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:

```
#!/bin/bash

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
```

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
}
```

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/bash

# Define your function here
Hello () {
   echo "Hello World"
 }

# Invoke your function
Hello
```

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

```
$./test
Hello World
```

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 *Zara* and *Ali* 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 Zara Ali
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
```

Returning Values from Functions:

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the **return** command whose syntax is as follows:

```
return code
```

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

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 Zara Ali

# Capture value returnd by last command
ret=$?

echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
Return value is 10
```

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...
```

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:

```
$. 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:

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

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.

```
$unset -f function_name
```

User Interface

Good program/shell script must interact with users. Even you can create menus to interact with user, first show menu option, then ask user to choose menu item, and take appropriate action according to selected menu item, this technique is show in following script:

```
#
# Script to create simple menus and take action according to that selected
# menu item
#
while :
 do
    clear
    echo "-------------------------------------"
    echo " Main Menu "
    echo "-------------------------------------"
    echo "[1] Show Todays date/time"
    echo "[2] Show files in current directory"
    echo "[3] Show calendar"
    echo "[4] Start editor to write letters"
    echo "[5] Exit/Stop"
    echo "======================="
    echo -n "Enter your menu choice [1-5]: "
    read yourch
    case $yourch in
      1) echo "Today is `date` , press a key. . ." ; read ;;
      2) echo "Files in `pwd`" ; ls -l ; echo "Press a key. . ." ; read ;;
      3) cal ; echo "Press a key. . ." ; read ;;
      4) nano ;;
      5) exit 0 ;;
      *) echo "Opps!!! Please select choice 1,2,3,4, or 5";
         echo "Press a key. . ." ; read ;;
 esac
done
```

Above all statement explained in following table:

Statement	Explanation
while :	Start infinite loop, this loop will only break if you select 5 (i.e. Exit/Stop menu item) as your menu choice
do	Start loop
clear	Clear the screen, each and every time
echo "-------------------------------------" **echo " Main Menu "** **echo "-------------------------------------"** **echo "[1] Show Todays date/time"** **echo "[2] Show files in current directory"** **echo "[3] Show calendar"** **echo "[4] Start editor to write letters"** **echo "[5] Exit/Stop"** **echo "======================="**	Show menu on screen with menu items
echo -n "Enter your menu choice [1-5]: "	Ask user to enter menu item number
read yourch	Read menu item number from user

**case $yourch in
1) echo "Today is `date` , press a key. . ." ; read ;;
2) echo "Files in `pwd`" ;
            ls -l ;
   echo "Press a key. . ." ;
            read ;;
3) cal ;
   echo "Press a key. . ." ;
   read ;;
4) nano ;;
5) exit 0 ;;
*) echo "Opps!!! Please select choice 1,2,3,4, or 5";
    echo "Press a key. . ." ; read ;;
  esac**
	Take appropriate action according to selected menu item, If menu item is not between 1 - 5, then show error and ask user to input number between 1-5 again

done	Stop loop , if menu item number is 5 (i.e. Exit/Stop)

User interface usually includes, menus, different type of boxes like info box, message box, Input box etc.