Shell Scripting
An Introduction

CDAC Mumbai

What is Shell?

- Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is pass to kernel
- It reads, interprets and process the command
- Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Introduction to shell scripts

- All shells has a built in language
- A user can write a script using the language and the shell will execute that script
- The script does not require compilation or linking
- On a UNIX system there are various shells
- A shell is an environment in which we can run our commands, programs, and shell scripts.
- There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions

Introduction to shell scripts

- Shell programs can be used for various tasks
  - Customizing the work environment
  - Automating some routine tasks
    - Backing up all the files
    - Producing the sales report every month
  - Executing system procedures
    - Shutting down
    - formatting the disk etc…

The shell of Linux

- Linux has a variety of different shells:
  - Bourne shell (sh), C shell (csh), Korn shell (ksh), Bourne Again shell (bash).
- Certainly the most popular shell is “bash” from GNU. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh).
- Additionally Bash, has the advantage that the source code is freely available.

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	See the man page. Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

What is shell Script ?

- shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is know as ***shell script***.
- Shell script defined as:
  "*Shell Script is* ***series of command*** *written* ***in plain text file***

Programming or Scripting ?

- bash is not only an excellent command line shell, but a scripting language in itself. Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.

- Difference between programming and scripting languages:

  - Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.

  - A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction. Interpreted programs are generally slower than compiled programs. The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Other examples of scripting languages are Perl, Lisp, and Tcl.

REDIRECTION

- Default (standard) input, output and error files
  - stdin (keyboard), stdout (screen), stderr (screen)
- Redirecting Standard Output (**>** or **>>**)
  - $ **cat file1 file2 > file3**
    - file3 is created if not there, overwritten if there
  - $ **cat file1 file2 >> file3**
    - file3 is appended to if it is there
- Redirecting Standard Error (**>&**)
  - $ **(cat myfile > yourfile) >& yourerrorfile**
  - $ **cat myfile >& yourfile**

  {generally redirect stdout & stderr to same place}
- Redirecting Standard Input (**<**)
  - $ **cat < oldfile > newfile**
  - **$ tr a-z A-Z < file1 > file2** {an example of filter}
- /dev/null – A virtual file that is always empty
  - $ **cp /dev/null myfile** {get an empty file}
  - $ **(ls -l > recordfile) >& /dev/null** {discard errors}

Shell Commands

- To find all available shells in your system
  - $ **cat /etc/shells**
- To find your current shell type following command
  - $ **echo $SHELL**
- To change to another shell
  - $ **/bin/sh**
  - $ **/bin/bash**
  - $ **/bin/csh**
  - $ **/bin/ksh**
- Shell prompt and secondary prompt defined by $PS1 ($) and $PS2 (>)
  - $ **echo $PS1**
  - $ **export PS1=%** {bash}
  - **$ setenv PS1 %** {csh}
- To customize the prompt string

\d : the date (e.g., "Tue May 26")	\t : the current time (HH:MM:SS)
\h : the hostname	\u : the current user name
\w : the working dir, (~ is $HOME)	\! : the history no. of this command

```
$ PS1="[\d \t \u@\h:\w ] $ ”
[Sat Oct 12 14:24:12 divesd@server:~ ] $      {~ stands for $HOME}
```

ECHO COMMAND

- echo [options] [string, variables...]
  Displays text or variables value on screen.
  Options
  -n Do not output the trailing new line.
  -e Enable interpretation of the following backslash escaped characters in the strings:
  \a alert (bell)
  \b backspace
  \c suppress trailing new line
  \n new line
  \r carriage return
  \t horizontal tab
  \\ backslash
- For e.g. **$ echo -e "An apple a day keeps away \a\t\tdoctor\n"**

The first bash program

- There are two major text editors in Linux:
  - vi, emacs (or xemacs).

- So open up a text editor; for example:

  ```
  $ vi &
  ```

  and type the following inside it:

  ```
  #!/bin/bash
  echo “Hello World”
  ```

- The first line tells Linux to use the bash interpreter to run this script. We call it hello.sh. Then, make the script executable:

  ```
  $ chmod 700 hello.sh
  $ ./hello.sh
  Hello World
  ```

The second bash program

- We write a program that copies all files into a directory, and then deletes the directory along with its contents. This can be done with the following commands:

```
$ mkdir trash
$ cp * trash
$ rm -rf trash

$ rmdir trash
```

- Instead of having to type all that interactively on the shell, write a shell program instead:

```
$ cat trash.sh
#!/bin/bash
# this script deletes some files
mkdir trash
cp * trash
rm -rf trash
mkdir trash
echo “Deleted all files!”
```

VARIABLES

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.

- We have no need to declare a variable, just assigning a value to its reference will create it.

- Example

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

- Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.

Warning

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
count=0
count=Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it, so it is recommended to use a variable for only a single TYPE of data in a script.

- \ is the bash escape character and it preserves the literal value of the next character that follows.

```
$ ls \*
ls: *: No such file or directory
```

PIPES

- A pipe is a way to connect the output of one program to the input of another program without any temporary file
- Pipe Defined as:
  "*A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line.*"
- *Syntax:*
  command1 | command2

EXAMPLE:

Command using Pipes	Meaning or Use of Pipes
$ ls | more	Output of ls command is given as input to more command So that output is printed one screen full page at a time.
$ who | sort	Output of who command is given as input to sort command So that it will print sorted list of users
$ who | sort > user_list	Same as above except output of sort is send to (redirected) user_list file
$ who | wc -l	Output of who command is given as input to wc command So that it will number of user who logon to system
$ ls -l | wc -l	Output of ls command is given as input to wc command So that it will print number of files in current directory.
$ who | grep raju	Output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see particular user is logon or not)

Single and Double Quote

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.

- Using double quotes to show a string of characters will allow any variables in the quotes to be resolved

```
$ var=“test string”
$ newvar=“Value of var is $var”
$ echo $newvar
Value of var is test string
```

- Using single quotes to show a string of characters will not allow variable resolution

```
$ var=’test string’
$ newvar=’Value of var is $var’
$ echo $newvar
Value of var is $var
```

Environmental Variables

- There are two types of variables:
  - Local variables
  - Environmental variables
- Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance:

```
$ echo $SHELL
/bin/bash
$ echo $PATH
/usr/X11/bin: /usr/local/bin: /bin: /usr/bin
```

- Environmental variables are defined in /etc/profile, /etc/profile.d/ and ~/.bash_profile. These files are the initialization files and they are read when bash shell is invoked.

Environment Variables

- HOME: The default argument (home directory) for cd.
- PATH: The search path for commands. It is a colon-separated list of directories that are searched when you type a command.

- Usually, we type in the commands in the following way:

```
$ ./command
```

- By setting PATH=$PATH:. our working directory is included in the search path for commands, and we simply type:

```
$ command
```

- If we type in

```
$ mkdir ~/bin
```

- and we include the following lines in the ~/.bash_profile:

```
PATH=$PATH:$HOME/bin
export PATH
```

- we obtain that the directory /home/userid/bin is included in the search path for commands.

System variables or environment variable

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

Environmental Variables

- LOGNAME: contains the user name
- HOSTNAME: contains the computer name.

- PS1: sequence of characters shown before the prompt

\t	hour
\d	date
\w	current directory
\W	last part of the current directory
\u	user name
\$	prompt character

Example:

```
[userid@homelinux userid]$ PS1='hi \u *'
hi userid* _
```

Exercise ==> Design your own new prompt.

READ COMMAND

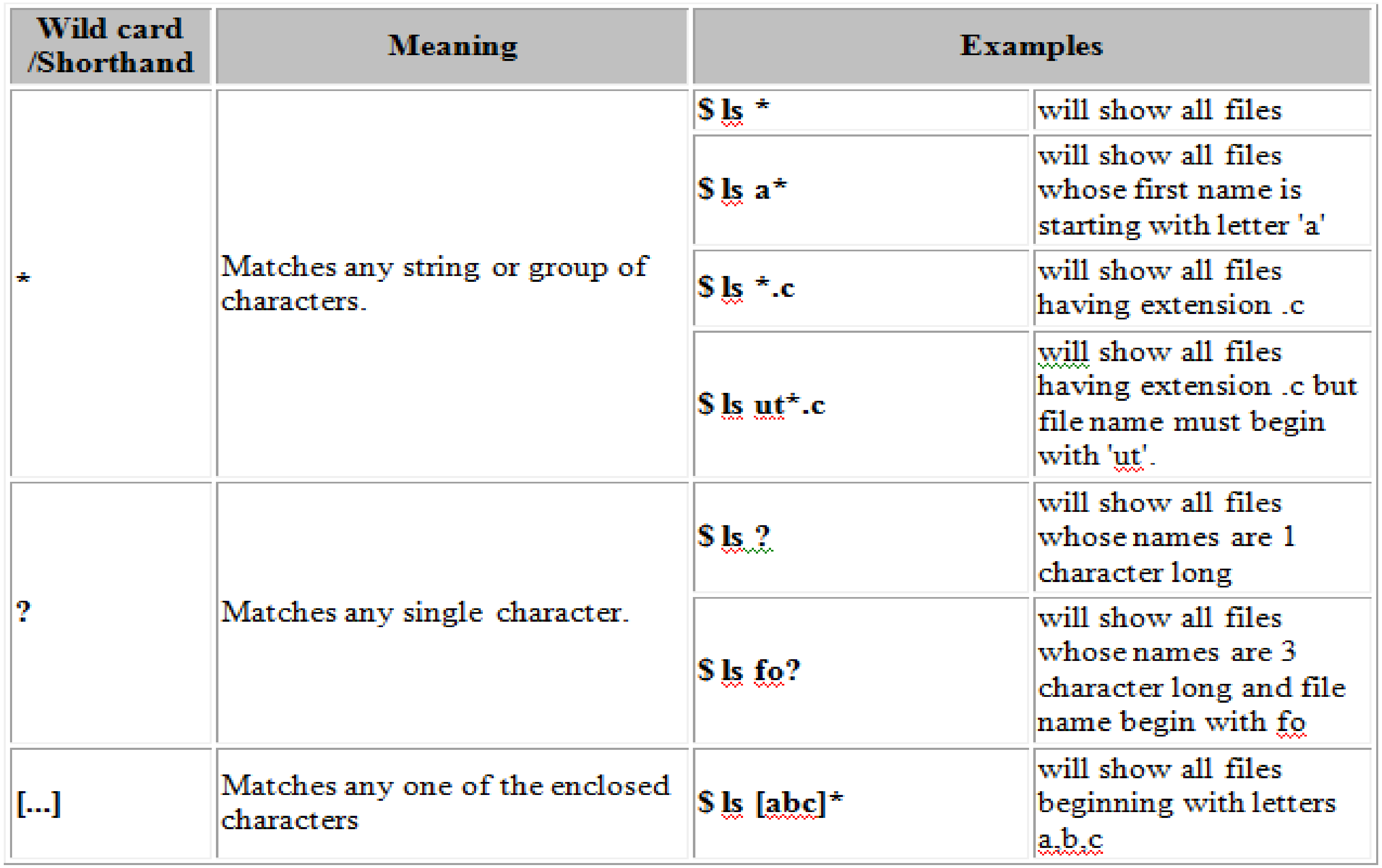
- The read command allows you to prompt for input and store it in a variable.

- Example:

```
#!/bin/bash
echo -n “Enter name of file to delete: ”
read file
echo “Type 'y' to remove it, 'n' to change your mind ... ”
rm -i $file
echo "That was YOUR decision!”
```

- Line 2 prompts for a string that is read in line 3. Line 4 uses the interactive remove (rm -i) to ask the user for confirmation.

Wild cards (Filename Shorthand or meta Characters)



Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	$ ls *	will show all files
		$ ls a*	will show all files whose first name is starting with letter 'a'
		$ ls *.c	will show all files having extension .c
		$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	$ ls ?	will show all files whose names are 1 character long
		$ ls fo?	will show all files whose names are 3 character long and file name begin with fo
[...]	Matches any one of the enclosed characters	$ ls [abc]*	will show all files beginning with letters a,b,c

ARITHMETIC EVALUATION

- The let statement can be used to do mathematical functions:

```
$ let X=10+2*7
$ echo $X
        24
$ let Y=X+2*4
$ echo $Y
        32
```

- An arithmetic expression can be evaluated by $[expression] or $((expression))

```
$ echo “$((123+20))”
        143
$ VALORE=$[123+20]
$ echo “$[123*$VALORE]”
        17589
```

Arithmetic Evaluation

- Available operators: +, -, /, *, %

- Example

```
$ cat arithmetic.sh
 #!/bin/bash
 echo -n “Enter the first number: ”; read x
 echo -n “Enter the second number: ”; read y
 add=$(($x + $y))
 sub=$(($x - $y))
 mul=$(($x * $y))
 div=$(($x / $y))
 mod=$(($x % $y))
 # print out the answers:
 echo “Sum: $add”
 echo “Difference: $sub”
 echo “Product: $mul”
 echo “Quotient: $div”
 echo “Remainder: $mod”
```

Command Substitution

- The backquote "`" is different from the single quote "´". It is used for command substitution: `command`

```
$ LIST =`ls`
$ echo $LIST
hello.sh read.sh

$ PS1 ="`pwd`>"
/home/userid/work> _
```

- We can perform the command substitution by means of $(command)

```
$ LIST=$(ls)
$ echo $LIST
hello.sh read.sh

$ rm $( find / -name "*.tmp" )

$ cat > backup.sh
#!/bin/bash
BCKUP=/home/userid/backup-$(date +%d-%m-%y).tar.gz
tar -czf $BCKUP $HOME
```

Command Line Arguments

- **$ ls grate_stories_of**
  It will print message something like - *grate_stories_of: No such file or directory*.
- **ls** is the name of an *actual command* and shell executed this command when you type command at shell prompt. Now it creates one more question **What are commands?** What happened when you type *$ ls grate_stories_of* ?
- The first word on command line is, **ls** - is name of the command to be executed. Everything else on command line is taken *as arguments to this command*. For e.g.
  $ tail +10 myf
  Name of command is **tail**, and the arguments are **+10** and **myf**.

- **$ ls grate_stories_of**
  It will print message something like - *grate_stories_of: No such file or directory*.
- **ls** is the name of an *actual command* and shell executed this command when you type command at shell prompt. Now it creates one more question **What are commands?** What happened when you type *$ ls grate_stories_of* ?
- The first word on command line is, **ls** - is name of the command to be executed. Everything else on command line is taken *as arguments to this command*. For e.g.
  $ tail +10 myf
  Name of command is **tail**, and the arguments are **+10** and **myf**.

Exercise

Try to determine command and arguments from following commands

```
$ ls foo
$ cp y y.bak
$ mv y.bak y.okay
$ tail -10 myf
$ mail raj
$ sort -r -n myf
$ date
$ clear
```

Command	No. of argument to this command (i.e $#)	Actual Argument
ls	1	foo
cp	2	y and y.bak
mv	2	y.bak and y.okay
tail	2	-10 and myf
mail	1	raj
sort	3	-r, -n, and myf
date	0	
clear	0	

Conditional Statements

- Conditionals let us decide whether to perform an action or not, this decision is taken by evaluating an expression. The most basic form is:

```
if [ expression ];
then
        statements
elif [ expression ];
then
        statements
else
        statements
fi
```

- the elif (else if) and else sections are optional

- Put spaces after [and before], and around the operators and operands.

EXPRESSIONS

- An expression can be: String comparison, Numeric comparison, File operators and Logical operators and it is represented by [expression]:

- String Comparisons:

=	compare if two strings are equal
!=	compare if two strings are not equal
-n	evaluate if string length is greater than zero
-z	evaluate if string length is equal to zero

- Examples:

[s1 = s2]	(true if s1 same as s2, else false)
[s1 != s2]	(true if s1 not same as s2, else false)
[s1]	(true if s1 is not empty, else false)
[-n s1]	(true if s1 has a length greater then 0, else false)
[-z s2]	(true if s2 has a length of 0, otherwise false)

Expressions

- Number Comparisons:

-eq	compare if two numbers are equal
-ge	compare if one number is greater than or equal to a number
-le	compare if one number is less than or equal to a number
-ne	compare if two numbers are not equal
-gt	compare if one number is greater than another number
-lt	compare if one number is less than another number

- Examples:

[n1 -eq n2]	(true if n1 same as n2, else false)
[n1 -ge n2]	(true if n1greater then or equal to n2, else false)
[n1 -le n2]	(true if n1 less then or equal to n2, else false)
[n1 -ne n2]	(true if n1 is not same as n2, else false)
[n1 -gt n2]	(true if n1 greater then n2, else false)
[n1 -lt n2]	(true if n1 less then n2, else false)

Examples

```
$ cat user.sh
 #!/bin/bash
     echo -n “Enter your login name: "
     read name
     if [ “$name” = “$USER” ];
     then
          echo “Hello, $name. How are you today ?”
     else
          echo “You are not $USER, so who are you ?”
     fi

$ cat number.sh
#!/bin/bash
     echo -n “Enter a number 1 < x < 10: "
     read num
     if [ “$num” -lt 10 ];              then
          if [ “$num” -gt 1 ]; then
                         echo “$num*$num=$(($num*$num))”
          else
                         echo “Wrong insertion !”
          fi
     else
          echo “Wrong insertion !”
     fi
```

EXPRESSIONS

- Files operators:

-d	check if path given is a directory
-f	check if path given is a file
-e	check if file name exists
-r	check if read permission is set for file or directory
-s	check if a file has a length greater than 0
-w	check if write permission is set for a file or directory
-x	check if execute permission is set for a file or directory

- Examples:

[-d fname]	(true if fname is a directory, otherwise false)
[-f fname]	(true if fname is a file, otherwise false)
[-e fname]	(true if fname exists, otherwise false)
[-s fname]	(true if fname length is greater then 0, else false)
[-r fname]	(true if fname has the read permission, else false)
[-w fname]	(true if fname has the write permission, else false)
[-x fname]	(true if fname has the execute permission, else false)

EXAMPLES

```
#!/bin/bash
    if [ -f /etc/fstab ];
    then
        cp /etc/fstab .
        echo “Done.”
    else
        echo “This file does not exist.”
        exit 1
    fi
```

Exercise.

- Write a shell script which:
  - accepts a file name
  - checks if file exists
  - if file exists, copy the file to the same name + .bak + the current date (if the backup file already exists ask if you want to replace it).
- When done you should have the original file and one with a .bak at the end.

EXPRESSIONS

- Logical operators:

!	negate (NOT) a logical expression
-a	logically AND two logical expressions
-o	logically OR two logical expressions

Example:

```
#!/bin/bash
    echo -n “Enter a number 1 < x < 10:”
    read num
    if [ “$num” -gt 1 –a “$num” -lt 10 ];
    then
        echo “$num*$num=$(($num*$num))”
    else
        echo “Wrong insertion !”
    fi
```

EXPRESSIONS

- Logical operators:

&&	logically AND two logical expressions
||	logically OR two logical expressions

Example:

```
#!/bin/bash
    echo -n "Enter a number 1 < x < 10: "
    read num
    if [ “$number” -gt 1 ] && [ “$number” -lt 10 ];
    then
        echo “$num*$num=$(($num*$num))”
    else
        echo “Wrong insertion !”
    fi
```

Example

```
$ cat iftrue.sh
    #!/bin/bash
    echo "Enter a path: "; read x
    if cd $x; then
        echo "I am in $x and it contains"; ls
    else
        echo "The directory $x does not exist";
        exit 1
    fi

$ iftrue.sh
Enter a path: /home
userid anotherid …

$ iftrue.sh
Enter a path: blah
The directory blah does not exist
```

SHELL PARAMETERS

- Positional parameters are assigned from the shell's argument when it is invoked. Positional parameter "N" may be referenced as "${N}", or as "$N" when "N" consists of a single digit.

- Special parameters

  $# is the number of parameters passed
  $0 returns the name of the shell script running as well as its location in the file system
  $* gives a single word containing all the parameters passed to the script
  $@ gives an array of words containing all the parameters passed to the script

```
$ cat sparameters.sh
 #!/bin/bash
 echo "$#; $0; $1; $2; $*; $@"
 $ sparameters.sh arg1 arg2
 2; ./sparameters.sh; arg1; arg2; arg1 arg2; arg1 arg2
```

Example (trash.sh)

```
$ cat trash.sh
  #!/bin/bash
  if [ $# -eq 1 ];
  then
     if [ ! –d “$HOME/trash” ];
     then
              mkdir “$HOME/trash”
     fi
     mv $1 “$HOME/trash”
  else
     echo “Use: $0 filename”
     exit 1
  fi
```

Case Statement

- Used to execute statements based on specific values. Often used in place of an if statement if there are a large number of conditions.

- Value used can be an expression
- each set of statements must be ended by a pair of semicolons;
- a *) is used to accept any value not matched with list of values

```
case $var in
   val1)
      statements;;
   val2)
      statements;;
   *)
      statements;;
   esac
```

Example

```
$ cat case.sh
#!/bin/bash
    echo -n “Enter a number 1 < x < 10: ”
  read x
    case $x in
        1) echo “Value of x is 1.”;;
        2) echo “Value of x is 2.”;;
        3) echo “Value of x is 3.”;;
        4) echo “Value of x is 4.”;;
        5) echo “Value of x is 5.”;;
        6) echo “Value of x is 6.”;;
        7) echo “Value of x is 7.”;;
        8) echo “Value of x is 8.”;;
        9) echo “Value of x is 9.”;;
        0 | 10) echo “wrong number.”;;
        *) echo “Unrecognized value.”;;
    esac
```

Iteration Statements

- The for structure is used when you are looping through a range of variables.

```
for var in list
    do
      statements
    done
```

- statements are executed with var set to each value in the list.

- Example

```
#!/bin/bash
    let sum=0
    for num in 1 2 3 4 5
      do
        let “sum = $sum + $num”
      done
    echo $sum
```

ITERATION STATEMENTS

```
#!/bin/bash
    for x in paper pencil pen
  do
      echo "The value of variable x is: $x"
      sleep 1
    done
```

- if the list part is left off, var is set to each parameter passed to the script ($1, $2, $3,…)

```
$ cat for1.sh
    #!/bin/bash
    for x
    do
      echo "The value of variable x is: $x"
      sleep 1
    done

$ for1.sh arg1 arg2
    The value of variable x is: arg1
    The value of variable x is: arg2
```

EXAMPLE (OLD.SH)

```
$ cat old.sh
#!/bin/bash
# Move the command line arg files to old directory.

 if [ $# -eq 0 ]          #check for command line arguments
 then
   echo “Usage: $0 file …”
 exit 1
 fi

 if [ ! –d “$HOME/old” ]
 then
   mkdir “$HOME/old”
 fi

 echo The following files will be saved in the old directory:
 echo $*
 for file in $*           #loop through all command line arguments
 do
   mv $file “$HOME/old/”
   chmod 400 “$HOME/old/$file”
 done
 ls -l “$HOME/old”
```

EXAMPLE (ARGS.SH)

```
$ cat args.sh
#!/bin/bash
# Invoke this script with several arguments: “one two three“
if [ ! -n “$1” ]; then
    echo “Usage: $0 arg1 arg2 ..." ; exit 1
fi
echo ; index=1 ;
echo “Listing args with \”\$*\”:”
for arg in “$*” ;
do
    echo “Arg $index = $arg”
    let “index+=1”              # increase variable index by one
done
echo “Entire arg list seen as single word.”
echo ; index=1 ;
echo “Listing args with \”\$@\”:”
for arg in “$@” ; do
    echo “Arg $index = $arg”
    let “index+=1”
done
echo “Arg list seen as separate words.” ; exit 0
```

Using Arrays with Loops

- In the bash shell, we may use arrays. The simplest way to create one is using one of the two subscripts:

```
pet[0]=dog
pet[1]=cat
pet[2]=fish
pet=(dog cat fish)
```

- We may have up to 1024 elements. To extract a value, type ${arrayname[i]}

```
$ echo ${pet[0]}
dog
```

- To extract all the elements, use an asterisk as:

```
echo ${arrayname[*]}
```

- We can combine arrays with loops using a for loop:

```
for x in ${arrayname[*]}
      do
            ...
      done
```

A C-like for loop

- An alternative form of the for structure is

```
for (( EXPR1 ; EXPR2 ; EXPR3 ))
do
        statements
done
```

- First, the arithmetic expression EXPR1 is evaluated. EXPR2 is then evaluated repeatedly until it evaluates to 0. Each time EXPR2 is evaluates to a non-zero value, statements are executed and EXPR3 is evaluated.

```
$ cat for2.sh
#!/bin/bash
echo –n “Enter a number: ”; read x
let sum=0
for (( i=1 ; $i<$x ; i=$i+1 )) ;  do
  let “sum = $sum + $i”
done
echo   “the sum of the first $x numbers is: $sum”
```

DEBUGGING

- Bash provides two options which will give useful information for debugging

-x : displays each line of the script with variable substitution and before execution

-v : displays each line of the script as typed before execution

- Usage:

  #!/bin/bash –v or #!/bin/bash –x or #!/bin/bash –xv

```
$ cat for3.sh
    #!/bin/bash –x
  echo –n "Enter a number: "; read x
    let sum=0
    for (( i=1 ; $i<$x ; i=$i+1 )) ; do
      let "sum = $sum + $i"
    done
    echo "the sum of the first $x numbers is: $sum"
```

DEBUGGING

```
$ for3.sh
+ echo –n ‘Enter a number: ’
Enter a number: + read x
3
+ let sum=0
+ (( i=0 ))
+ (( 0<=3 ))
+ let ‘sum = 0 + 0’
+ (( i=0+1 ))
+ (( 1<=3 ))
+ let ‘sum = 0 + 1’
+ (( i=1+1 ))
+ (( 2<=3 ))
+ let ‘sum = 1 + 2’
+ (( i=2+1 ))
+ (( 3<=3 ))
+ let ‘sum = 3 + 3’
+ (( i=3+1 ))
+ (( 4<=3 ))
+ echo ‘the sum of the first 3 numbers is: 6’
the sum of the first 3 numbers is: 6
```

While Statement

- The while structure is a looping structure. Used to execute a set of commands while a specified condition is true. The loop terminates as soon as the condition becomes false. If condition never becomes false, loop will never exit.

```
while expression
do
        statements
done
```

```
$ cat while.sh
    #!/bin/bash
    echo –n "Enter a number: "; read x
    let sum=0; let i=1
    while [ $i –le $x ]; do
      let "sum = $sum + $i"
          i=$i+1
    done
  echo "the sum of the first $x numbers is: $sum"
```

EXAMPLE (MENU.SH)

```
$ cat menu.sh
#!/bin/bash
     clear ; loop=y
     while [ "$loop" = y ] ;
     do
       echo "Menu";  echo "===="
       echo "D: print the date"
       echo "W: print the users who are currently log on."
       echo "P: print the working directory"
       echo "Q: quit."
       echo
       read –s choice                        # silent mode: no echo to terminal
       case $choice in
          D | d) date ;;
          W | w) who ;;
          P | p) pwd ;;
          Q | q) loop=n ;;
          *) echo "Illegal choice." ;;
       esac
       echo
     done
```

Continue Statement

- The continue command causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

```
 $ cat continue.sh
#!/bin/bash
     LIMIT=19
     echo
     echo “Printing Numbers 1 through 20 (but not 3 and 11)”
     a=0
     while [ $a -le “$LIMIT” ]; do
       a=$(($a+1))
       if [ “$a” -eq 3 ] | | [ “$a” -eq 11 ]
       then
          continue
       fi
       echo -n “$a ”
     done
```

Break Statement

- The break command terminates the loop (breaks out of it).

```
 $ cat break.sh
#!/bin/bash
     LIMIT=19
     echo
     echo “Printing Numbers 1 through 20, but something happens after 2 … ”
     a=0
     while [ $a -le “$LIMIT” ]
     do
       a=$(($a+1))
       if [ “$a” -gt 2 ]
       then
      break
       fi
    echo -n “$a ”
  done
  echo; echo; echo
  exit 0
```

Until Statement

- The until structure is very similar to the while structure. The until structure loops until the condition is true. So basically it is “until this condition is true, do this”.

```
until [expression]
do
        statements
done
```

```
$ cat countdown.sh
    #!/bin/bash
    echo “Enter a number: ”; read x
    echo ; echo Count Down
    until [ “$x” -le 0 ]; do
   echo $x
   x=$(($x –1))
   sleep 1
  done
echo ; echo GO !
```

FUNCTIONS

- Functions make scripts easier to maintain. Basically it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish.

```
#!/bin/bash
    hello()
    {
    echo “You are in function hello()”
    }

    echo “Calling function hello()…”
    hello
    echo “You are now out of function hello()”
```

- In the above, we called the hello() function by name by using the line: hello . When this line is executed, bash searches the script for the line hello(). It finds it right at the top, and executes its contents.

EXAMPLE (FUNCTION.SH)

```
$ cat function.sh
#!/bin/bash
function check() {
if [ -e "/home/$1" ]
then
  return 0
else
  return 1
fi
}
echo “Enter the name of the file: ” ; read x
if check $x
then
  echo “$x exists !”
else
  echo “$x does not exists !”
fi.
```