**Shell Scripting**

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

* The Bourne Shell
* The C Shell
* The Korn Shell
* The GNU Bourne-Again Shell (bash)

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

$date

Thu Jun 25 08:30:19 MST 2009

## Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the****#****symbol is called a hash, and the ! symbol is called a bang*.

To create a script containing these commands, you put the shebang line first and then add the commands −

#!/bin/bash

pwd

ls

## Shell Comments

You can put your comments in your script as follows −

#!/bin/bash

# Author : Zara Ali

# Script follows here:

pwd

ls

Save the above content and make the script executable −

$chmod +x test.sh

The shell script is now ready to be executed −

$./test.sh

Upon execution, you will receive the following result −

/home/amrood

index.htm unix-basic\_utilities.htm unix-directories.htm

test.sh unix-communication.htm unix-environment.htm

**Note** − To execute a program available in the current directory, use **./program\_name**

## Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Author : Zara Ali

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

What is your name?

Zara Ali

Hello, Zara Ali

$

# Read a number in shell script

In Shell Script, we can read number using **read** keyword.

## Syntax

read name

## Example

#read number in shell script

echo "Enter a Number"

read num

echo "Number = $num"

## Pass arguments from command line - shell script

To pass arguments from the command line, add the argument values after the file name while executing the script.

Like,

**sh file\_name.sh arg1 arg2**

## Add two variables using command line argument in shell script

#shell script to add two numbers

#using command line arguments

#$1 refers to the first argument

#$2 refers the second argument and so on

sum=**$((**$1 + $2**))**

echo "Sum = $sum"

#### Run the script

**sh add.sh 10 20**

Output

Sum = 30

**sh add.sh 100 20**

Output

Sum = 120

## Add two variables without using expr in shell script

#shell program to add two variables

var1=10

var2=20

sum=**$((**$var1 + $var2**))**

echo $sum

#### Output

30

## Add two variables using expr in shell script

#shell program to add two variables

var1=10

var2=20

sum=`expr $var1 + $var2`

echo $sum

#### Output

30

## Multiplication of two numbers without using expr

#shell program to multiply two numbers

num1=10

num2=20

ans=**$((**num1 \* num2**))**

echo $ans

#### Output

200

## Multiplication of two numbers using expr in shell script

In shell, **\*** represents all files in the current directory.

So, in order to use \* as a multiplication operator, we should escape it like **\\***.

If we directly use \* in expr, we will get error message.

#shell program to multiply two numbers using expr

num1=10

num2=20

ans=`expr $num1 **\\*** $num2`

echo $ans

#### Output

200

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as **!**, **\***, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows −

variable\_name=variable\_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"

VAR2=100

Accessing Values

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

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

[Live Demo](http://tpcg.io/AP7zgT)

#!/bin/sh

NAME="Zara Ali"

echo $NAME

The above script will produce the following value −

Zara Ali

Read-only Variables

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

For example, the following script generates an error while trying to change the value of NAME −

[Live Demo](http://tpcg.io/tawT1C)

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

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

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

**Special Variables**

For example, the **$** character represents the process ID number, or PID, of the current shell −

$echo $$

The above command writes the PID of the current shell −

29949

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

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  The process number of the last background command. |

Command-Line Arguments

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

Following script uses various special variables related to the command line −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

Special Parameters $\* and $@

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

Both the parameters specify the command-line arguments. However, the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $\* or $@ special parameters −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

Here is a sample run for the above script −

$./test.sh Zara Ali 10 Years Old

Zara

Ali

10

Years

Old

**Note** − Here **do...done** is a kind of loop that will be covered in a subsequent tutorial.

Exit Status

The **$?** variable represents the exit status of the previous command.

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

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

Following is the example of successful command −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

$echo $?

0

$

Defining Array Values

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Zara"

NAME02="Qadir"

NAME03="Mahnaz"

NAME04="Ayan"

NAME05="Daisy"

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

array\_name[index]=value

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

As an example, the following commands −

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

If you are using the **ksh** shell, here is the syntax of array initialization −

set -A array\_name value1 value2 ... valuen

If you are using the **bash** shell, here is the syntax of array initialization −

array\_name=(value1 ... valuen)

Accessing Array Values

After you have set any array variable, you access it as follows −

${array\_name[index]}

Here *array\_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

[Live Demo](http://tpcg.io/AMsECl)

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

The above example will generate the following result −

$./test.sh

First Index: Zara

Second Index: Qadir

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

Here **array\_name** is the name of the array you are interested in. Following example will help you understand the concept −

[Live Demo](http://tpcg.io/r8Dol0)

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Method: ${NAME[\*]}"

echo "Second Method: ${NAME[@]}"

The above example will generate the following result −

$./test.sh

First Method: Zara Qadir Mahnaz Ayan Daisy

Second Method: Zara Qadir Mahnaz Ayan Daisy

**Operators**

We will now discuss the following operators −

* Arithmetic Operators
* Relational Operators
* Boolean Operators
* String Operators
* File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers −

[Live Demo](http://tpcg.io/zURE2C)

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

* There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* The complete expression should be enclosed between **‘ ‘**, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/unix/unix-arithmetic-operators.htm)

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

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

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

Assume variable **a** holds 10 and variable **b** holds 20 then −

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

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[ $a <= $b ]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

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

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

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

**Decision Making**

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

* The **if...else** statement
* The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

* [if...fi statement](https://www.tutorialspoint.com/unix/if-fi-statement.htm)
* [if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)
* [if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm)

Most of the if statements check relations using relational operators discussed in the previous chapter.

The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

* [case...esac statement](https://www.tutorialspoint.com/unix/case-esac-statement.htm)

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

if [ expression ]

then

Statement(s) to be executed if expression is true

fi

The *Shell expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

Example

[Live Demo](http://tpcg.io/EGdFBg)

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

The above script will generate the following result −

a is not equal to b

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

if [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

[Live Demo](http://tpcg.io/FqxOwQ)

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

Upon execution, you will receive the following result −

a is not equal to b

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

if [ expression 1 ]

then

Statement(s) to be executed if expression 1 is true

elif [ expression 2 ]

then

Statement(s) to be executed if expression 2 is true

elif [ expression 3 ]

then

Statement(s) to be executed if expression 3 is true

else

Statement(s) to be executed if no expression is true

fi

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

[Live Demo](http://tpcg.io/i5Hw31)

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

Upon execution, you will receive the following result −

a is less than b

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

case word in

pattern1)

Statement(s) to be executed if pattern1 matches

;;

pattern2)

Statement(s) to be executed if pattern2 matches

;;

pattern3)

Statement(s) to be executed if pattern3 matches

;;

\*)

Default condition to be executed

;;

esac

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

[Live Demo](http://tpcg.io/fuk3nK)

#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in

"apple") echo "Apple pie is quite tasty." ;;

"banana") echo "I like banana nut bread." ;;

"kiwi") echo "New Zealand is famous for kiwi.";;

esac

Upon execution, you will receive the following result −

New Zealand is famous for kiwi.

A good use for a case statement is the evaluation of command line arguments as follows −

#!/bin/sh

option="${1}"

case ${option} in

-f) FILE="${2}"

echo "File name is $FILE"

;;

-d) DIR="${2}"

echo "Dir name is $DIR"

;;

\*)

echo "`basename ${0}`:usage: [-f file] | [-d directory]"

exit 1 # Command to come out of the program with status 1

;;

esac

Here is a sample run of the above program −

$./test.sh

test.sh: usage: [ -f filename ] | [ -d directory ]

$ ./test.sh -f index.htm

$ vi test.sh

$ ./test.sh -f index.htm

File name is index.htm

$ ./test.sh -d unix

Dir name is unix

$

**Loops**

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

* [The while loop](https://www.tutorialspoint.com/unix/while-loop.htm)
* [The for loop](https://www.tutorialspoint.com/unix/for-loop.htm)
* [The until loop](https://www.tutorialspoint.com/unix/until-loop.htm)
* [The select loop](https://www.tutorialspoint.com/unix/select-loop.htm)

**While Loop**

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

## Syntax

while command

do

Statement(s) to be executed if command is true

done

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

## Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

[Live Demo](http://tpcg.io/4zznfu)

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

Upon execution, you will receive the following result −

0

1

2

3

4

5

6

7

8

9

Each time this loop executes, the variable **a** is checked to see whether it has a value that is less than 10. If the value of **a** is less than 10, this test condition has an exit status of 0. In this case, the current value of **a** is displayed and later **a** is incremented by 1.

**For Loop**

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

## Syntax

for var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

## Example

Here is a simple example that uses the **for** loop to span through the given list of numbers −

[Live Demo](http://tpcg.io/jEjIA1)

#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9

do

echo $var

done

Upon execution, you will receive the following result −

0

1

2

3

4

5

6

7

8

9

Following is the example to display all the files starting with **.bash** and available in your home. We will execute this script from my root −

#!/bin/sh

for FILE in $HOME/.bash\*

do

echo $FILE

done

The above script will produce the following result −

/root/.bash\_history

/root/.bash\_logout

/root/.bash\_profile

/root/.bashrc

**Until Loop**

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

until command

do

Statement(s) to be executed until command is true

done

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

Upon execution, you will receive the following result −

0

1

2

3

4

5

6

7

8

9

**Select Loop**

The **select** loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

## Syntax

select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and **word1** to **wordN** are sequences of characters separated by spaces (words). Each time the **for** loop executes, the value of the variable var is set to the next word in the list of words, **word1** to **wordN**.

For every selection, a set of commands will be executed within the loop. This loop was introduced in **ksh** and has been adapted into bash. It is not available in **sh**.

## Example

Here is a simple example to let the user select a drink of choice −

[Live Demo](http://tpcg.io/0z7tLI)

#!/bin/ksh

select DRINK in tea cofee water juice appe all none

do

case $DRINK in

tea|cofee|water|all)

echo "Go to canteen"

;;

juice|appe)

echo "Available at home"

;;

none)

break

;;

\*) echo "ERROR: Invalid selection"

;;

esac

done

The menu presented by the select loop looks like the following −

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$

You can change the prompt displayed by the select loop by altering the variable PS3 as follows −

$PS3 = "Please make a selection => " ; export PS3

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

Please make a selection => juice

Available at home

Please make a selection => none

$

Sometimes you need to stop a loop or skip iterations of the loop.

In this chapter, we will learn following two statements that are used to control shell loops−

* The **break** statement
* The **continue** statement

## The infinite Loop

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

### Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

#!/bin/sh

a=10

until [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This loop continues forever because **a** is always **greater than** or **equal to 10** and it is never less than 10.

## The break Statement

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

### Syntax

The following **break** statement is used to come out of a loop −

break

The break command can also be used to exit from a nested loop using this format −

break n

Here **n** specifies the **nth** enclosing loop to the exit from.

### Example

Here is a simple example which shows that loop terminates as soon as **a** becomes 5 −

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

if [ $a -eq 5 ]

then

break

fi

a=`expr $a + 1`

done

Upon execution, you will receive the following result −

0

1

2

3

4

5

Here is a simple example of nested for loop. This script breaks out of both loops if **var1 equals 2** and **var2 equals 0** −

[Live Demo](http://tpcg.io/NcYiyO)

#!/bin/sh

for var1 in 1 2 3

do

for var2 in 0 5

do

if [ $var1 -eq 2 -a $var2 -eq 0 ]

then

break 2

else

echo "$var1 $var2"

fi

done

done

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.

1 0

1 5

## The continue statement

The **continue** statement is similar to the **break** command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

### Syntax

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here **n** specifies the **nth** enclosing loop to continue from.

### Example

The following loop makes use of the **continue** statement which returns from the continue statement and starts processing the next statement −

[Live Demo](http://tpcg.io/SN4DzV)

#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

Q=`expr $NUM % 2`

if [ $Q -eq 0 ]

then

echo "Number is an even number!!"

continue

fi

echo "Found odd number"

done

Upon execution, you will receive the following result −

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number