SHELL VARIABLES

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

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 would 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="SVEW CSE"

Above example defines the variable NAME and assigns it the value "SVEW CSE". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

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

VAR1="SVEW CSE"

VAR2=100

Accessing Values

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

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

#!/bin/sh
NAME="SVEW CSE"
echo $NAME
This would produce following value −

SVEW CSE

Read-only Variables

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

For example, following script would give error while trying to change the value of NAME −

#!/bin/sh
NAME="SVEW CSE"
readonly NAME
NAME="SV CSE"

This would produce following result −

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

Unsetting Variables

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

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

unset variable_name

Above command would unset the value of a defined variable. Here is a simple example −

#!/bin/sh

NAME="SVEW CSE"
unset NAME
echo $NAME

Above example would not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types

When a shell is running, three main types of variables are present −

- **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
- **Environment Variables** − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Non-alphanumeric characters in your variable names are those characters that are used in the names of special UNIX variables. These variables are reserved for specific functions.

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

$ echo $$

Above command would write PID of the current shell −

29949

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

Variable	Description
$0	The filename of the current script.
$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).
$#	The number of arguments supplied to a script.

$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	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 command line −

```
#!/bin/sh
echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh SVEW CSE
File Name : ./test.sh
First Parameter : SVEW
Second Parameter : CSE
Quoted Values: SVEW CSE
Quoted Values: SVEW CSE
Total Number of Parameters : 2
```

Special Parameters $* and $@

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

Both the parameter specifies all command-line arguments but 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 shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters −

```
#!/bin/sh
for TOKEN in $*
do
  echo $TOKEN
done
```

There is one sample run for the above script −

```
$./test.sh SVEW CSE 10 Years Old
SVEW
CSE
10
Years
Old
```

Note: Here **do**...**done** is a kind of loop which we would cover in 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 SVEW CSE
File Name : ./test.sh
First Parameter : SVEW
Second Parameter : CSE
Quoted Values: SVEW CSE
Quoted Values: SVEW CSE
Total Number of Parameters : 2
$echo $?
```

0
$

A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.

Shell supports a different type of variable called an array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

Defining Array Values

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

Say that 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 is to assign a value to one of its indices. This is expressed as follows −

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 **ksh** shell the here is the syntax of array initialization −

```
set -A array_name value1 value2 ... valuen
```

If you are using **bash** shell the 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 the simplest example −

```
#!/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]}"
```

This would produce 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 is the simplest example −

```
#!/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[@]}"
```

This would produce following result −

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

There are various operators supported by each shell.

There are following operators which we are going to discuss −

Arithmetic Operators.

Relational Operators.

Boolean Operators.

String Operators.

File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either awk or the must simpler program expr.

Here is simple example to add two numbers −

```
#!/bin/sh
val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result −

```
Total value : 4
```

There are following points to note down −

There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.

Complete expression should be enclosed between ``, called inverted commas.

Arithmetic Operators

There are following arithmetic operators supported by Bourne Shell.

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

Show Examples

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 % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$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 note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators:

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

For example, following operators would 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 −

Show Examples

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

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

Boolean Operators

There are following Boolean operators supported by Bourne Shell.

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

Show Examples

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 condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

There are following string operators supported by Bourne Shell.

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

Show Examples

Operator	**Description**	**Example**
=	Checks if the value of two operands are equal or not, if yes then 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 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 non-zero length then it returns true.	[-z $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

There are following operators to test various properties associated with a Unix file.

Assume a variable file holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on −

Show Examples

Operator	**Description**	**Example**
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.