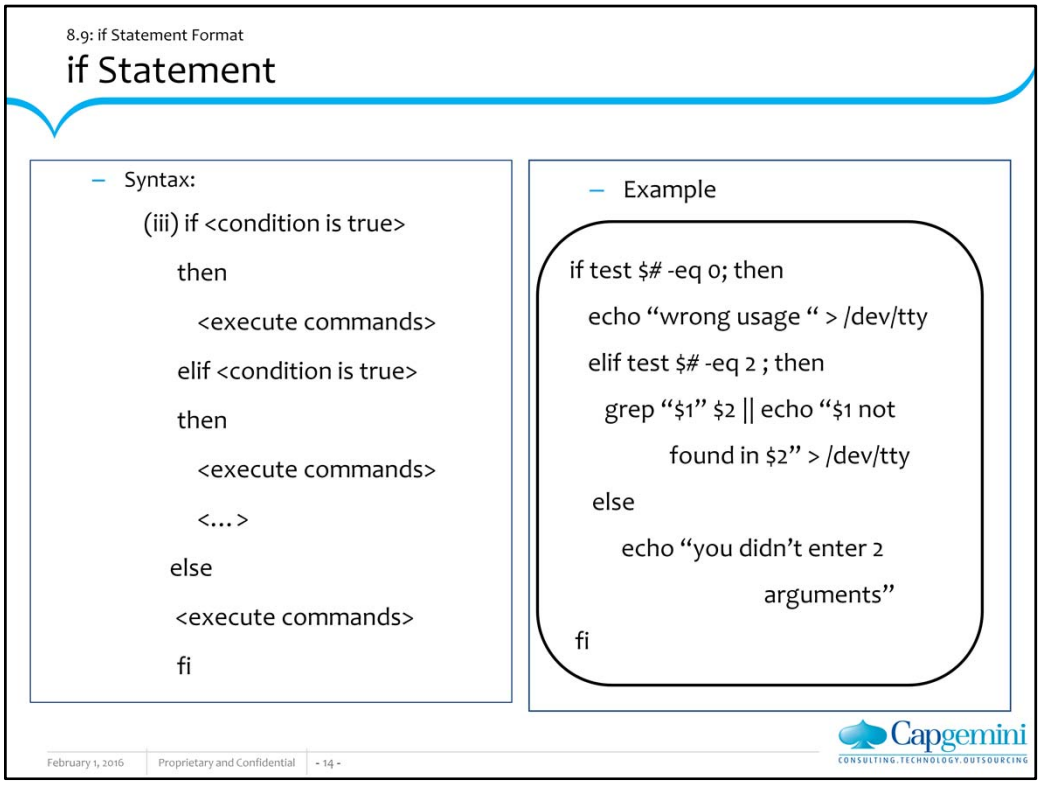
```
$ grep "WAP" books.lst || echo "Not Found..."
“Not Found...”
```




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.




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

8.9: test Statement

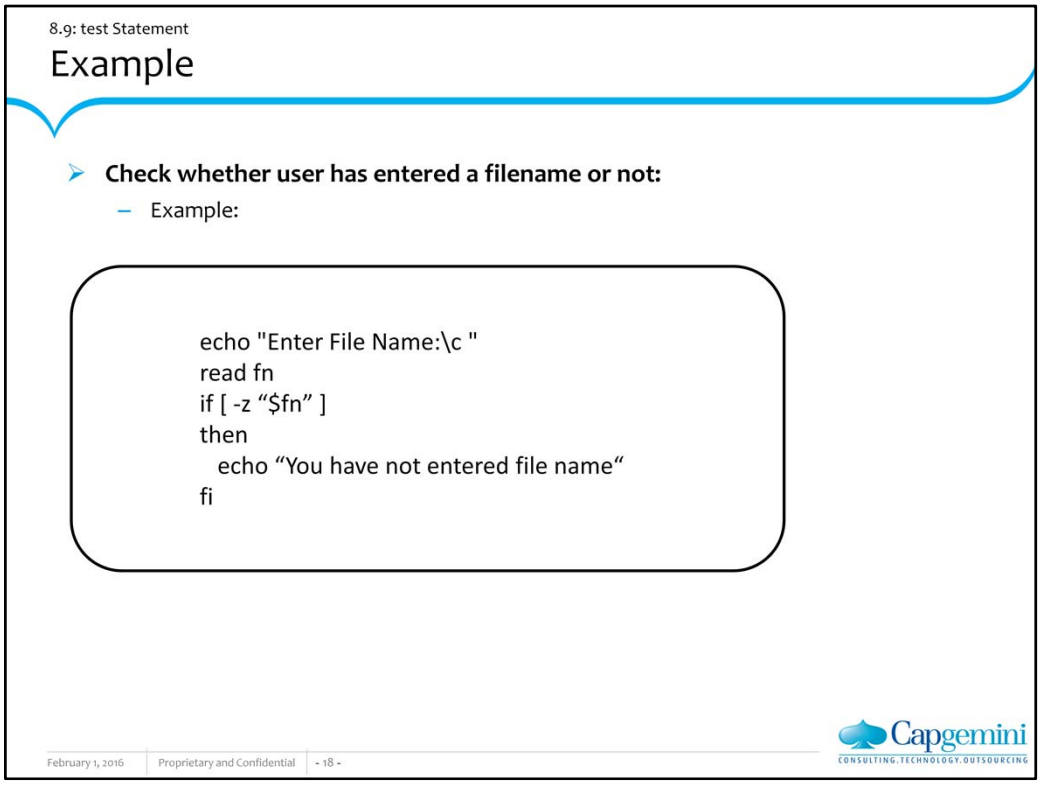
## Relational Operator for strings and logical operators

- **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 ≠ S2
  - str — True, if str is assigned and not null
- **Logical Operators**
  - -a .AND.
  - -o .OR.
  - ! Not

9.9: test Statement

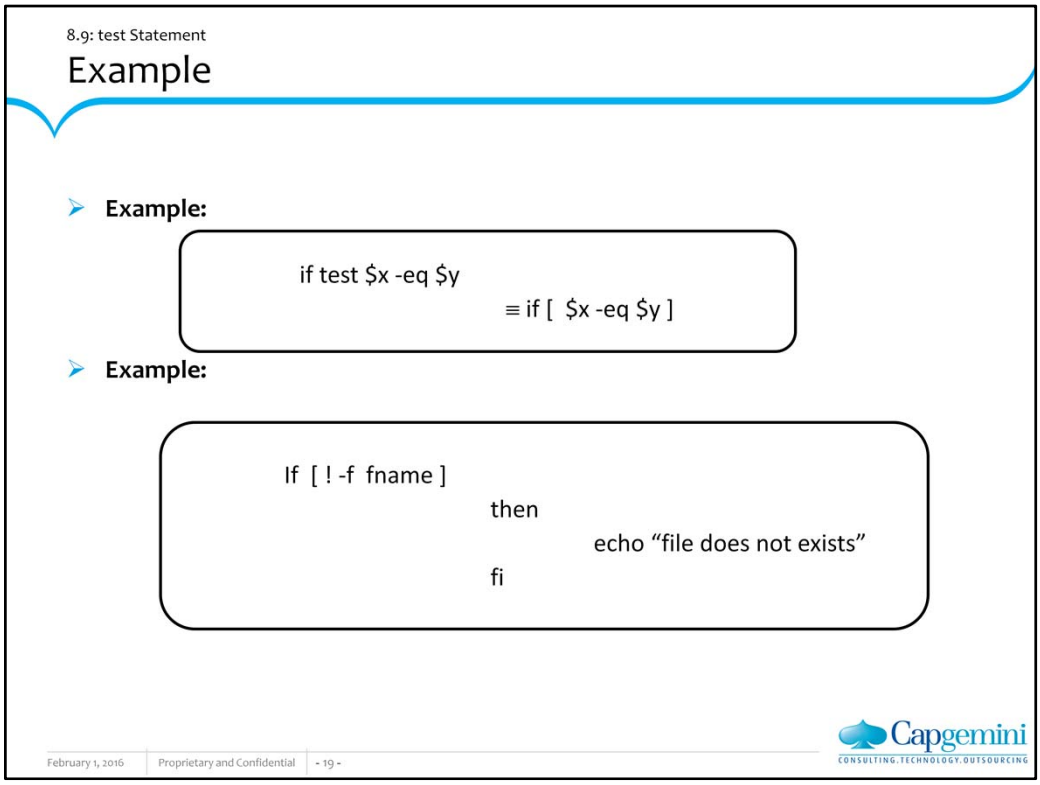
## File related operators

- **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




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".



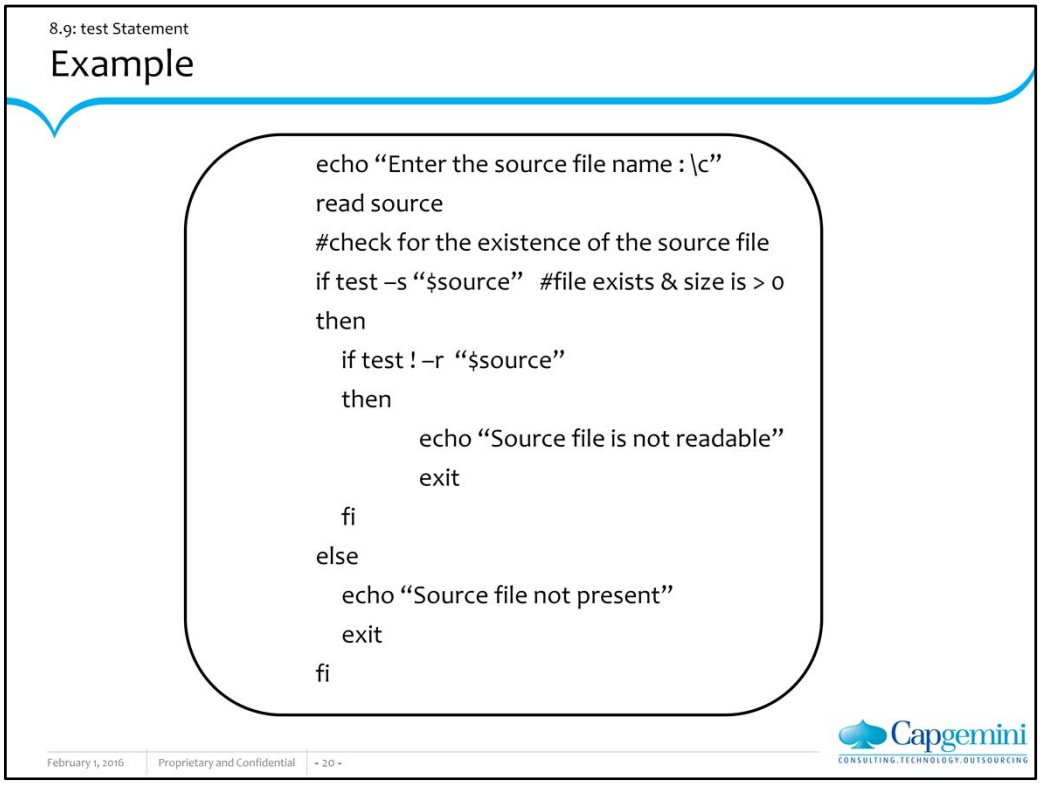

**if test $x -eq $y**
**≡ if [ $x -eq $y ]**

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

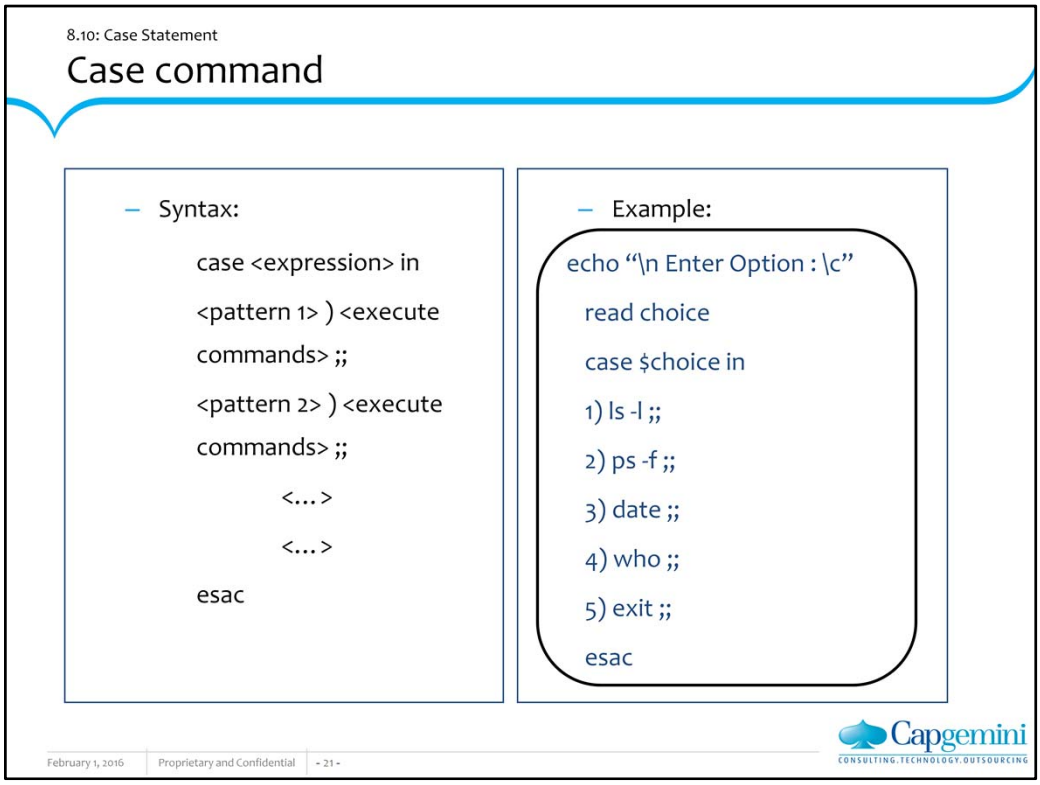
**test $x –eq $y** returns true if the values of variables x and y are equal. You can write the same condition as [ $x –eq $y ]. Here, instead of *test* command we use "[" (square bracket).

**If [ ! -f fname ]**

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




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



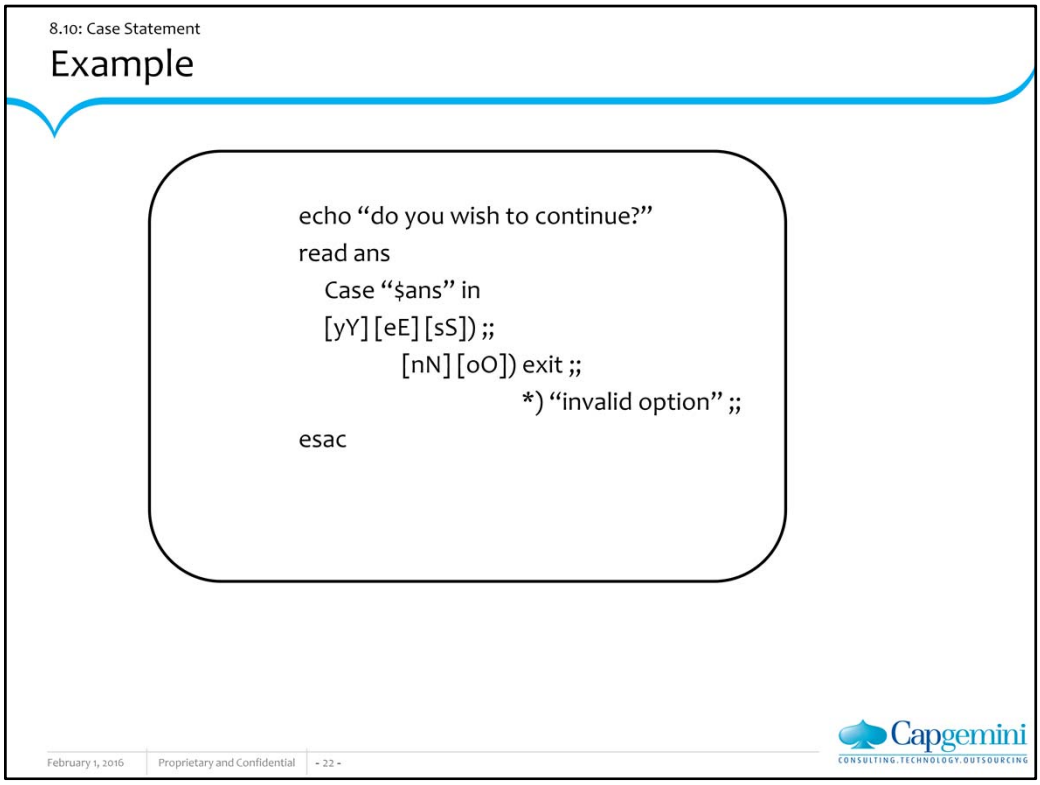

In a *case* statement you can also use commands enclosed in *backquotes*. The given example executes command **`date | cut -d " " –f1`** which returns only the day part. The output is used to execute the appropriate case.

**Example:**

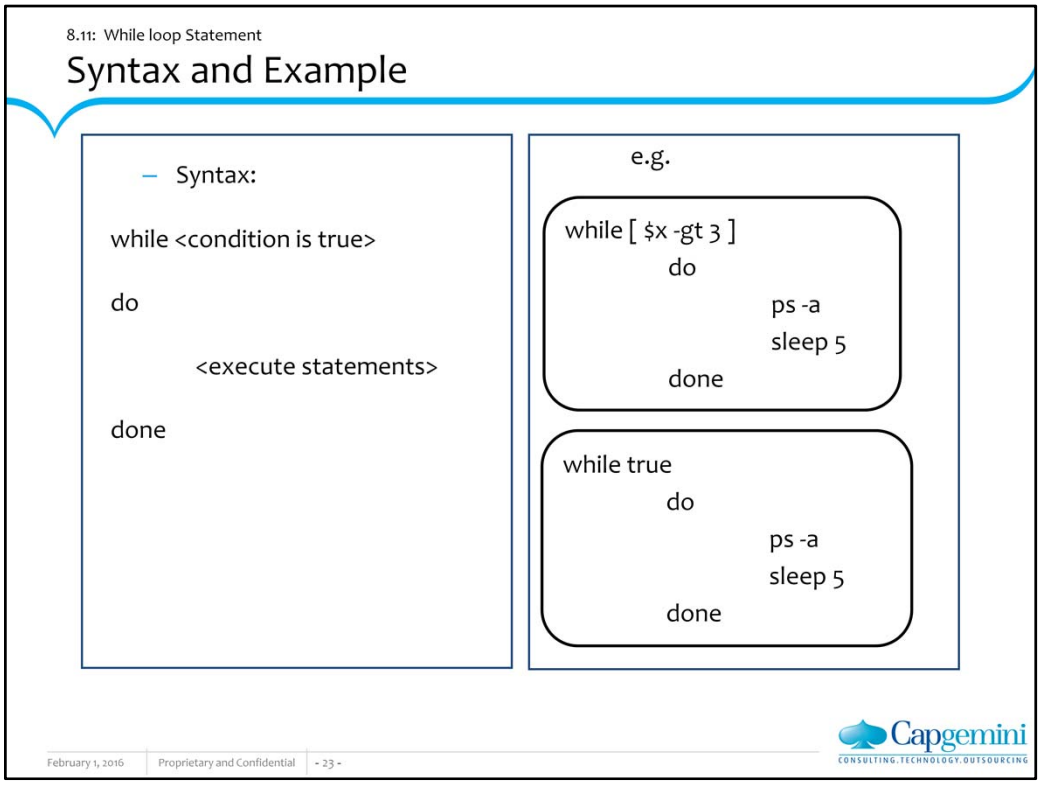
```
case `date | cut -d" " –f1` in
Mon ) <commands> ;;
Tue ) <commands> ;;
                :
esac
```

**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
```

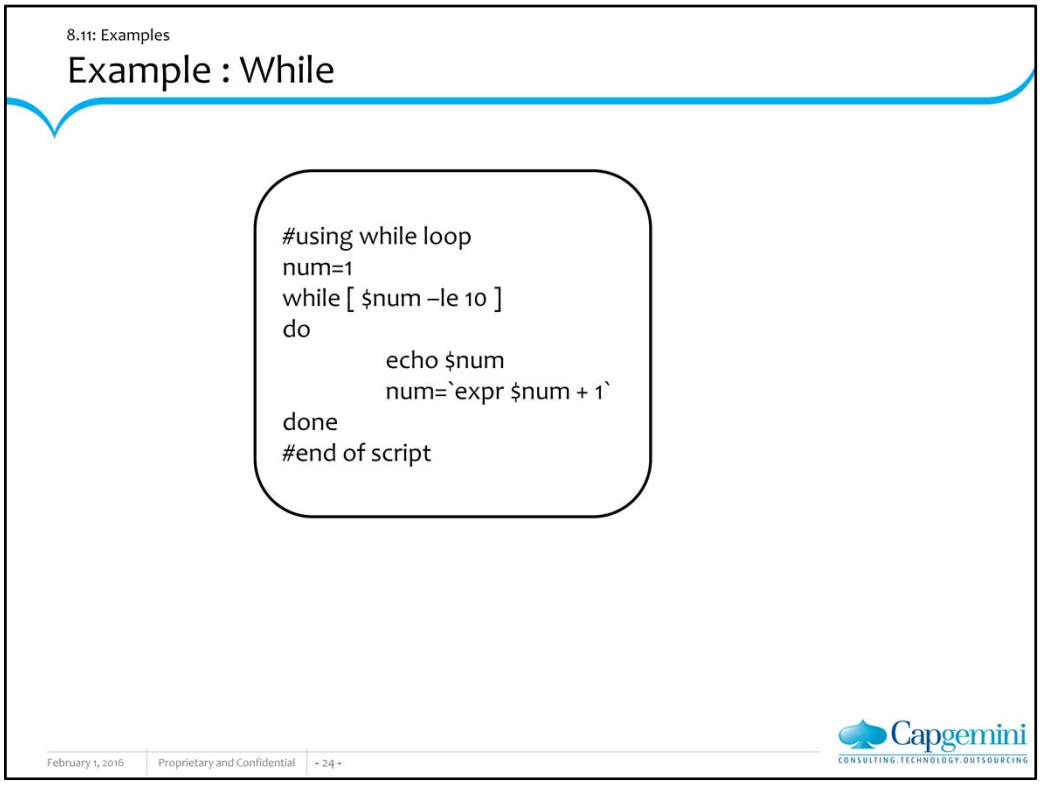



In the above example, the first case matches with "yes" or "YES". Similarly, the second case matches with "no" or "NO".




**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
```

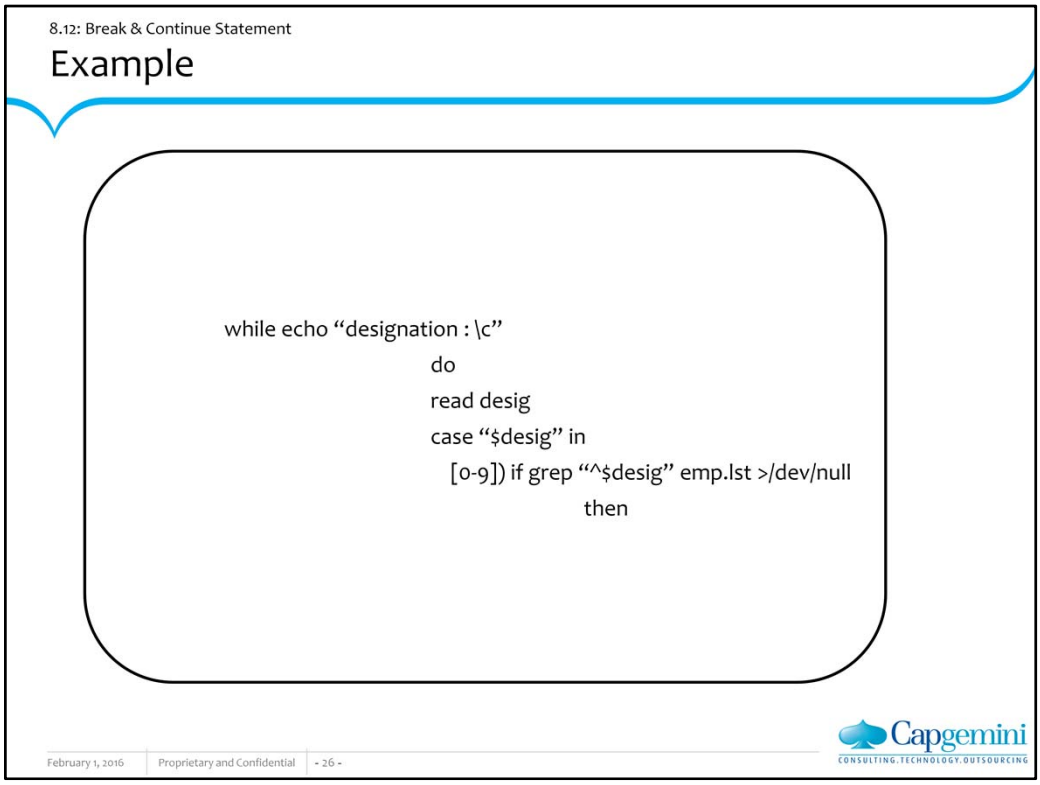



In the above example, the loop executes till the condition is true. This is till the value of the variable num is < 10.

8.12: Break & Continue Statement

## break and continue statement

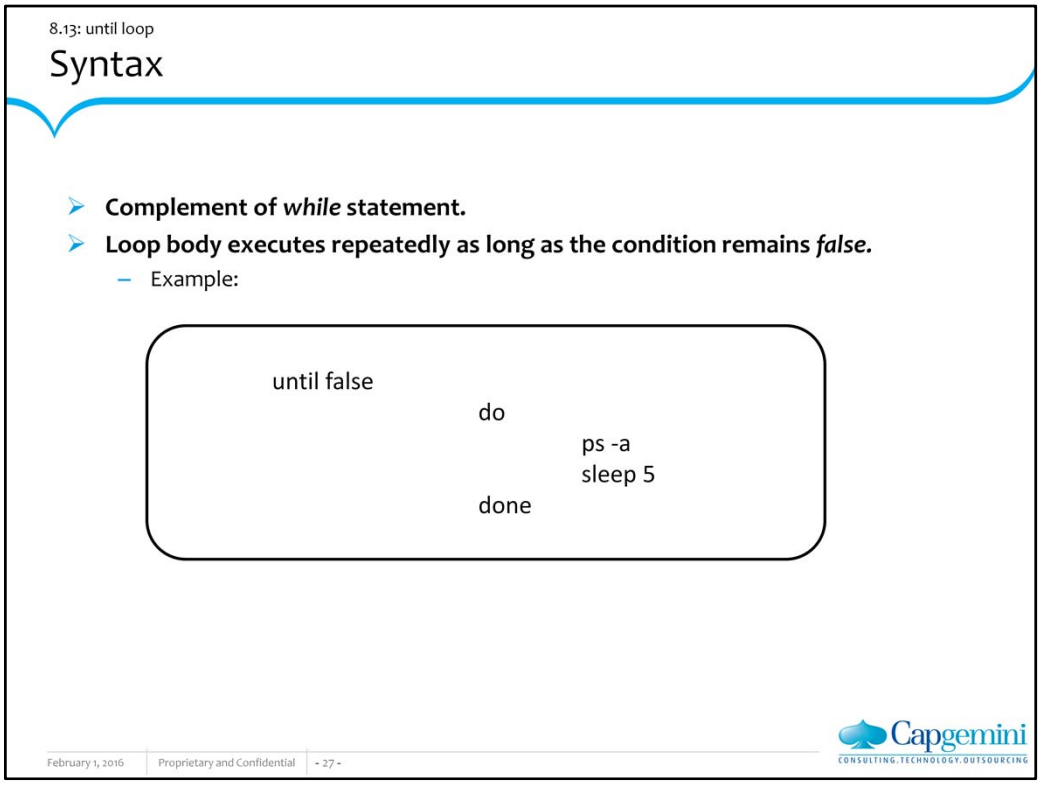
- **Continue:**
  - Suspends statement execution following it.
  - Switches control to the top of loop for the next iteration.
- **Break:**
  - Causes control to break out of the loop.




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"**

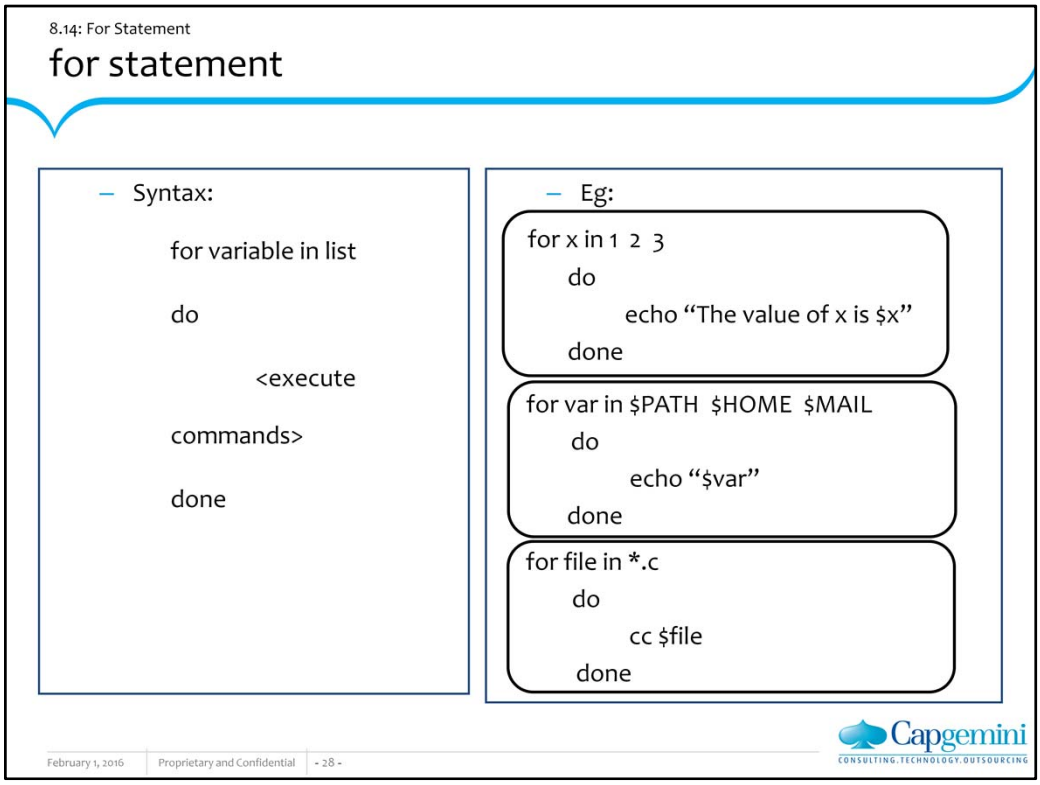
In the above program if you enter a designation as a two digit number, it matches with case [0-9][0-9]. If the designation found in the file break statement is executed, control comes out of the loop and the program halts.
Otherwise, the default case is executed. Continue statement transfers the control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




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.




**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
```

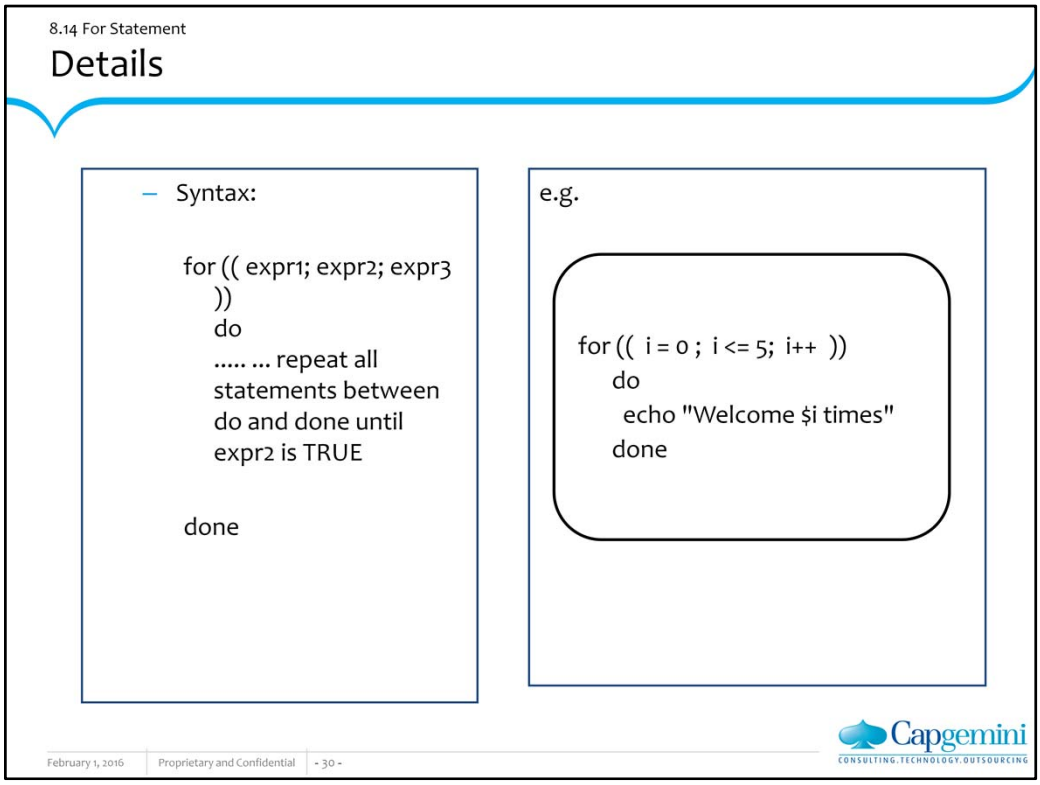
## Example : for

```
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
```

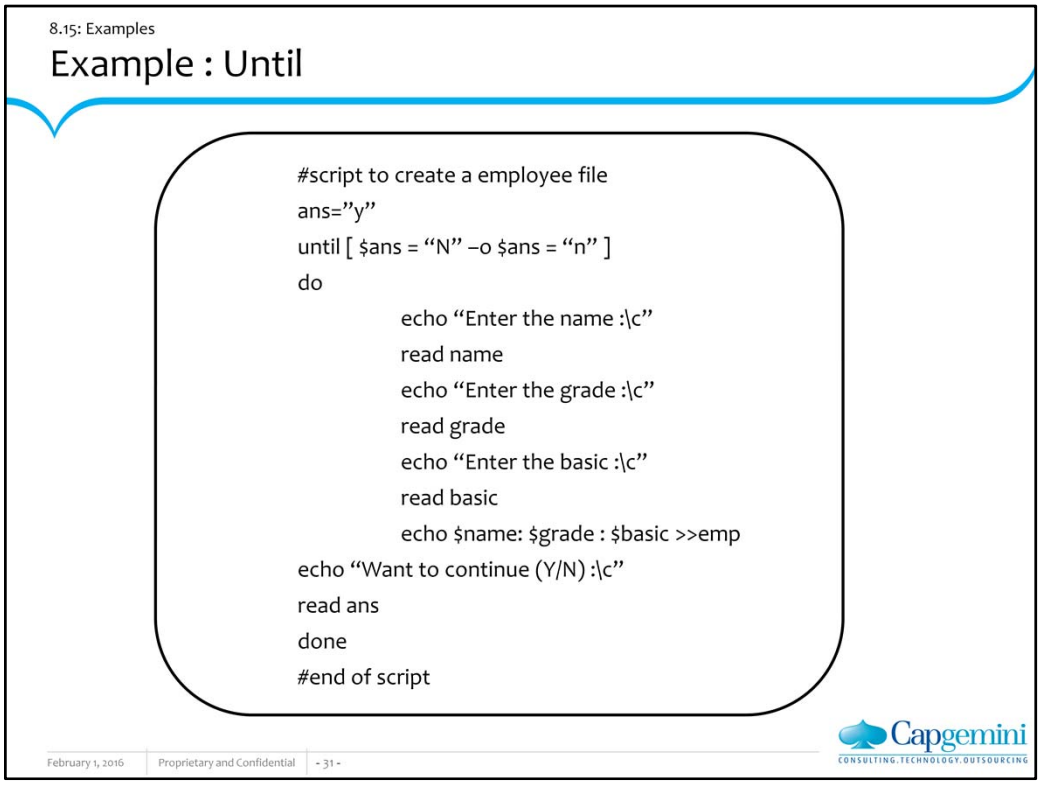
```
for pattern in "$@"; do
    grep "$pattern" emp.lst || echo "$pattern not found"
done
```




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




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:

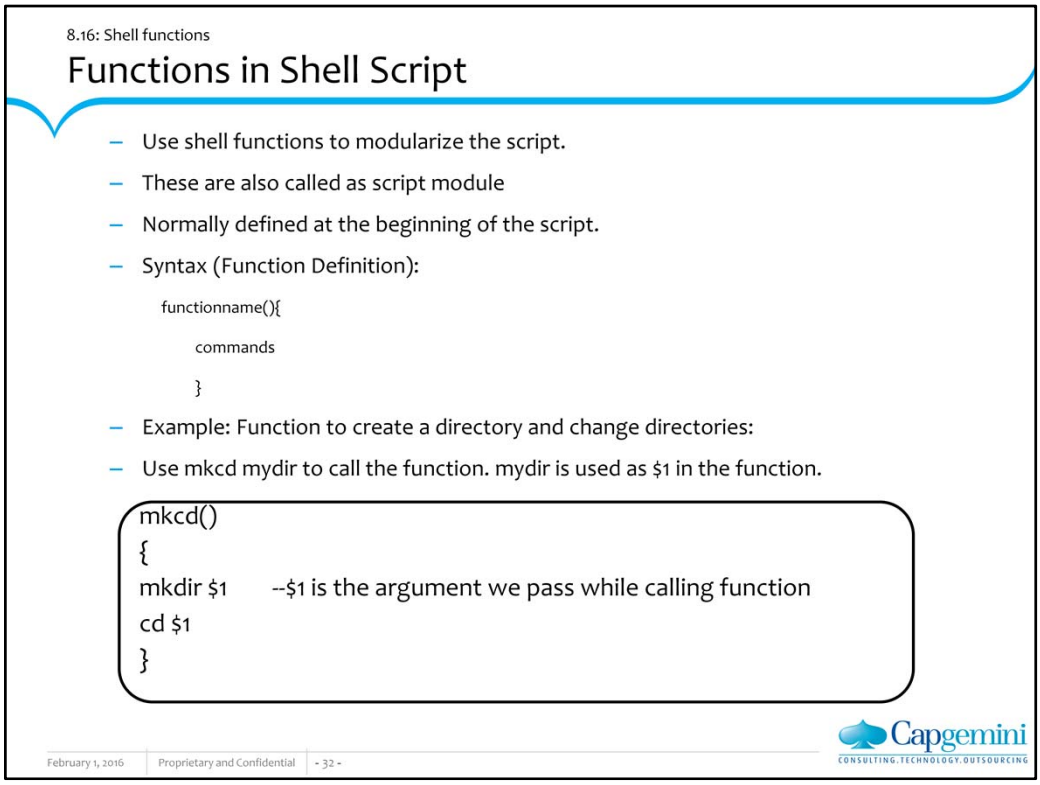
**Script to accept five numbers and display their sum:**

```
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.



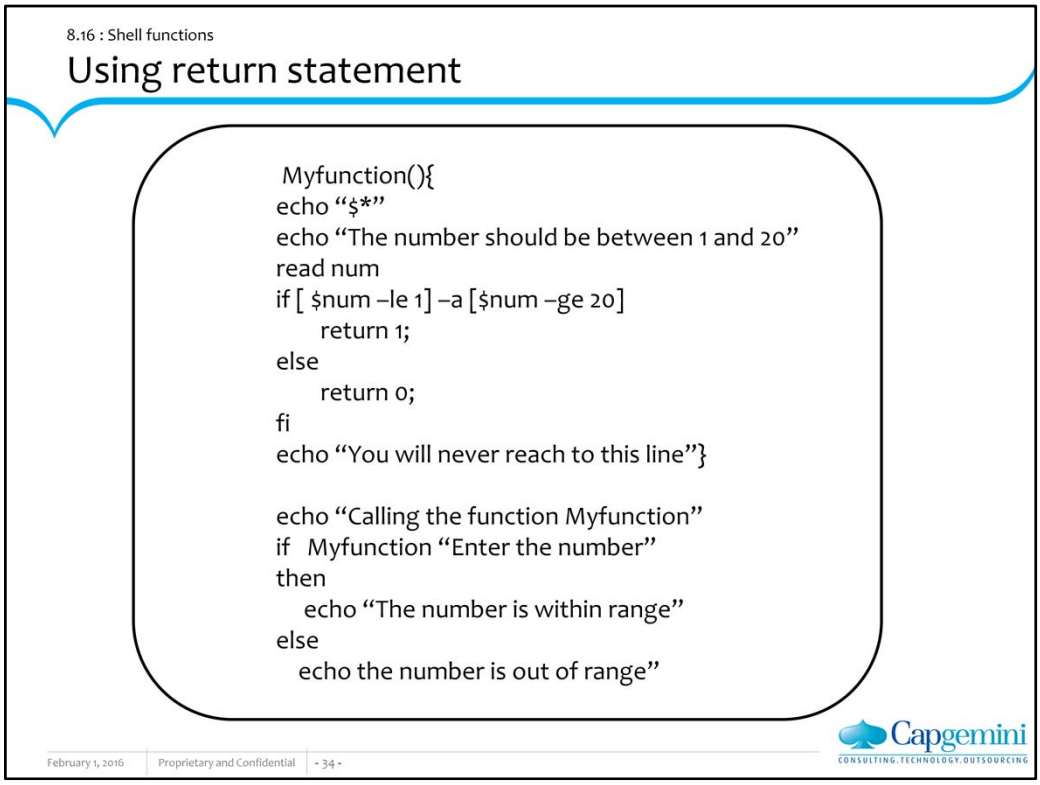

You can also call the shell function *script module* as it makes a whole script section available under a single name. Normally, shell functions are defined at the beginning of the script. Or several functions can be stored in a file and read whenever they are needed. Files are stored in the *bin* directory. 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
}
```

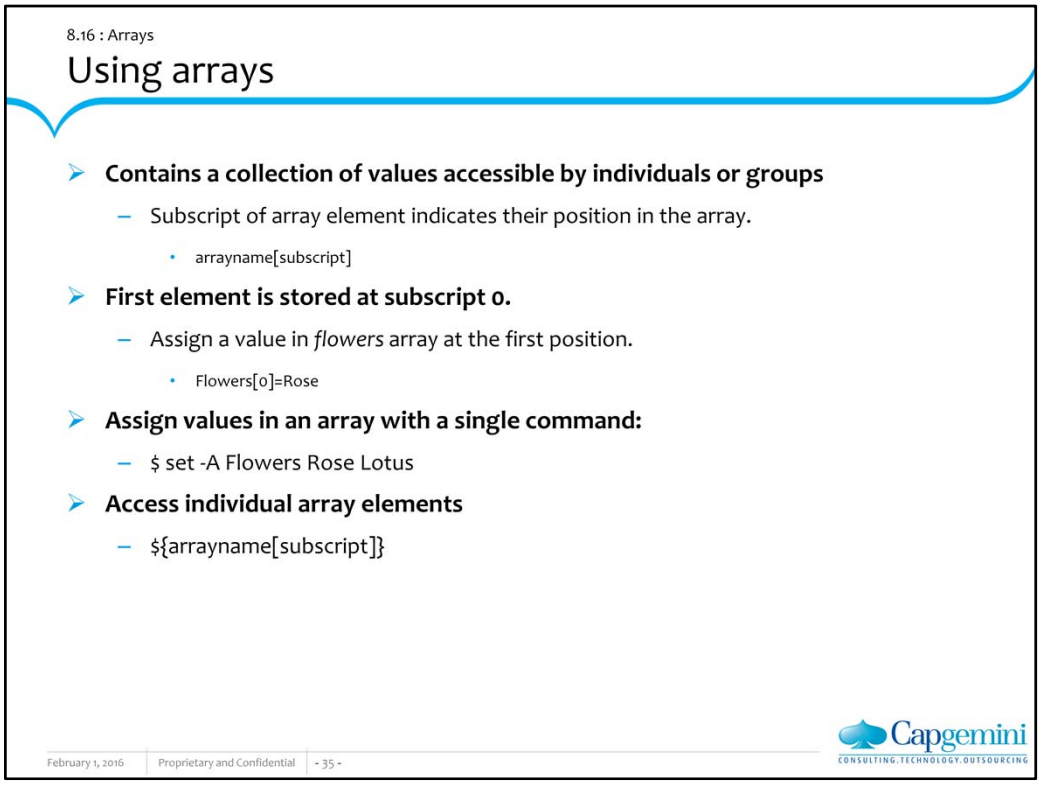


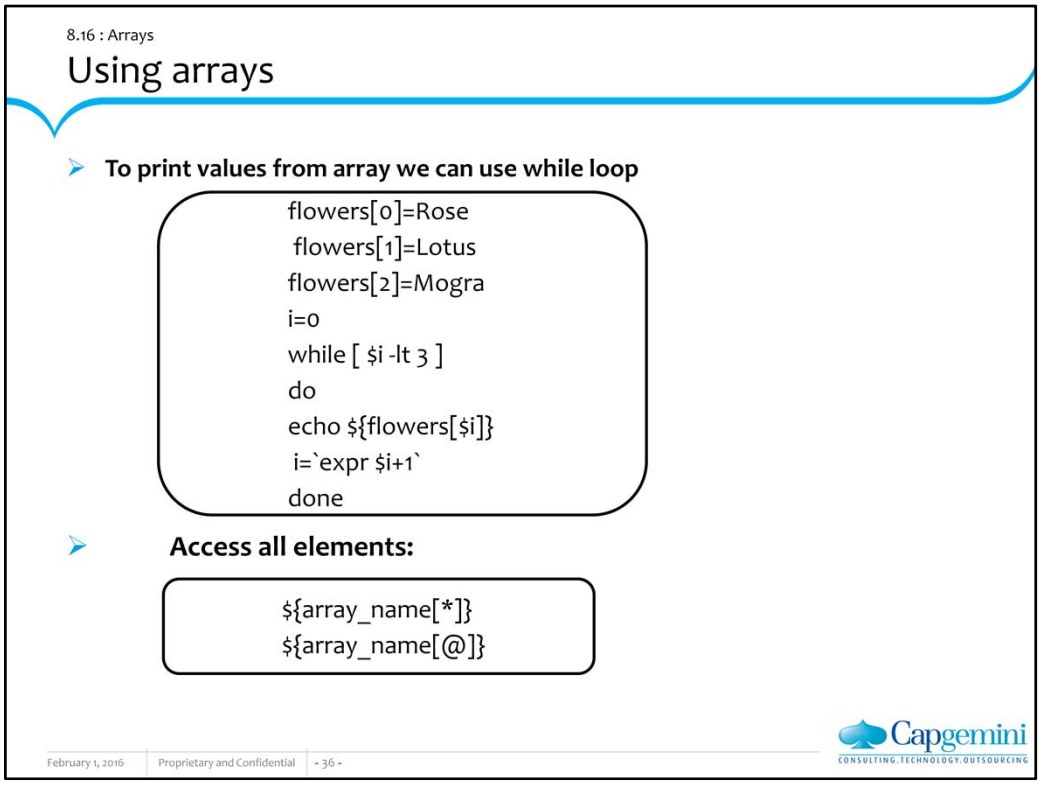

In the above example, *Myfunction* is called in the *if* statement with message "enter the number". This message is passed as three arguments.

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 : 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]}
February 1, 2016
Proprietary and Confidential
- 35 -
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING




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

## 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: