

SHELL SCRIPTING WITH BASH

A Very Simple Introduction to the Eerie Gorgeous
World of Linux and Script

first Edition

Maedeh Mahmoudi

Intoduction

In a world where data grows exponentially and automation is the key to efficiency, Scripting plays a vital role in the world of IT and tech development.

Imagine being able to automate repetitive tasks, simplify complex processes, and step into a realm where control and smart system management are no longer just dreams.

Join us on this journey as we explore the powerful tool of Bash scripting an instrument capable of transforming the way we intract with technology forever.

I kindly request that you share your feedback and any corrections regarding my mistakes via the email below:

Email: maedeemahmoudi5@gmail.com

Linkdin: <https://www.linkedin.com/in/maede-mahmoudi-5ab18a284/>

also you can access to the code: <https://github.com/MaedeMahmoudi/bash-script>

Hoping for good days ahead.

contents

1.What is Bash?	5
2.What is Shell?	6
3.Why Should You Learn Bash Scripting?	7
4.Quick overview of common linux commands	8
5.Enviroment setup	12
6.Getting Started with Bash Script	13
7. Bash shell interpreter	15
8.Hello world	16
10.Variables	18
10.1.Defining Variables	18
10.2.Using & Reading Variables	20
10.3.Naming Variables	22
11.Inputs	23
11.1.Basic Usage of Read	23
11.2.Using Read With Prompts	24
11.3.Reading Multiple Input	25
11.4.Silent Input	26
12.Comments	27
12.1.Single-line Comment	27
12.2.Explaining Different Parts of Your Scripts	28
12.3.Comments For Debugging	28
13.Arguments	30
13.1.What Are Arguments in Bash?	30
13.2.How to Access Arguments	30
13.3.Default Arguments	32

14.Array	34
14.1.Deffining Array	34
14.2.Access Elements	34
14.3.Display All Elements	34
14.4.Adding & Changing Elements	35
14.4.Removing Elements	35
14.5.Array Length	35
15.Conditional Expression	37
15.1.Numeric Comparisons	37
15.2.String Comparisons	38
15.3.File Checks	38
15.4.Logical Expressions	39
16.Conditionals	40
16.1.If Statement	41
16.1.1.Structure of The If Statement	41
16.1.2.Different Uses of Conditions in If Statement	43
16.2. If Else Statement	46
16.2.1.Structure of The If Else Statement	46
16.2.2.How to Write & Work With Conditions in Bash	47
16.3.Switch Case Statement	50
17.Loops	53
17.1.For Loop	54
17.2.While Loop	59
17.3.Untile Loop	62
18.Continue & Break	65
19.The Last Word	69

What is Bash?

In the world of technology and computers, automating tasks can save time and make things more efficient. Bash is one of the most powerful and commonly used tools for automation on Unix-based systems like linux and mac.

Bash stands for “Bourne Again Shell,” and it is a type of command-line interface that allows users to interact with the operating system using text commands.

Bash is more than just a simple command prompt. It allows user to write scripts collections of commands saved in a file that can automatically perform complex or repetitive tasks.

This makes managing computers much easier. For example, if you want to back up important files every day, you can write a Bash script to do it automatically.

Bash was created in the 1980s and has become the standard shell in linux and many other Unix-like systems. With Bash, users and system administrators can control the system, run program, manage files, install software, and automate many others tasks.

Learning Bash helps you become more efficient in managing computers, avoiding mistakes, and saving time. Whether you’re a developer, system administrator, or just someone interested in technology, mastering Bash is very valuable.

In short, Bash is a language and tool for talking directly to your computer’s operating system.

Learning Bash is an important skill that can make you more effective in the tech world and open up many opportunities.

What is Shell?

In computers, a shell is a tool that acts as a middleman between you and the operating system. Think of the operating system as a powerful manager, but you need a special tool to talk to it.

A shell is a text-based interface that executes commands and programs. Working with the shell is similar to talking to your computer by typing commands, telling it what to do. Unix-based and Linux systems usually have different types of shells, such as Bash, Zsh, and Sh. Each has its own features, but they all share the main goal of controlling and managing the system.

One reason shells are popular is because you can write simple yet powerful scripts to automate tasks. For example, you can move or delete many files, make backups, install programs, or schedule daily routines with just a few lines of code. In this way, a shell acts like a smart assistant that performs repetitive and time-consuming tasks for you, making your work faster and less error-prone.

Overall, a shell is a very useful and important tool for managing computers. Learning how to use it is essential for system administrators and developers, and it helps everyone work more efficiently and communicate smarter with their computers.

Why Should You Learn Bash Scripting?

Learning Bash scripting is not just an important technical skill, but also a powerful tool for managing and improving computer systems' efficiency. Today, the amount of data and repetitive tasks that take time is growing daily. If you can automate these tasks with Bash scripts, you will save a lot of time and reduce mistakes.

Additionally, many server systems, development projects, and tools are built with Bash scripts. Learning this skill opens many doors for you, such as monitoring systems, automating software installation, backing up data, and doing daily tasks without being present all the time.

Moreover, knowing Bash scripting helps you understand how operating systems work and how to talk to them directly. This deep knowledge, strengthened through practice, is very valuable for your technical and professional growth.

In the end, writing Bash scripts brings many benefits: saving time, avoiding errors, increasing productivity, and sharpening your problem-solving skills. So, if you want to succeed in the world of technology, learning Bash scripting is one of the best investments you can make.

Quick overview of common linux commands

1. **ls** : Lists files and folders in the current directory.

```
[maede][sunflower][~]  
└─.  ls
```

2. **cd** : Changes to a different directory.

```
[maede][sunflower][~]  
└─.  cd Documents  
/home/maede/Documents
```

```
[maede][sunflower][~/Documents]  
└─.
```

3. **pwd** : Shows the full path of the current directory.

```
[maede][sunflower][~]  
└─.  pwd  
/home/maede
```

4. **mkdir** : Creates a new directory.

```
[maede][sunflower][~/Documents]  
└─.  mkdir myfolder
```

5. **rm** : Deletes file or directory.

```
[maede][sunflower][~/Documents]  
└─.  rm file.txt
```


6. **cp** : Copies files or directory.

```
└─[maede][sunflower][~/Documents]  
└─. cp file.txt backup/
```

7. **mv** : Moves or renames file or directory.

```
└─[maede][sunflower][~/Documents]  
└─. mv file.txt newfile.txt
```

8. **cat** : Displays the content of a file.

```
└─[maede][sunflower][~/Documents]  
└─. cat file.txt
```

9. **head** : Shows the first few lines of a file.

```
└─[maede][sunflower][~/Documents]  
└─. head file.txt
```

10. **tail** : Shows the last few lines of a file.

```
└─[maede][sunflower][~/Documents]  
└─. tail file.txt
```

11. **touch** : Creates a new empty file or updates timestamp.

```
└─[maede][sunflower][~/Documents]  
└─. touch newFile.txt
```

12. **find** : Finds specific files or directories.

```
└─[maede][sunflower][~]  
└─ find /home/user -name "file.txt"
```

13. **grep** : Searches for a pattern inside a file.

```
└─[maede][sunflower][~]  
└─ grep "hello" file.txt
```

14. **chmod** : Changes file or directory permissions.

```
└─[maede][sunflower][~]  
└─ chmod 775 script.sh
```

15. **chown** : Changes the owner of a file or directory.

```
└─[maede][sunflower][~]  
└─ chown user:user file.txt
```

16. **tar** : Compresses or extracts archive files.

```
[maede][sunflower][~/Documents]  
└─ tar -czvf archive.tar.gz folder/
```

17. **wget** : Downloads files from the internet.

```
[maede][sunflower][~/Documents]  
└─ wget http://example.com/file.zip
```

18. **ssh** : Connects to remote servers.

```
[maede][sunflower][~]
```

```
└─ ssh user@server
```

19. **ps** : Shows running processes.

```
[maede][sunflower][~]
```

```
└─ ps aux
```

20. **kill** : Stops running processes.

```
└─ [maede][sunflower][~]
```

```
└─ kill PID
```

Enviroment Setup

When you want to wite a Bash script, you need a good environment to wirte your code, these environment include programs like **Nano**, **VS Code**, **vim**, and other text editors.

Using these tools makes it easier to write, edit, and organize your code properly.
For example :

Nano : A **simple** and fast program for writing short scripts.

VS Code : An advanced editor with syntax highlighting and many features.

Vim : A powerful and popular text editor in Unix/Linux systems.

(Since we're just beginning and this book is only an introduction, we're using the Nano Editor.)

Getting Started with Bash Scripting

How and where can I run Bash scripts?

When you want to write and run a Bash script, there are some steps to know. First, you need to create the script file. This file contains Bash commands you want to run. Usually, we save it with a **.sh** extension, for example, `myscript.sh`.

To create a file as I told, you can use the `touch` command :

```
| touch fileName.sh
```

also you can use your text editor :

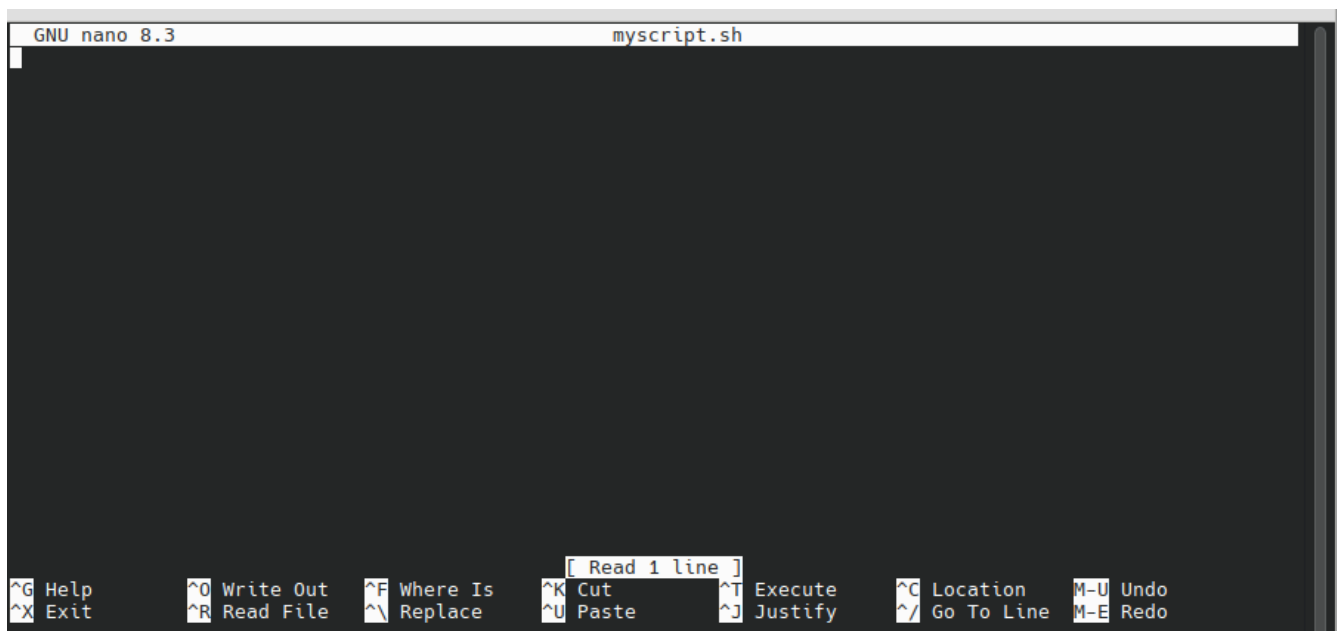
```
| nano fileName.sh
```

Now, let's do it together :

first make a file :

```
[maede][sunflower][~/Documents]  
└─ nano myscript.sh
```

and now you can see inside the file :



Note : Nano is usually installed by default on most Linux distributions like Ubuntu and Debian. If it's not installed, you can easily add it using the package manager.

Bash Shell Interpreter

A shell interpreter is a program that takes commands entered by the user in the command line and executes them. Basically, a shell acts as a bridge between the user and the operating system. When you type a command in the terminal or console, the shell interprets and runs it.

In Linux and Unix systems, there are different shells, but Bash is one of the most popular. Other shells include Zsh, Fish, and Csh. Each shell has its own way of writing commands and unique features.

The main role of the shell interpreter is to convert user commands into actions that the operating system can perform. For example, if you type "ls" to list files in a directory, the shell interprets this command and tells the OS to execute it. Then, it displays the results to you.

When you want to write a script or put your code into a file, you first need to open the file and include a special line at the very top. This line tells the system which interpreter should run the file. Usually, it looks like `#!/bin/bash`. This line is called the "shebang" and ensures that when you execute the script, the system knows to use Bash (or another specified shell) to run it.

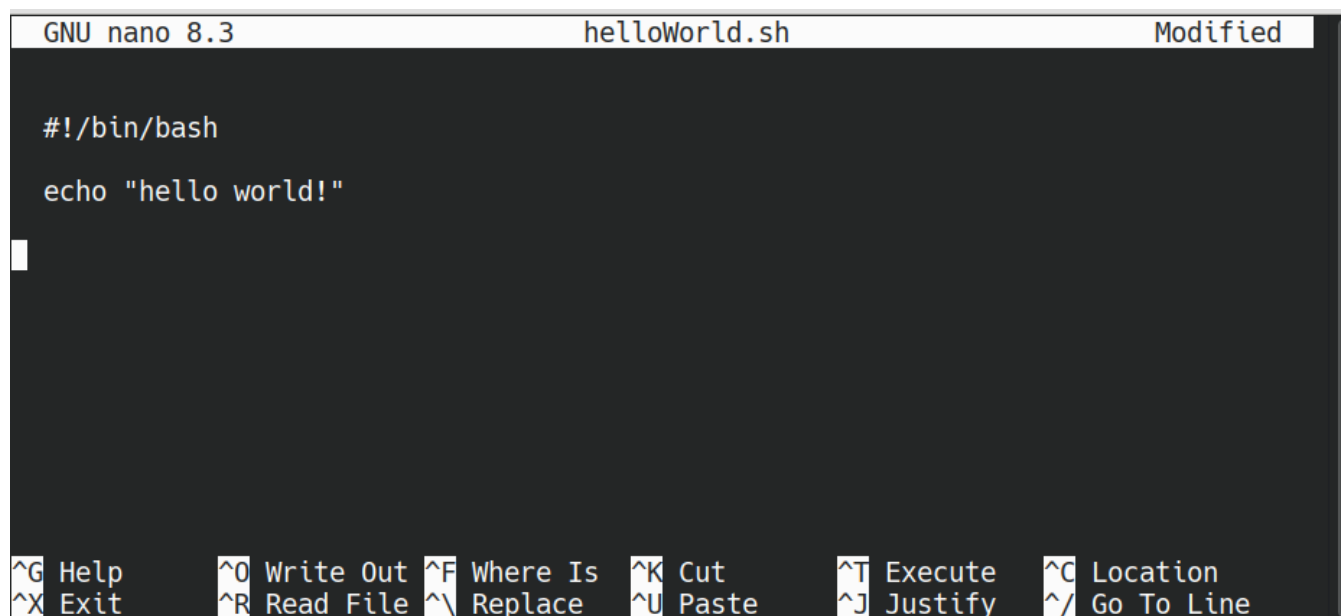
All that shebang does is to instruct the operating system to run the script with the `/bin/bash` executable.

```
#!/bin/bash
```

Hello World !

Alright, everyone! You've done very well so far. As always, now it's time to write our very first script.

In Bash scripting, echo is used to **display** strings. later on, I will explain strings more thoroughly. But for now, don't worry and just go ahead and write your first script, and enjoy it.



```
GNU nano 8.3                helloWorld.sh                Modified

#!/bin/bash
echo "hello world!"

^G Help    ^O Write Out  ^F Where Is  ^K Cut      ^T Execute   ^C Location
^X Exit    ^R Read File  ^\ Replace   ^U Paste    ^J Justify   ^_ Go To Line
```

after that press **ctrl + x** and then y to save the file and after that press enter.

Now **for running code** we have several ways but **simplest** way for start is to write bash, before file name :

```
bash fileName
```


Now, let's do it and see the result :

```
└─[maede][sunflower][±][main ? : 1 ✕][~/IdeaProjects/bash]
└─┐ bash helloWorld.sh
hello world!
```

it's greatttttt :D

Note : you can make your script **executable** with the command `chmod +x myscript.sh`, but there's no need to worry about more complicated things right now.

```
└─┐ chmod +x fileName.sh
```

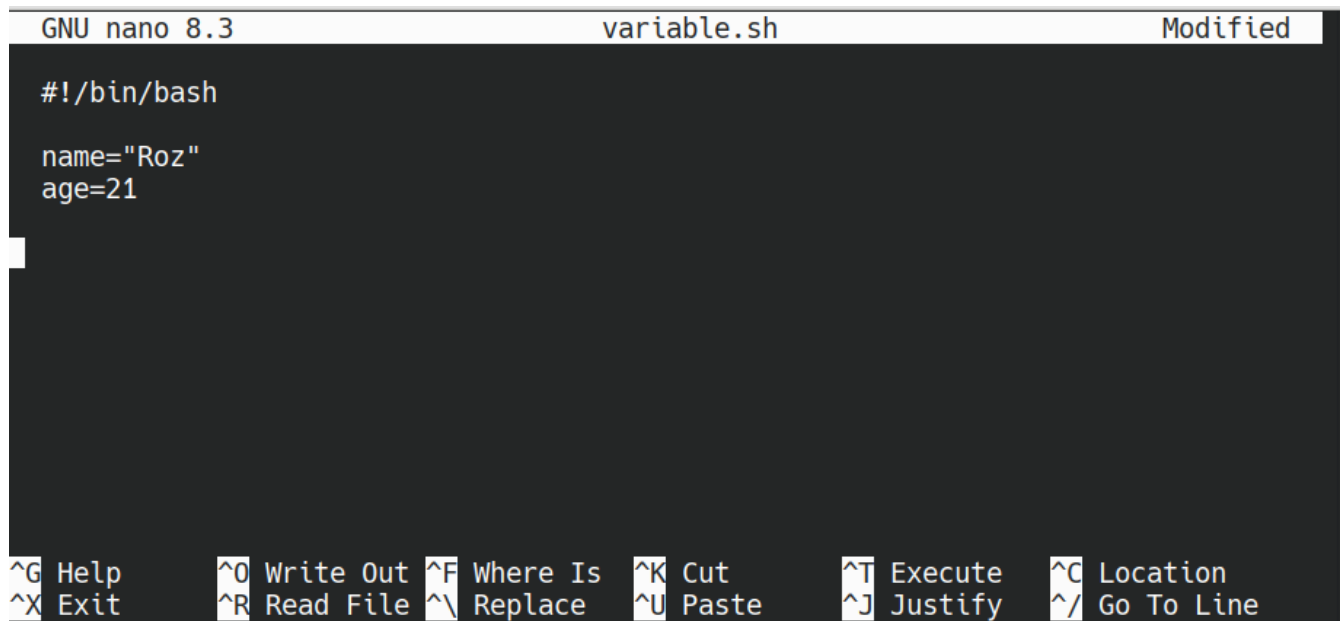
Variables

In Bash scripting, variables play an important role in storing and holding data. Using variables, you can save specific values and use them throughout your script.

Defining Variables :

To define a variable in Bash, simply write the variable name and assign a value to it with an equal sign (=).

Example:



```
GNU nano 8.3 variable.sh Modified
#!/bin/bash
name="Roz"
age=21
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

Now I have defined two variables named **name** and **age** and passed values to them.

Note : There should be no spaces around the =
For example:

```
| name = "Roz"           This is false !
```

```
| name="Roz"           This is true
```

Using and Reading Variables

To use a variable's value in your script, prefix the variable name with \$, like:

```
GNU nano 8.3 read.sh Modified

#!/bin/bash

name="Roz"
age=21

echo "hello $name"
echo "$name is $age years old."
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

This will output:

hello Roz

Roz is 21 years old.

Let's do it and see the result :

```
└─[maede][sunflower][~][main ?:1 X][~/IdeaProjects/bash]
└─┬─ bash read.sh
hello Roz
Roz is 21 years old.
```

Important tips:

- **Assigning Values:** In Bash, data types like numbers or strings are not explicitly defined; everything is treated as a **string**.
- **No Spaces:** When defining a variable, do not put spaces before or after the =.
- **Quoted Values:** If a variable contains spaces, it's better to enclose its value in quotes.

Different way to use variable :

```
GNU nano 8.3                                4_differentWay.sh                                Modified

#!/bin/bash

name="Roz"
age=21

echo "hello $name"
echo $name
echo ${name}

echo "$age years old"
echo $age
echo ${age}
```

File Name to Write: 4_differentWay.sh

^G Help	M-D DOS Format	M-A Append	M-B Backup File
^C Cancel	M-M Mac Format	M-P Prepend	^T Browse

output :

```
~[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─. bash 4_differentWay.sh
hello Roz
Roz
Roz
21 years old
21
21
```

- **Reusing Variables:** You can change the value of a variable later in the script:

Example:

```
GNU nano 8.3                    5_reusingVariables.sh                    Modified

#!/bin/bash

name="Roz"
name="Maede"

echo $name
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

output :

```
└─[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─. bash 5_reusingVariables.sh
Maede
```

- **Special Variables:** Bash has some special variables, such as \$? (the exit status of the last command), \$\$ (current process ID), and \$USER (current user name).

Naming Variables

- Must start with a letter or underscore.
- Should not contain spaces or special characters.
- Usually, English letters are used.

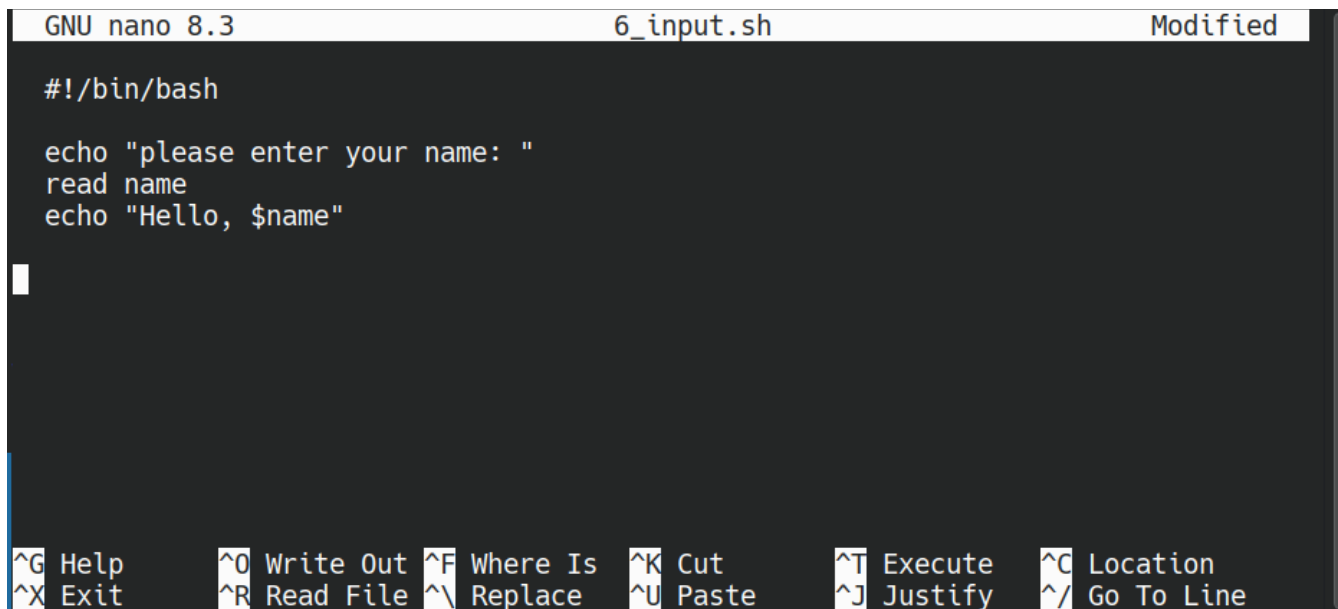
Inputs

In Bash scripting, **receiving input from the user** is a fundamental task that allows your script to interact dynamically with the person running it. This is mainly done using the `read` command. Understanding how to use `read` effectively is essential for creating flexible and user-friendly scripts.

Basic Usage of read

The simplest way to get input is by declaring a variable and assigning it a value entered by the user.

Example:



The image shows a terminal window with the GNU nano 8.3 editor. The file being edited is named 6_input.sh. The script content is as follows:

```
#!/bin/bash

echo "please enter your name: "
read name
echo "Hello, $name"
```

The bottom of the window displays a series of keyboard shortcuts for nano editor functions:

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute	^C Location
^X Exit	^R Read File	^N Replace	^U Paste	^J Justify	^_ Go To Line

Output:

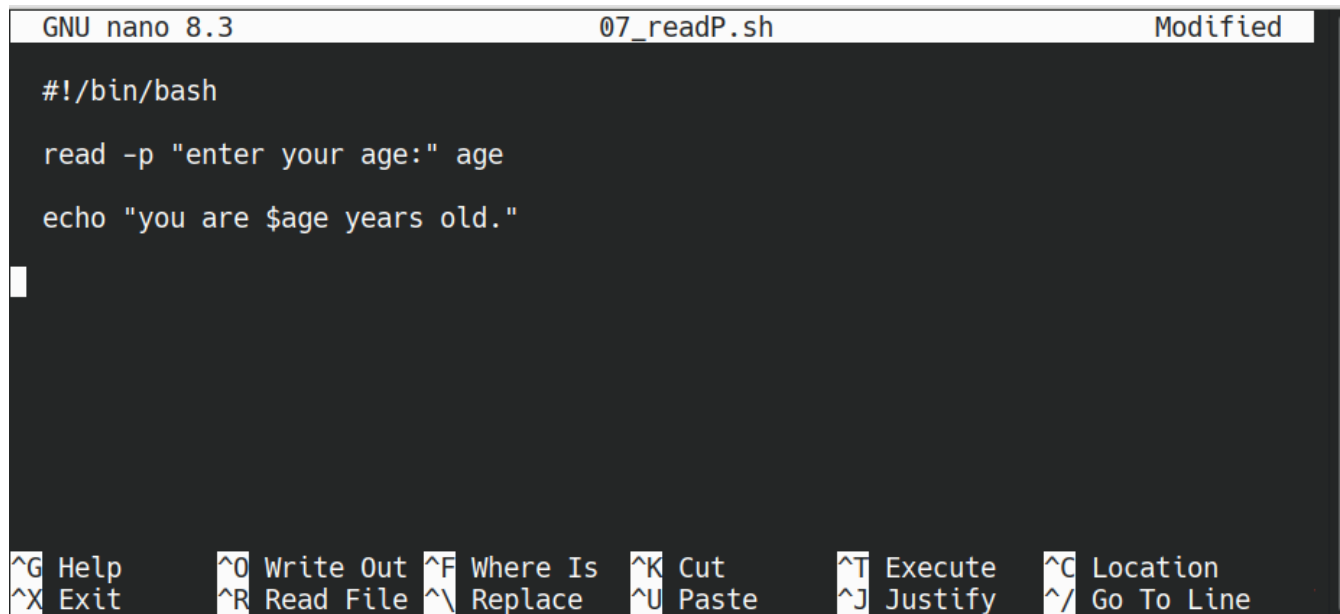
```
└─[maede][sunflower][±][main ? :1 ✕][~/IdeaProjects/bash]
└─. bash 6_input.sh
please enter your name:
Maede
Hello, Maede
```

Now better way :

Using read with Prompts

Instead of calling echo first, Bash offers a nifty feature: the -p option with read, which displays a prompt directly.

Example:

A screenshot of a terminal window with the GNU nano 8.3 editor. The title bar shows '07_readP.sh' and 'Modified'. The script content is:

```
#!/bin/bash
read -p "enter your age:" age
echo "you are $age years old."
```

 The cursor is at the end of the second line. The bottom status bar shows various keyboard shortcuts: ^G Help, ^O Write Out, ^F Where Is, ^K Cut, ^T Execute, ^C Location, ^X Exit, ^R Read File, ^\ Replace, ^U Paste, ^J Justify, ^_ Go To Line.

This line shows the message, waits for input, and stores it in age.

Output:

```
└─[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─. bash 07_readP.sh
enter your age:99
you are 99 years old.
```

Reading Multiple Inputs

To accept multiple inputs at once, list several variable names:
The user can input both values separated by spaces.

Example:

```
GNU nano 8.3                                08_multipleInput.sh                Modified

#!/bin/bash

read -p "enter your first and last name:" firstName lastName
echo "first name : $firstName, last name : $lastName"


```

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute	^C Location
^X Exit	^R Read File	^_ Replace	^U Paste	^J Justify	^_ Go To Line

output:

```
└─[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─. bash 08_multipleInput.sh
enter your first and last name:Maede mm
first name : Maede, last name : mm
```

Note: In future I will learn about loops and how to handle input line by line.

Silent Input (Password)

When asking for sensitive information like passwords, you can hide the input characters with the -s option.

The -s makes the input silent (not visible).

Example:

```
GNU nano 8.3                                09_silentInput.sh                Modified

#!/bin/bash

read -s -p "enter your pasword:" password
echo
echo "password recieved."
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

output:

```
└─[maede][sunflower][+][main ? : 1 X][~/IdeaProjects/bash]
└─┐ ── bash 09_silentInput.sh
enter your pasword:
password recieved.
```

Comments

In Bash, as in many programming languages, comments are parts of the code that are not executed. They are used only to explain the code so that anyone reading it, including yourself in the future, can understand what it does. Using comments is very important because it makes your scripts more understandable and easier to maintain.

In Bash, any line starting with the `#` symbol is a comment and is ignored during execution. You can use `#` at the beginning of a line or in the middle of a line next to code to add notes.

Examples of using comments:

Single-line comments:

One line comment.

```
GNU nano 8.3                                09_comment.sh                                Modified

#!/bin/bash

# this is a comment and does not run

echo "Hi" # this is an explanation beside an instruction
```

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute
^X Exit	^R Read File	^\ Replace	^U Paste	^J Justify

Explaining different parts of your script:

For complex scripts, it's good to write comments about different sections so you and others can quickly understand the script later.

Comments for debugging:

You can temporarily disable lines or add notes about what the code does.

```
GNU nano 8.3                                10_commentDebug.sh                                Modified

#!/bin/bash

#read -p "enter your username:" userName #this line is inactive

^G Help      ^O Write Out  ^F Where Is   ^K Cut        ^T Execute
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify
```

Important tips :

- **Where to place them:**

Comments can be at the start, middle, or end of lines. But best practice is to write each explanation on a separate line starting with # to keep the code clear.

- **Keep comments short:**

Don't write too much — only include important notes or parts that might be confusing later.

- **Long comments:**

If your explanation is lengthy, write it over multiple lines:

```
GNU nano 8.3      11_longComment.sh      Modified
#!/bin/bash

# this script calculates the sum of several numbers.
# the user enters multiple numbers, and the program shows their total.
```

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute
^X Exit	^R Read File	^\ Replace	^U Paste	^J Justify

Arguments

In Bash scripting, arguments are the inputs you give to a script when you run it. They make scripts flexible and customizable. Knowing how to work with arguments is essential for creating powerful scripts.

What Are Arguments in Bash?

When you run a script, you can add extra information (called arguments). Inside the script, you can access these arguments to control how the script works. Each argument is numbered, starting from 1.

How to Access Arguments

- `$0` – The name of the script itself.
- `$1, $2, $3, ...` – The first, second, third arguments, and so on.
- `${n}` – The proper way to refer to argument number `n`, especially when combining with text.
- `$#` – The total number of arguments you passed.
- `$*` – All arguments together as one string, separated by spaces.
- `$@` – All arguments as separate strings, useful when looping through them.

Note: The difference between `$*` and `$@` is that `$@` works better inside loops and when quoted.

Example:

```
GNU nano 8.3      12_acessArguments.sh      Modified

#!/bin/bash

echo "script name : $0"
echo "first argument : $1"
echo "second argument : $2"
echo "third argument : $3"
echo "number of arguments : $#"
```

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute
^X Exit	^R Read File	^\ Replace	^U Paste	^J Justify

Outputs:

```
[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─. bash 12_acessArguments.sh hello world test
script name : 12_acessArguments.sh
first argument : hello
second argument : world
third argument : test
number of arguments : 3
all arguments (as one string): hello world test
all arguments (as separate items): hello world test
```

Default Arguments

If the user doesn't give an argument, you can assign a default value:

Example:

```
GNU nano 8.3      13_defaultArg.sh      Modified

#!/bin/bash

name=${1:-guest}
echo "hello, $name"
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify

If no argument is provided, it will say "Hello, Guest!".

Output:

```
[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─ bash 13_defaultArg.sh
hello, guest

[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─ bash 13_defaultArg.sh Maede
hello, Maede
```

Advanced Tips

- Always quote arguments when they contain spaces or special characters:
" \$@ " or '...'.
- Arguments can be passed to other scripts or programs.
 - To access specific arguments, use conditions or patterns.
-

Array

In Bash scripting, arrays are powerful data structures that allow you to store and manage multiple values in a single variable. An array is essentially a list of items, each associated with an index (number). Bash arrays are supported by default in Bash versions, but not in some older shells like sh.

Overall, arrays are essential tools for handling multiple data items efficiently in Bash scripting.

Defining an Array:

To define an array, you use parentheses ().

```
arrayName=(value0 value1 value2 value3 ... )
```

Accessing Elements:

To get a specific element:

```
echo ${arrayName[number of index]}
```

Display all elements:

```
echo ${arrayName[@]}
```

Adding and Changing Elements:

You can add or change an element.

```
arrayName[index you want]=new value for assigning
```

Removing Elements:

To remove an element, use unset.

```
Unset 'arrayName[index you want to remove]'
```

#Array Length:

To find out how many items are in the array.

```
Echo ${#arrayName[@]}
```

Example:

```
GNU nano 8.3                                14_array.sh                                Modified
#!/bin/bash

fruits=(apple banana cherry)
echo "First element: ${fruits[0]}"           # Output: apple
echo "All array elements: ${fruits[@]}"       # Output: apple banana cherry

fruits[3]=cucumber # Add new element
echo "Array after adding new element: ${fruits[@]}"

unset 'fruits[1]' # Remove element
echo "Array after removing element 1: ${fruits[@]}"
echo "Array length: ${#fruits[@]}"           # Array length (not "array Length")

^G Help      ^O Write Out ^F Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

output:

```
—[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─. bash 14_array.sh
First element: apple
All array elements: apple banana cherry
Array after adding new element: apple banana cherry cucumber
Array after removing element 1: apple cherry cucumber
Array length: 3
```

Important Notes:

- Bash arrays are numbered (indexed) starting from 0.
 - You can easily add, edit, or remove elements.
 - Arrays are very useful for managing collections of data in scripts.
-

Conditional Expression

In Bash scripting, conditional expressions are sets of comparison, logical, string, numeric, and file tests used to evaluate conditions within if, test, or [] commands. These expressions form the core logic of Bash programs, helping determine the flow based on certain criteria.

A conditional expression is a combination of operators that returns true or false when evaluated. The results are used to make decisions in the script.

Types of Conditional Expressions:

Numeric Comparisons:

- The output is true if number1 is equal to number2.

```
[[ ${number1} eq ${number2} ]]
```

- The output is true if number1 & number2 are not equal.

```
[[ ${number1} ne ${number2} ]]
```

- The output is true if number1 greater than number2.

```
[[ ${number1} -gt ${number2} ]]
```

- The output is true if number1 less than number2.

```
[[ ${number1} -lt ${number2} ]]
```

- The output is true if number1 is greater than or equal number2.

```
[[ ${number1} -ge ${number2} ]]
```

- The output is true if number1 is less than or equal than number2.

```
[[ ${number1} -le ${number2} ]]
```

String Comparisons:

- The output is true if the strings are equal.

```
[[ ${str1} == ${str2} ]]
```

- The output is true if the strings are not equal.

```
[[ ${str1} != ${str2} ]]
```

- The output is true if string is empty.

```
[[ -z ${str} ]]
```

- The output is true if the string is not empty.

```
[[ -n ${str} ]]
```

File Checks:

- The output is true if the file exists.

```
[[ -e ${file} ]]
```

- The output is true if file exists and is a regular file.

```
[[ -f ${file} ]]
```

- The output is true if it is a directory.

```
[[ -d ${file} ]]
```

- The output is true if the file is readable.

```
[[ -r ${file} ]]
```

- The output is true if the file is writable.

```
[[ -w ${file} ]]
```

Logical Expressions:

- The output is true if both conditions are true.

```
[ "$a" -gt 10 ] && [ "$b" -lt 20 ]    #And
```

- The output is true if at least one condition is true.

```
[ "$a" -gt 10 ] || [ "$b" -lt 20 ]    #Or
```

Important tips:

- Always include spaces around brackets and operators.
- For combining conditions, you can use && and ||.
- Expressions are evaluated inside if, test, or [].

Note: In the next lecture when you learn about conditionals, I will use these expressions with them to write scripts and practice together, see you soon in the next lecture :D

Conditionals

A conditional is a programming concept that allows a program to make decisions based on certain conditions. It checks whether a specific statement or expression is true or false, and then it executes different parts of the code depending on the result.

In essence, conditionals help the program choose between different paths, making the code dynamic and adaptable to different situations. They are like asking a question: "Is this true?" If yes, do one thing; if no, do something else.

Common examples of conditionals include "if" statements, which test a condition and run certain commands if the condition is true. Sometimes, additional options like "else" or "elif" are used to handle more than one condition.

If Statement

In Bash scripts, one of the most important tools for controlling the flow of the program is the if statement. It allows you to run certain parts of the script only if a specific condition is true. This makes your script smarter and more flexible because it can react differently depending on different situations.

Structure of the `if` Statement:

The basic syntax of the `if` statement in Bash looks like this:

```
if [ condition ]; then
    # commands if the condition is true
fi
```

- `[condition]` is where you put the condition you want to check.
- `then` indicates that the commands after it will run if the condition is true.
- `fi` marks the end of the `if` statement (it's "if" backwards).

Example:

Suppose you want to check if a file called `example.txt` exists, if the file exists, the message "The file exists" will be printed.

```
GNU nano 8.3 15_if.sh

#!/bin/bash/

if [ -e example.txt ]; then
    echo "The file exists"
fi
```

[Read 7 lines]

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute	^C Location
^X Exit	^R Read File	^I Replace	^U Paste	^J Justify	^/_ Go To Line

Output:

```
—[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─ touch example.txt

└─[maede][sunflower][±][main ? :2 ×][~/IdeaProjects/bash]
└─ bash 15_if.sh
The file exists
```

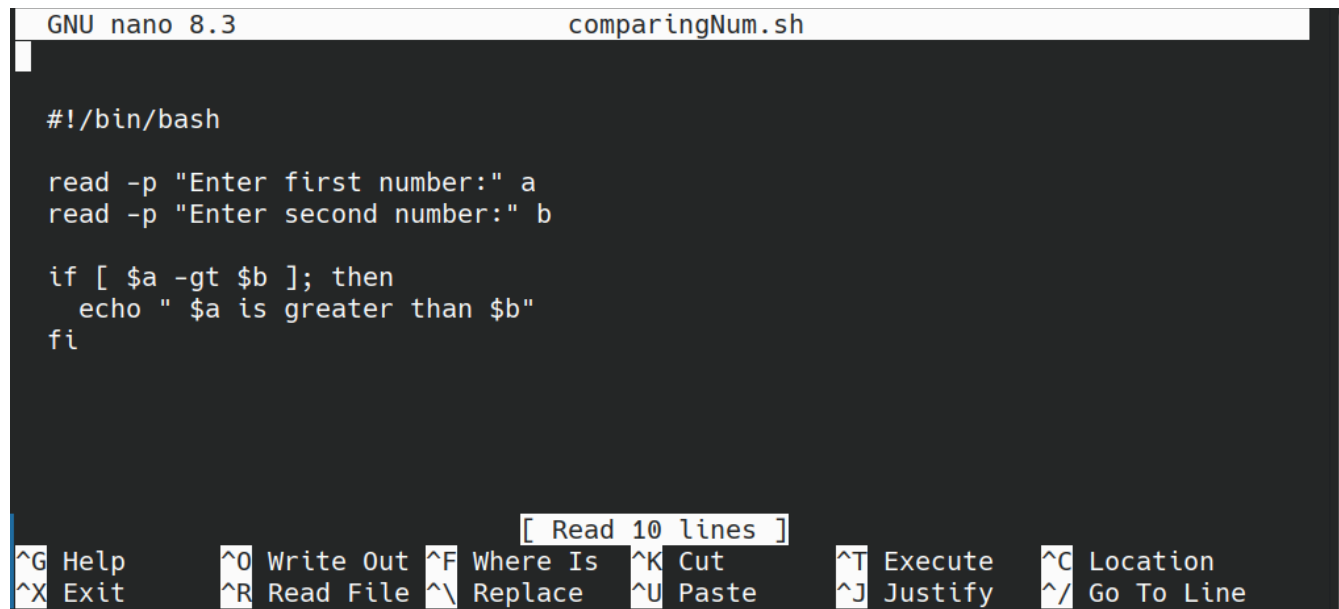
Different Uses of Conditions in `if`

1. Comparing Numbers

You can compare numbers using these operators:

- `-eq` (equal to)
- `-ne` (not equal to)
- `-gt` (greater than)
- `-lt` (less than)
- `-ge` (greater than or equal to)
- `-le` (less than or equal to)

Example:



```
GNU nano 8.3 comparingNum.sh

#!/bin/bash

read -p "Enter first number:" a
read -p "Enter second number:" b

if [ $a -gt $b ]; then
    echo " $a is greater than $b"
fi
```

[Read 10 lines]

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute	^C Location
^X Exit	^R Read File	^N Replace	^U Paste	^J Justify	^_ Go To Line

output:

```
[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─ bash comparingNum.sh
Enter first number:15
Enter second number:12
15 is greater than 12
```

2. Comparing Strings

For comparing text strings:

- = or == (are equal)
- != (are not equal)

Example:

```
GNU nano 8.3 17_comparingStr.sh Modified
#!/bin/bash/

read -p "enter the name : " name

if [ "$name" = "bash" ]; then
    echo "the name is bash"
fi
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

output:

```
└─[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─. bash 17_comparingStr.sh
enter the name : bash
the name is bash
```

3. Combining Multiple Conditions

To check more than one condition at the same time, you can use `-a` (and) and `-o` (or), or in newer versions, use `&&` and `||`

Example:













```
GNU nano 8.3 18_combiningCondition.sh

#!/bin/bash/

read -p "Enter first number = " a
read -p "Enter second number = " b

if [ "$a" -gt 10 ] && [ "$b" -lt 5 ]; then
    echo "both conditions are true."
fi
```

[Read 12 lines]

 Help	 Write Out	 Where Is	 Cut	 Execute	 Location
 Exit	 Read File	 Replace	 Paste	 Justify	 Go To Line

output:

```
└─[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─. bash 18_combiningCondition.sh
Enter first number = 15
Enter second number = 4
both conditions are true.
```

Practical Examples

1. Checking if a number is positive
 2. Checking if a string matches "yes"
-

If Else Statement

In Bash scripts, we use conditional statements to make decisions based on different situations.

Structure of the If Else Statement:

The syntax of the if else statement in Bash looks like this:

```
if [ condition ]; then
    # commands if the condition is true
elif [ condition ]; then
    # commands if the elif condition is true
else
    # commands if none of the above conditions are true
fi
```

Important tips:

- We put the `condition` inside square brackets `[]`.
 - After `if` and `elif`, we must write `then`.
 - The entire structure ends with `fi` (which is `if` backward).
 - We can have multiple `elif` to check different conditions.
 - The `else` goes at the end and runs if none of the previous conditions were true.
-

How to write and work with conditions in Bash

1. Review types of comparisons:

As I told you, For numbers and text, Bash has specific comparison operators:

Data type	Operator	Meaning	Example
Numbers	-gt	greater than >	["\$a" -gt "\$b"]
	-lt	less than <	["\$a" -lt "\$b"]
	-ge	greater or equal >=	["\$a" -ge "\$b"]
	-le	Less or equal <=	["\$a" -le "\$b"]
	-eq	Equal =	["\$a" -eq "\$b"]
	-ne	not equal !=	["\$a" -ne "\$b"]

For strings:

Operator	Meaning	Example
=	equal	["\$str1" = "\$str2"]
!=	not equal	["\$str1" != "\$str2"]
<	alphabetically before	["\$str1" \< "\$str2"]
>	alphabetically after	["\$str1" \> "\$str2"]

Note1: When comparing strings, make sure to use quotes around variables.

Note2:

< means the first string comes before the second string in alphabetical order.

> means the first string comes after the second string in alphabetical order.

Note3: In Bash, when using < or > inside [], you need to escape them with a \ to make the comparison work correctly, because without it, they may not work or could cause a syntax error. Like this \< or \> .

Example1: if else statement in strings

```
GNU nano 8.3 19_ifElseStr.sh Modified
#!/bin/bash

read -p "Enter the first word:" str1
read -p "Enter the second word:" str2
if [ "$str1" \< "$str2" ]; then
    echo "$str1 comes before $str2 in alphabet order."
elif [ "$str1" \> "$str2" ]; then
    echo "$str1 comes after $str2 in alphabet order."
else
    echo "Both words are the same."
fi
```

Help Exit Write Out Read File Where Is Replace Cut Paste Execute Justify Location Go To Line

output:

```
[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─ bash 19_ifElseStr.sh
Enter the first word:maede
Enter the second word:mino
maede comes before mino in alphabet order.
```


Example 2: Check if a number is positive, negative, or zero

```
GNU nano 8.3                                20_ifElseExample.sh                                Modified
#!/bin/bash/
read -p "Enter a number to check: " number
if [ "$number" -gt 0 ]; then
    echo "Number is positive"
elif [ "$number" -lt 0 ]; then
    echo "Number is negative"
else
    echo "Number is zero"
fi
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

output:

```
└─[maede][sunflower][±][main ✓][~/IdeaProjects/bash]
└─. bash 20_ifElseExample.sh
Enter a number to check: -20
Number is negative
```

Switch Case Statement

In Bash, to handle different situations, you can use the case ... in ... esac structure, which is similar to switch in languages like C, Java, or JavaScript.

Structure of Switch Case Statement:

```
case <variable> in
    <pattern1>)
        # Commands for pattern 1
        ;;
    <pattern2>)
        # Commands for pattern 2
        ;;
    *)
        # Default case if no pattern matches
        ;;
esac
```

- <variable> is the value you check.
- <pattern> is the value or pattern you match against.
-) ends each pattern line.
- ;; indicates the end of commands for each pattern.
- * is the default, running if nothing else matches.

Important tips:

- Use `)` to end each pattern.
 - End each pattern block with `;;`.
 - You can combine patterns with `|`.
 - Enclose string values in quotes.
 - Don't forget the `;;` at the end of each pattern; otherwise, you'll get a syntax error.
 - Use `|` to combine patterns with similar output.
-

Example 1:

```
GNU nano 8.3                21_swithCase1.sh                Modified
#!/bash/bin

read -p "Enter a number between 1 and 3: " number

case "$number" in
    1)
        echo " You entered one."
        ;;
    2)
        echo "You entered two."
        ;;
    3)
        echo "You entered three."
        ;;
    *)
        echo "Invalid number"
        ;;
esac
File Name to Write: 21_swithCase1.sh
^G Help          M-D DOS Format   M-A Append       M-B Backup File
^C Cancel        M-M Mac Format   M-P Prepend      ^T Browse
```

output:

```
└─[maede][sunflower][±][main ? :1 ✕][~/IdeaProjects/bash]
└─. bash 21_switchCase1.sh
Enter a number between 1 and 3: 3
You entered three.
```

Example 2:



```
GNU nano 8.3                                22_switchCase2.sh
#!/bash/bin

read -p "Enter a color: " color
case "$color" in
    "red"|"orange")
        echo "It's a warm color."
        ;;
    "blue"|"green")
        echo "It's a cool color."
        ;;
esac

[ Wrote 20 lines ]
^G Help      ^O Write Out ^F Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

output:

```
└─[maede][sunflower][±][main ? :1 ✕][~/IdeaProjects/bash]
└─. bash 22_switchCase2.sh
Enter a color: orange
It's a warm color.
```

Note: In this example:

- If \$color is "red" or "orange", it will display: "It's a warm color."
- If \$color is "blue" or "green", it will display: "It's a cool color."

Loops

Loops in programming are tools that let us repeat a specific part of the code many times. For example, if you want to show numbers from 1 to 10 or do the same task multiple times without writing it again each time. Loops do this automatically. With loops, you can execute commands as long as a certain condition is true, or process each item in a list one by one.

In Bash, there are different types of loops, and the most common ones are:

1. for loop
2. while loop
3. until loop

For Loop

A for loop in Bash is used to repeat commands based on each item in a list or collection. It is very useful when you want to perform actions on each element.

Structure of For Loop:

```
for <variable> in <list>
do
    <commands>
done
```

- <variable>: the name of a variable that will take each list item's value during each iteration.
- <list>: a list of items like strings, numbers, or phrases.
- do and done: enclose the commands to run in each loop.

Important tips:

- The list can include any kind of data, usually strings or numbers.
 - You can write long lists or generate them dynamically.
 - Inside the loop, you can run any commands.
 - If you have a file with many items, you can read and loop through its lines.
-

Example 1: Loop over a list of fruits

```
GNU nano 8.3                23_forOverList.sh                Modified

#!/bin/bash

for fruit in apple banana cherry
do
    echo "Fruit: $fruit "
done
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line

Output:

```
—[maede][sunflower][±][main ? :1 X][~/IdeaProjects/bash]
└─. bash 23_forOverList.sh
Fruit: apple
Fruit: banana
Fruit: cherry
```

Example 2: Loop over numbers

```
GNU nano 8.3                24_forLoop2.sh                Modified
#!/bin/bash

for i in 1 2 3 4 5
do
    echo "Number $i"
done
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line

Output:

```
—[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─. bash 24_forLoop2.sh
Number 1
Number 2
Number 3
Number 4
Number 5
```


Example 3: Using brace expansion to generate numbers

```
GNU nano 8.3                25_forLoop3.sh                Modified
#!/bin/bash
for i in {10..15}
do
    echo "Number: $i "
done
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify

Output:

```
[maede][sunflower][±][main ?:1 ×][~/IdeaProjects/bash]
└─. bash 25_forLoop3.sh
Number: 10
Number: 11
Number: 12
Number: 13
Number: 14
Number: 15
```

Example 4: Looping through lines of a file

Suppose you have a file called `hosts.txt`.

```
GNU nano 8.3                26_forLoop4.sh                Modified
#!/bin/bash
for host in $(cat hosts.txt)
do
    echo "Processing host: $host"
done
```

Help **Write Out** **Where Is** **Cut** **Execute**
Exit **Read File** **Replace** **Paste** **Justify**

Advanced tips:

- You can change the separator with `IFS`.
 - Loop commands run for each item in the list.
 - You can also nest `for` loops.
 - Be careful with the list format, especially when using brace expansion or command substitution.
-

While Loop

A while loop in Bash repeatedly runs commands as long as a certain condition is true. It keeps checking the condition before each iteration. If the condition is true, it runs the commands inside the loop, then checks again. This continues until the condition becomes false or you stop the loop.

Structure of For Loop:

```
while <condition>
do
    <commands>
done
```

Important tips:

- The <condition> should evaluate to true or false, like ["\$num" -lt 10].
 - The condition is checked before each loop cycle.
 - If true, commands inside run.
 - You can exit early with break.
 - You can skip to the next iteration with continue.
-

Example 1: Counting from 1 to 5

```
GNU nano 8.3 26_while1.sh
#!/bin/bash

num=1

while [ $num -le 5 ]
do
    echo "Number: $num"
    ((num++))
done
```

[Read 15 lines]

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute
^X Exit	^R Read File	^\ Replace	^U Paste	^J Justify

Output:





```
└─[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─ bash 26_while1.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Example 2: Getting user input until "exit" is entered

This script keeps asking the user to type something. If they type "exit", the program stops. Otherwise, it repeats what they typed.

```
GNU nano 8.3                27_while2.sh                Modified
#!/bin/bash

while true
do
    read -p "Enter 'exit' to quit " input
    if [ "$input" == "exit" ]; then
        break
    fi
    echo "You entered: $input"
done
```

Help	Write Out	Where Is	Cut	Execute
Exit	Read File	Replace	Paste	Justify

Output:

```
[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─. bash 27_while2.sh
Enter 'exit' to quit hi
You entered: hi
Enter 'exit' to quit no
You entered: no
Enter 'exit' to quit bash
You entered: bash
Enter 'exit' to quit exit
```

Until loop

An until loop in Bash runs commands repeatedly until a certain condition becomes true. In other words, it keeps executing as long as the condition is false. When the condition turns true, the loop stops.

Structure of For Loop:

```
until <condition>
do
    <commands>
done
```

Important tips:

- The condition is checked before each cycle.
 - If the condition is false, commands inside the loop run.
 - When the condition becomes true, the loop ends.
 - You can use break to stop the loop early.
 - Inside the loop, you can update the condition to control the loop's execution.
-

Example 1: Counting down from 5 to 1

```
GNU nano 8.3                28_untilLoop1.sh                Modified
#!/bin/bash

num=5

until [ $num -le 0 ]
do
    echo "Number: $num "
    ((num --))
done
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

Output:

```
└─[maede][sunflower][±][main ? : 1 X][~/IdeaProjects/bash]
└─. bash 28_untilLoop1.sh
Number: 5
Number: 4
Number: 3
Number: 2
Number: 1
```

Example 2: Getting user input until enter 'exit'

```
GNU nano 8.3                29_untilLoop2.sh                Modified
#!/bin/bash
until [ "$input" == "exit" ]
do
    read -p "If you want to exit, type 'exit': " input
    echo "You typed: $input"
done
```

Help Write Out Where Is Cut Execute Location
Exit Read File Replace Paste Justify Go To Line

Output:

```
[maede][sunflower][±][main ? :1 ×][~/IdeaProjects/bash]
└─ bash 29_untilLoop2.sh
If you want to exit, type 'exit': hi
You typed: hi
If you want to exit, type 'exit': hello
You typed: hello
If you want to exit, type 'exit': script
You typed: script
If you want to exit, type 'exit': exit
You typed: exit
```


Continue & Break

break:

The break command is used to immediately exit from a loop (for, while, or until).

When to use break?

When you want to stop the loop early, even if the loop's condition is still true.

Example:

```
GNU nano 8.3          30_break.sh          Modified
#!/bin/bash

for i in {1..10}
do
    if [ $i -eq 7 ];then
        echo "Found 7, stopping the loop"
        break
    fi
    echo "Number: $i "
done
```

^G Help	^O Write Out	^F Where Is	^K Cut	^T Execute	^C Location
^X Exit	^R Read File	^N Replace	^U Paste	^J Justify	^/ Go To Line

Output:

```
└─[maede][sunflower][±][main ? : 1 ✕][~/IdeaProjects/bash]
└─. bash 30_break.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Found 7, stopping the loop
```

continue:

The continue command is used to skip the current iteration and move on to the next one in a loop.

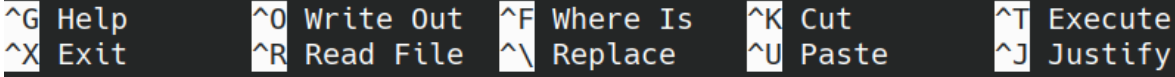
When to use continue?

When you want to ignore some specific cases during looping, and continue with the next cycle.

Example:

```
GNU nano 8.3          31_continue.sh          Modified
#!/bin/bash

for i in {1..5}
do
    if [ $i -eq 3 ]; then
        continue
    fi
    echo "Number: $i"
done
```



Output:

```
└─[maede][sunflower][+][main ? :1 X][~/IdeaProjects/bash]
└─. bash 31_continue.sh
Number: 1
Number: 2
Number: 4
Number: 5
```

Note: The number 3 is not shown because continue skips that iteration.

Important tips:

- break completely stops the loop.
 - continue skips only the current iteration and continues with the next.
 - They are very useful for controlling how loops behave under certain conditions.
-

The Last word...

I'm very happy that you reached this page of the book. It is an honor for me to have taken even a small step to share knowledge and to have helped you. Congratulations on finishing this book! Rest assured, you are already ahead of where you were yesterday.

Next, I and some friends who are experts in Linux are writing the second volume for this book. The second book will be different from the first one. Firstly, because writing the first volume helped me find many eager people to contribute to the second volume. It will include the knowledge of many experienced people. Secondly, the second volume will be in Persian, so we can better explain complex topics in our language. I hope everything goes well.

I'm looking forward to your feedback and comments on the first volume, so we can write an even better second volume based on your experiences and suggestions.

Best regards Maede Mahmoudi.