

Bash/Git

Introduction

In this course, you will learn how to improve your own programming process by writing bash scripts and Learning One of the most Known Control Version Tool Which is Git

Table of Contents:

Shell Scripting

Introduction to bash scripting

- 1. What is Bash?*
- 2. What is Shell?*
- 3. What is the Command Line?*
- 4. Why Learn the Command Line?*
- 5. RTFM*

Basic Shell Scripting

- 1. Shell basic*
- 2. Shell permission*
- 3. I/O Redirection and Filters*
- 4. Shell init files , variable and expansion*

Advanced Shell Scripting

- 1. Variable Assignment*
- 2. Shell Conditions*
- 3. Shell Loops*

Git and Github

Getting Started with Git

- 1. About Version Control*
- 2. What is Git?*
- 3. Installing Git*
- 4. First-Time Git Setup*
- 5. Git and GitHub*

Git Basics

- 1. Getting a Git Repository***
- 2. Using git with github***
- 3. Initializing a Repository***
- 4. Cloning an Existing Repository***
- 5. Remote and recording Changes***
- 6. Viewing the Commit History***
- 7. Undoing Things***

Branching

- 1. Branches in Nutshell***
- 2. Basic Branching and Merging***
- 3. Branch Management***
- 4. Branching Workflow***

Resources:

- **Bash Reference Manual**
- **Git and Github**
- **Learn Git and Github**(offred by Github)

Shell Scripting - Introduction

Bash scripting is a great way to automate repetitive tasks and can save you a ton of time as a developer. Bash scripts execute within a Bash shell interpreter terminal. Any command you can run in your terminal can be run in a Bash script. When you have a command or set of commands that you will be using frequently, consider writing a Bash script to perform it

1. What is Bash?

Bash, or the Bourne-Again SHell, is a command line interface that was created in 1989 by Brian Fox as a free software replacement for the Bourne Shell. A Shell is a specific kind of command line interface. Bash is “open source,” which means that anyone can read the code and suggest changes. Since its beginning, it has been supported by a large community of engineers who have worked to make it an incredible tool. Bash is the default Shell for Linux and Mac up through macOS 10.14 (Mojave). For these reasons, Bash is the most used and widely distributed shell. Shell scripting is scripting in any shell, whereas Bash scripting is scripting specifically for Bash. In practice, however, "Shell script" and "Bash script" are often used interchangeably, unless the shell in question is not Bash.

2. What is Shell?

At its base, a shell is simply a macro processor that executes commands. The term macro processor means functionality where text and symbols are expanded to create larger expressions.

A Unix shell is both a **command interpreter** and a **programming language**. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands, allowing users or groups to establish custom environments to automate their common tasks.

Shells may be used interactively or non-interactively. In interactive mode, they accept input typed from the keyboard. When executing non-interactively, shells execute commands read from a file.

A shell allows execution of GNU commands, both synchronously and asynchronously. The shell waits for synchronous commands to complete before accepting more input; asynchronous commands continue to execute in parallel with the shell while it reads and executes additional commands. The redirection constructs permit fine-grained control

of the input and output of those commands. Moreover, the shell allows control over the contents of commands' environments.

Shells also provide a small set of built-in commands (builtins) implementing functionality impossible or inconvenient to obtain via separate utilities. We will look into them

3. What is the Command Line?

The command line is a text interface for the computer's operating system. You can use it to traverse and edit your computer's filesystem. Through the command line, you can create new files, edit the contents of those files, delete files, and more!

4. Why Learn the Command Line?

We use our mouse and fingers to click images of icons and access files, programs, and folders on our devices. However, this is just one way for us to communicate with computers. The command line is a quick, powerful, text-based interface developers use to more effectively and efficiently communicate with computers to accomplish a wider set of tasks. Learning how to use it will allow you to discover all that your computer is capable of!

5. RTFM

We will now talk about one of the most important things that you should remember and use Which is Man Page. Every single command has something called Manual Page or Man . When you are going through learning in this course or your future daily job it will be useful to help you navigate the **flags** and option of command and how to use it .

Basic Shell Scripting

1. Shell basic

In this chapter we will talk about how to get you started navigating the system and before that i invite you to look to some basic Unix File System and some Concepts

- **Unix/Linux Getting Started**
- **Unix/Linux File Management**

➤ Unix / Linux - Directory Management

A. Navigating the file system

The most fundamental skills you need to master are moving around the filesystem and getting an idea of what is around you. We will discuss the tools that allow you to do this in this section.

➔ *Finding Where You Are with the “pwd” Command:*

When you log into your server, you are typically dropped into your user account’s home directory. A home directory is a directory set aside for your user to store files and create directories. It is the location in the filesystem where you have full dominion. To find out where your home directory is in relationship to the rest of the filesystem, you can use the pwd command. This command displays the directory that we are currently in:

```
$ pwd
```

Example:

```
themis@Ahmed-Belhaj:~$ pwd
/home/themis
themis@Ahmed-Belhaj:/bin$ pwd
/bin
```

In this Example on top we can see that the first time our **Parent Working Directory** is “/home/themis”

And in the Second is “/bin”

The home directory is named after the user account, so the above example is what the value would be if you were logged into the server with an account called themis . This directory is within a directory called /home, which is itself within the top-level directory, which is called “root” but represented by a single slash “/”.

In the Second Example we we were in the /bin directory which is also a top-level directory

➔ *Looking at the Contents of Directories with “ls”*

Now that you know how to display the directory that you are in, we can show you how to look at the contents of a directory. Currently, your home directory that we saw above does not have much to see, so we will go to another which is /bin the one shown in the second example you . /bin is a more populated directory to explore. Type the following

in your terminal to move to this directory (we will explain the details of moving directories in the next section). Afterward, we'll use pwd to confirm that we successfully moved:

```
themis@Ahmed-Belhaj:$ cd /bin
themis@Ahmed-Belhaj:$ pwd
/bin
```

Now that we are in a new directory, let's look at what's inside. To do this, we can use the ls command:

```
themis@Ahmed-Belhaj:/bin$ ls
expr                mtrace              sg_prevent
Xkill               factor              mv
sg_raw              xlsatoms            faillog
my_print_defaults  sg_rbuf             xlsclients
faked-sysv          myisam_ftdump       sg_rdac
Xlsfonts            faked-tcp           myisamchk
sg_read             xmessage            sbsiglist
...
```

As you can see there are a lot of directory and files , we can add some optional flags to the command to modify the default behavior . but before that lets move to another directory so we can see the difference clearer . As you can see, there are many items in this directory. We can add some optional flags to the command to modify the default behavior. For instance, to list all of the contents in an extended form, we can use the -l flag (for "long" output):

```
themis@Ahmed-Belhaj:/bin$ cd /usr/share
themis@Ahmed-Belhaj:/usr/share$ pwd
/usr/share
themis@Ahmed-Belhaj:/usr/share$ ls
GConf                doc-base            libthai
ImageMagick-6        dpkg                 licenses
PackageKit           drirc.d              lintian
X11                  emacs                locale
...
themis@Ahmed-Belhaj:/usr/share$ ls -l
total 712
drwxr-xr-x   3 root root  4096 Aug 19 22:41 GConf
drwxr-xr-x   2 root root  4096 Dec 28 13:19 ImageMagick-6
drwxr-xr-x   3 root root  4096 Aug 19 22:41 PackageKit
```

```
drwxr-xr-x    4 root root  4096 Aug 19 23:20 X11
drwxr-xr-x    2 root root  4096 Nov 16 13:12 acllocal
drwxr-xr-x    2 root root  4096 Aug 19 22:40 adduser
drwxr-xr-x    7 root root  4096 Nov 16 13:12 alsa
drwxr-xr-x    6 root root  4096 Sep 30 11:29 apache2
```

PS : Command are case sensitive for example LS will not work the Correct one is ls

This view gives us plenty of information, most of which looks rather unusual. The first block describes the file type (if the first column is a “d” the item is a directory, if it is a “-”, it is a normal file) and permissions. Each subsequent column, separated by white space, describes the number of hard links, the owner, group owner, item size, last modification time, and the name of the item. We will describe some of these at another time, but for now, just know that you can view this information with the -l flag of ls. To get a listing of all files, including hidden files and directories, you can add the -a flag. Since there are no real hidden files in the /usr/share directory, let’s go back to our home directory and try that command. You can get back to the home directory by typing cd with no arguments:

```
themis@Ahmed-Belhaj:/usr/share$ cd
themis@Ahmed-Belhaj:~$ ls -a
.  .. .bash_logout .bashrc .profile
```

As you can see, there are three hidden files in this demonstration, along with . and .., which are special indicators. You will find that often, configuration files are stored as hidden files, as is the case here. For the dot and double dot entries, these aren’t exactly directories as much as built-in methods of referring to related directories. The single dot indicates the current directory, and the double dot indicates this directory’s parent directory. This will come in handy in the next section. We will take a look at those Three files later

→ *Moving Around the Filesystem with “cd”*

We have already made two directory moves in order to demonstrate some properties of ls in the last section. Let’s take a better look at the command here. Begin by going back to the /usr/share directory by typing this:

```
themis@Ahmed-Belhaj:~$ cd /usr/share
```

This is an example of changing a directory by giving an **absolute path**. In Linux, every file and directory is under the top-most directory, which is called the “root” directory,

but referred to by a single leading slash “/”. An absolute path indicates the location of a directory in relation to this top-level directory. This lets us refer to directories in an unambiguous way from any place in the filesystem. Every absolute path must begin with a slash. The alternative is to use **relative paths**. Relative paths refer to directories in relation to the current directory. For directories close to the current directory in the hierarchy, this is usually easier and shorter. Any directory within the current directory can be referenced by name without a leading slash. We can change to the “**locale**” directory within /usr/share from our current location by typing:

```
themis@Ahmed-Belhaj:/usr/share$ cd locale
themis@Ahmed-Belhaj:/usr/share/locale$
```

We can likewise move multiple directory levels with relative paths by providing the portion of the path that comes after the current directory’s path. From here, we can get to the LC_MESSAGES directory within the en directory by typing:

```
themis@Ahmed-Belhaj:/usr/share/locale$ cd en/LC_MESSAGES/
themis@Ahmed-Belhaj:/usr/share/locale/en/LC_MESSAGES$
```

You can see the path change in your prompt but if your prompt is different you can check as we do before using **pwd**

To go back up, travelling to the parent of the current directory, we use the special double dot indicator we talked about earlier. For instance, we are now in the /usr/share/locale/en/LC_MESSAGES directory. To move up one level, we can type:

```
themis@Ahmed-Belhaj:/usr/share/locale/en/LC_MESSAGES$ cd ..
themis@Ahmed-Belhaj:/usr/share/locale/en$
```

This takes us to the /usr/share/locale/en directory.

A shortcut that you saw earlier that will always take you back to your home directory is to use **cd** without providing a directory:

```
themis@Ahmed-Belhaj:/usr/share/locale/en$ cd
themis@Ahmed-Belhaj:~$ pwd
/home/themis
```


You can do that also by using **cd ~** or **cd \$HOME**

```
themis@Ahmed-Belhaj:/usr/share/locale/en$ cd ~
themis@Ahmed-Belhaj:~$ pwd
/home/themis
```

```
themis@Ahmed-Belhaj:/usr/share/locale/en$ cd $HOME
themis@Ahmed-Belhaj:~$ pwd
/home/themis
```

PS : \$HOME is called Environment Variable We will talk about the later

You can also navigate to the last visited directory using **cd -**

```
themis@Ahmed-Belhaj:~$ cd -
/usr/share/locale/en
themis@Ahmed-Belhaj:/usr/share/locale/en$
```

→ What do you find in the most common/important directories/files

Although organizations have made strides toward consistency via standards such as the Linux Filesystem Hierarchy Standard (FHS), different Linux distributions still have somewhat different directory structures. The following rendering exemplifies a typical Red Hat effort toward standardization of where files are stored according to type and use.

Directory	Description
/bin	All binaries needed for the boot process and to run the system in single-user mode, including essential commands such as cd, ls, etc.
/boot	Holds files used during the boot process along with the Linux kernel itself
/dev	Contains device files for all hardware devices on the system

/etc	Files used by application subsystems such as mail, the Oracle database, etc
/home	User home directories
/lib	Some shared library directories, files, and links
/mnt	The typical mount point for the user-mountable devices such as floppy drives and CDROM
/proc	Virtual file system that provides system statistics. It doesn't contain real files but provides an interface to runtime system information.
/root	Home directory for the root user
/sbin	Commands used by the super user for system administrative functions
/tmp	A standard repository for temporary files created by applications and users.
/var	Administrative files such as log files, locks, spool files, and temporary files used by various utilities
/usr	Directory contains subdirectories with source code, programs, libraries, documentation, etc.

The contents of these directories will vary from system to system but most of these directories will typically be present. Often when you install software or a new device on a Linux system files will be added or modified in these directories to make everything work.

Resources:

- **Important Files and Directories**
- **man ls**
- **man pwd**
- **man cd**

B. Manipulating Files

These five commands are among the most frequently used Bash commands.

- ❖ **touch** - create empty files
- ❖ **mkdir** - create directories
- ❖ **cp** - copy files and directories
- ❖ **mv** - move or rename files and directories
- ❖ **rm** - remove files and directories

They are the basic commands for manipulating both files and directories. Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, you can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs ? The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs.

→ *The «mkdir» Command*

The mkdir command can be used to create a directory without any content. To create empty directory , we just mkdir directory name or names :

```
themis@Ahmed-Belhaj:~$ mkdir bash
themis@Ahmed-Belhaj:~$ ls
bash
themis@Ahmed-Belhaj:~$ mkdir git github
themis@Ahmed-Belhaj:~$ ls
bash  git  github
themis@Ahmed-Belhaj:~$
```

The mkdir command will complain if the directory we're about to create exists already. For instance, we'll get an error if we attempt to create another directory called "bash":

```
themis@Ahmed-Belhaj:~$ mkdir bash
mkdir: cannot create directory 'bash': File exists
```

A very convenient option on the mkdir command is -p. The -p option allows us to create parent directories as necessary and not complain if the directories exist.

Let's create some subdirectories under the existing one directory – we'll list our result with tree:

```
themis@Ahmed-Belhaj:~$ mkdir -p course/bash course/git
themis@Ahmed-Belhaj:~$ tree -d
.
```

```
|— bash
|— course
|   |— bash
|   |— git
|— git
|— github
```

6 directories

Resource:

- **tree Command**
- **man mkdir**

→ *The «touch» Command*

The touch command can be used to create files without any content. And it a bit similar to mkdir when it come to using , lets move up to course directory and try creating a file in it called script

```
themis@Ahmed-Belhaj:~$ cd course
themis@Ahmed-Belhaj:~/course$ ls
bash  git  script
```

You can also create multiple files and you can create in another directory just by supplying the path to the filename. Let create 1 files called git_config in the git directory and and another file called git_setup in bash directory

```
themis@Ahmed-Belhaj:~/course$ touch bash/git_setup git/git_config
themis@Ahmed-Belhaj:~/course$ tree
```

```
.
|— bash
|   |— git_setup
|— git
|   |— git_config
|— script
```

2 directories, 3 files

Except for creating empty files, the touch command can help us to update the access time and the modification time of a file. We can use the -a option to update the access time for script

```

themis@Ahmed-Belhaj:~/course$ ls -u --full-time
total 8
drwxr-xr-x 2 themis themis 4096 2022-01-11 16:01:24.414074300 +0100 bash
drwxr-xr-x 2 themis themis 4096 2022-01-11 16:01:24.414074300 +0100 git
-rw-r--r-- 1 themis themis 0 2022-01-11 16:04:31.204074300 +0100 script
themis@Ahmed-Belhaj:~/course$ touch -a --date='1992-08-08 17:17:59' script
themis@Ahmed-Belhaj:~/course$ ls -u --full-time
total 8
drwxr-xr-x 2 themis themis 4096 2022-01-11 16:01:24.414074300 +0100 bash
drwxr-xr-x 2 themis themis 4096 2022-01-11 16:01:24.414074300 +0100 git
-rw-r--r-- 1 themis themis 0 1992-08-08 17:17:59.000000000 +0100 script

```

In the above example, we gave the `-u` option to the `ls` command in order to show the access time in the file list.

The `-m` option is for changing the modification time of a file. For example, let's change the modification time of the `git_config` to some time in the '70s:

```

themis@Ahmed-Belhaj:~/course$ touch -m --date='1976-11-18 18:19:59'
git/git_config
themis@Ahmed-Belhaj:~/course$ ls --full-time git/
total 0
-rw-r--r-- 1 themis themis 0 1976-11-18 18:19:59.000000000 +0100 git_config

```

→ The « *rm* & *rmdir* » Command

The `rm` command does the opposite of creating files and directories: It removes them. To remove files with `rm` is easy, we just add the filenames we want to remove after the `rm` command. For example, let's say I want to remove the `script` file located in the `course` directory.

```

themis@Ahmed-Belhaj:~/course$ ls
bash  git  script
themis@Ahmed-Belhaj:~/course$ rm script
themis@Ahmed-Belhaj:~/course$ ls
bash  git

```

By default, `rm` does not remove directories. We can make use of the `-d` option to remove empty directories, let's start by creating an empty directory and removing it using `rm`

```

themis@Ahmed-Belhaj:~/course$ mkdir test

```

```

themis@Ahmed-Belhaj:~/course$ ls
bash git test
themis@Ahmed-Belhaj:~/course$ rm -d test/
themis@Ahmed-Belhaj:~/course$ ls
bash git

```

We can also do the same using `rmdir`, which is a command to remove directory by default it delete only empty folder

```

themis@Ahmed-Belhaj:~/course$ mkdir test
themis@Ahmed-Belhaj:~/course$ ls
bash git test
themis@Ahmed-Belhaj:~/course$ rmdir test/
themis@Ahmed-Belhaj:~/course$ ls
bash git

```

If we apply the same command on the directory `git`, we'll get an error since the directory one is not empty:

```

themis@Ahmed-Belhaj:~/course$ rm -d git/
rm: cannot remove 'git/': Directory not empty
themis@Ahmed-Belhaj:~/course$ rmdir git/
rmdir: failed to remove 'git/': Directory not empty
themis@Ahmed-Belhaj:~/course$

```

If we want to remove directories and their contents recursively, we should use the `rm`'s `-r` option:

```

themis@Ahmed-Belhaj:~/course$ rm -r git/
themis@Ahmed-Belhaj:~/course$ ls
bash

```

At some point we have files who are write-protected and can't be removed, However `-f` option overrides this minor protection and remove the file forcefully, let's start by creating new directory called "git" and subdirectory called `github`, inside the `github` directory create a file called "git_credential" and change it's permissions to readonly using "chmod"(we will discuss in the next the chapter the permissions)

```

themis@Ahmed-Belhaj:~/course$ mkdir git/github
themis@Ahmed-Belhaj:~/course$ touch git/github/git_credential
themis@Ahmed-Belhaj:~/course$ chmod ugo=r git/github/git_credential
themis@Ahmed-Belhaj:~/course$ ls -l git/github/
total 0

```

```
-r--r--r-- 1 themis themis 0 Jan 12 12:01 git_credential
themis@Ahmed-Belhaj:~/course$ rm git/github/git_credential
rm: remove write-protected regular empty file 'git/github/git_credential'?
y
themis@Ahmed-Belhaj:~/course$ ls -l git/github/
total 0
-r--r--r-- 1 themis themis 0 Jan 12 12:01 git_credential
```

Let's now try using -f option :

```
themis@Ahmed-Belhaj:~/course$ rm -f git/github/git_credential
themis@Ahmed-Belhaj:~/course$ ls -l git/github/
total 0
```

The rm command works normally silently. So, we should be very careful while executing the rm command, particularly with the -r and the -f options. Once the files or directories get deleted, it's really hard to recover them again.

→ *The «cp» Command*

The cp command is used to copy files or directories.

Let's take a look at how to make a copy of git_credential and name it git_credential_v2. But first we need to create it again .

```
themis@Ahmed-Belhaj:~/course$ cd git/github/
themis@Ahmed-Belhaj:~/course/git/github$ touch git_credential
themis@Ahmed-Belhaj:~/course/git/github$ ls
git_credential
themis@Ahmed-Belhaj:~/course/git/github$ cp git_credential
git_credential_v2
themis@Ahmed-Belhaj:~/course/git/github$ ls
git_credential  git_credential_v2
```

Often we want to copy multiple files in Linux. To do that, we just pass the names of the files followed by the destination directory to the cp command:

```
themis@Ahmed-Belhaj:~/course/git/github$ cd ../../
themis@Ahmed-Belhaj:~/course$ tree
.
├── bash
├── git_setup
└── git
```

```

├── github
│   ├── git_credential
│   └── git_credential_v2
└── 3 directories, 3 files

themis@Ahmed-Belhaj:~/course$ cp git/github/git_credential
git/github/git_credential_v2 git/
themis@Ahmed-Belhaj:~/course$ tree
.
├── bash
├── git_setup
├── git
│   ├── git_credential
│   ├── git_credential_v2
│   └── github
│       ├── git_credential
│       └── git_credential_v2
└── 3 directories, 5 files

```

Of course, we can do the same by file globbing:

```

themis@Ahmed-Belhaj:~/course$ cd git/github
themis@Ahmed-Belhaj:~/course/cd git/github$ cp * ..
themis@Ahmed-Belhaj:~/course/cd git/github$ cd ../../
themis@Ahmed-Belhaj:~/course$ tree
.
├── bash
├── git_setup
├── git
│   ├── git_credential
│   ├── git_credential_v2
│   └── github
│       ├── git_credential
│       └── git_credential_v2
└── 3 directories, 5 files

```

PS: the “*” is a Wildcard in our case it means all the files in current directory

We copied all the files from git/github to the “..” directory which is git

Another everyday usage of file copying would be copying some source directory and all contents under it to a target directory. To do that, we pass the -R option, and cp will recursively copy the source directory:

```
themis@Ahmed-Belhaj:~/course$ tree -F
.
├── bash/
│   ├── git_setup
│   └── git/
│       ├── git_credential
│       ├── git_credential_v2
│       └── github/
│           ├── git_credential
│           └── git_credential_v2
└── 3 directories, 5 files

themis@Ahmed-Belhaj:~/course$ cp -R git/github/ bash/
themis@Ahmed-Belhaj:~/course$ tree -F
.
├── bash/
│   ├── git_setup
│   ├── github/
│   │   ├── git_credential
│   │   └── git_credential_v2
│   └── git/
│       ├── git_credential
│       ├── git_credential_v2
│       └── github/
│           ├── git_credential
│           └── git_credential_v2
└──
```

→ *The «mv» Command*

We can use the mv command to move files or directories. The command syntax of mv is similar to cp. Let's take a look at how to use the mv command to move a file or a directory but first let's setup some work directories right

```
themis@Ahmed-Belhaj:~/course$ rm -rf bash/github/
themis@Ahmed-Belhaj:~/course$ tree
.
├── bash
│   └── git_setup
└──
```

```
└─ git
   ├── git_credential
   ├── git_credential_v2
   └─ github
      ├── git_credential
      └─ git_credential_v2
```

3 directories, 5 files

```
themis@Ahmed-Belhaj:~/course$ mv git/git_credential_v2 bash/
themis@Ahmed-Belhaj:~/course$ mv bash/git_credential_v2 first_script.sh
themis@Ahmed-Belhaj:~/course$ mv git/github/ .
themis@Ahmed-Belhaj:~/course$ tree
```

```
.
├─ bash
│   └─ git_setup
├─ first_script.sh
├─ git
│   └─ git_credential
└─ github
   ├── git_credential
   └─ git_credential_v2
```

3 directories, 5 files

In this example on top in the first time we used mv we moved git_credential_v2 located in git directory to the bash directory , in second time we actually didn't move any file we just renamed the git_credential_v2 to first_script.sh

And third example we just moved the github directory out of the git to course
Renaming a file or directory is a common usage of the mv command.

We can move multiple files to a target directory as well

Syntax:

```
$ mv src_file src_file2 src_file3 targetdir
```

And yet we can use globbing as we did before with cp command

```
$ mv *.sh targetdir
```

In this case we moved all the file ending with .sh to a targetdir

→ **Wildcards**

These special characters are called wildcards. Wildcards allow you to select filenames based on patterns of characters. We will list some wildcards and what they select:

- ***** : Matches any characters
- **?** : Matches any single character
- **[characters]** : Matches any character that is a member of the set characters. The set of characters may also be expressed as a POSIX character class such as one of the following:
- **[!characters]** : Matches any character that is not a member of the set characters

Posix Character Classes :

- **[:alnum:]** Alphanumeric characters
- **[:alpha:]** Alphabetic characters
- **[:digit:]** Numerals
- **[:upper:]** Uppercase alphabetic characters
- **[:lower:]** Lowercase alphabetic characters

Using wildcards, it is possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

- ***** : all filenames
- ***.py** : all filenames that end with .py
- **g*** : all the filenames that begin with g
- **b*.txt** : all the filenames that start with b and end with .txt
- **DATA???** : any file that starts with "DATA" followed by exactly 3 more characters
- **[abc]*** : any filename that begins with "a" or "b" or "c" followed by any other characters
- **[:upper:]*** : Any filename that begins with an uppercase letter. This is an example of a character class.
- **BACKUP.[[:digit:]][[:digit:]]** : Another example of character classes. This pattern matches any filename that begins with the characters "BACKUP." followed by exactly two numerals.
- ***[![:lower:]]** : Any filename that does not end with a lowercase letter.

We can use wildcards with any command that accepts filename arguments.

C. Bulletin and commands(type command and which)

The type command is used to describe how its argument would be translated if used as commands. It is also used to find out whether it is built-in or external binary file

A shell builtin is nothing but command or a function, called from a shell, that is executed directly in the shell itself. The bash shell executes the command directly, without invoking another program. You can view information for Bash built-ins with help command. There are different types of built-in commands.

Lets see some examples:

```
themis@Ahmed-Belhaj:~/course$ type ls # we will look into alias in other chapters
ls is aliased to `ls --color=auto'
themis@Ahmed-Belhaj:~/course$ type cd
cd is a shell builtin
themis@Ahmed-Belhaj:~/course$ type cp
cp is hashed (/usr/bin/cp)
themis@Ahmed-Belhaj:~/course$ type mv
mv is hashed (/usr/bin/mv)
themis@Ahmed-Belhaj:~/course$ type touch
touch is hashed (/usr/bin/touch)
themis@Ahmed-Belhaj:~/course$ type mkdir
mkdir is hashed (/usr/bin/mkdir)
```

“which” Linux which command is used to identify the location of a given executable that is executed when you type the executable name (command) in the terminal prompt. The command searches for the executable specified as an argument in the directories listed in the PATH environment variable. (we will talk about environment variable later)

Let”s see some examples:

```
themis@Ahmed-Belhaj:~/course$ which ls
/usr/bin/ls
themis@Ahmed-Belhaj:~/course$ which cp
/usr/bin/cp
```

Let's see some other examples , let's say you are not sure if you have python3 installed you can check using which command .

```
themis@Ahmed-Belhaj:~/course$ which python3
/usr/bin/python3
```

2. Shell permission

On a Linux system, each file and directory is assigned access rights for the owner of the file, the members of a group of related users, and everybody else. Rights can be assigned to read a file, to write a file, and to execute a file.

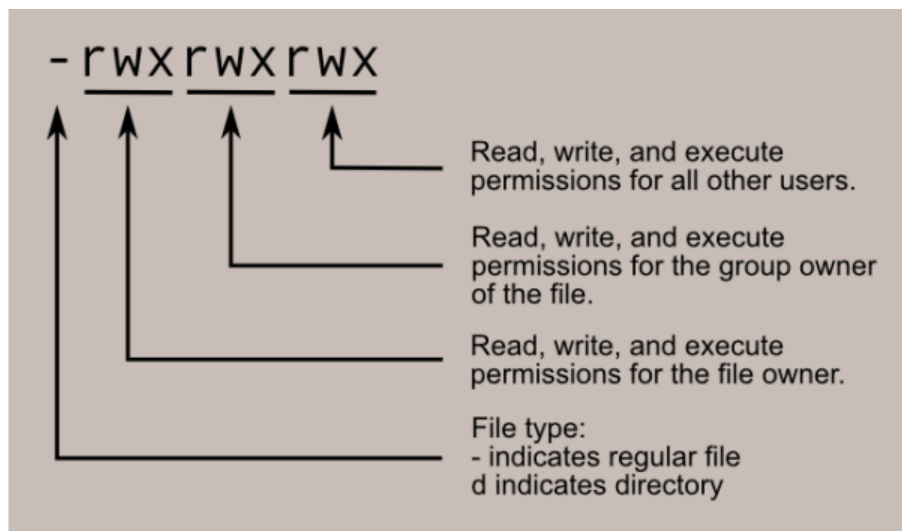
To see the permission settings for a file, we can use the `ls` command. As an example, we will look at the `bash` program which is located in the `/bin` directory:

```
themis@Ahmed-Belhaj:~$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1183448 Jun 18 2020 /bin/bash
```

Here we can see:

- The file `/bin/bash` is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

In the diagram below, we see how the first portion of the listing is interpreted. It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.



A. Changing permissions

The `chmod` command is used to change the permissions of a file or directory. To use it, we specify the desired permission settings and the file or files that we wish to modify. There are two ways to specify the permissions. In this lesson we will focus on one of

these, called the octal notation method. It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works

```
rwX  rwX  rwX  = 111 111 111
rw-  rw-  rw-  = 110 110 110
rwX  ---  ---  = 111 000 000
and so on...
rwX  = 111 in binary = 7
rw-  = 110 in binary = 6
r-x  = 101 in binary = 5
r--  = 100 in binary = 4
```

Now, if we represent each of the three sets of permissions (owner, group, and other) as a single digit, we have a pretty convenient way of expressing the possible permissions settings. For example, if we wanted to set some_file to have read and write permission for the owner, but wanted to keep the file private from others, we would:

```
$ chmod 600 some_file
```

Here is a permission number that covers all the common settings. The ones beginning with "7" are used with programs (since they enable execution) and the rest are for other kinds of files

- **777 : (rwxrwxrwx)** No restrictions on permissions. Anybody may do anything. Generally not a desirable setting
- **755 : (rwxr-xr-x)** The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
- **700 : (rwx-----)** The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
- **666 : (rw-rw-rw-)** All users may read and write the file
- **644 : (rw-r--r--)** The owner may read and write a file, while all others may
- only read the file. A common setting for data files that everybody
- may read, but only the owner may change
- **600 : (rw-----)** The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Now let's head back to our files in course directory and make some changes to our files permission using chmod .

```
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rw-r--r-- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod u+x first_script.sh
```

```

themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxr--r-- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod 777 first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxrwx 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod o=r first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxr-- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod g-r first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwx-wxr-- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod a-r first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
--wx-wx--- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
--wx-wx--- 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chmod a=rwx first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxrwx 1 themis themis 0 Jan 12 12:20 first_script.sh

```

In the up top examples i showed you the most common way to change file permission , first example we u+x , which is add execution to user , U is for user , g for group , o for other , a is for all ,you can either add by + or remove by - or set with = , and you can also use digits instead like chmod 700

B. Directory Permissions

The chmod command can also be used to control the access permissions for directories. Again, we can use the octal notation to set permissions, but the meaning of the r, w, and x attributes is different: • r - Allows the contents of the directory to be listed if the x attribute is also set. • w - Allows files within the directory to be created, deleted, or renamed if the x attribute is also set. • x - Allows a directory to be entered (i.e. cd dir).

C. Becoming the Superuser for a Short While

It is often necessary to become the superuser to perform important system administration tasks, but as we know, we should not stay logged in as the superuser. In most distributions, there is a program that can give you temporary access to the superuser's privileges. This program is called su (short for substitute user) and can be used in those cases when you need to be the superuser for a small number of tasks. To

become the superuser, simply type the su command. You will be prompted for the superuser's password:

```
themis@Ahmed-Belhaj:~$  
Password:  
root@Ahmed-Belhaj#
```

After executing the su command, we have a new shell session as the superuser. To exit the superuser session, type exit and we will return to your previous session.

In most modern distributions, an alternate method is used. Rather than using su, these systems employ the sudo command instead. With sudo, one or more users are granted superuser privileges on an as needed basis. To execute a command as the superuser, the desired command is simply preceded with the sudo command. After the command is entered, the user is prompted for their own password rather than the superuser's:

```
themis@Ahmed-Belhaj:~$ sudo apt install nano  
[sudo] password for themis:  
Reading package lists... Done  
...
```

In this case we used sudo so we can install Text editor nano

In fact, modern distributions don't even set the root account password thus making it impossible to log in as the root user. A root shell is still possible with sudo by using the "-i" option:

```
themis@Ahmed-Belhaj:~$ su  
Password:  
su: Authentication failure  
themis@Ahmed-Belhaj:~$ sudo -i  
root@Ahmed-Belhaj:~#
```

D. Changing File Ownership

We can change the owner of a file by using the chown command. Here's an example: Suppose we wanted to change the owner of some_file from "me" to "you". We could:

```
themis@Ahmed-Belhaj:~$ sudo chown you some_file
```

Let's see an example


```

themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxrwx 1 themis themis 0 Jan 12 12:20 first_script.sh
themis@Ahmed-Belhaj:~/course$ chown root first_script.sh
chown: changing ownership of 'first_script.sh': Operation not permitted
themis@Ahmed-Belhaj:~/course$ sudo chown root first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxrwx 1 root themis 0 Jan 12 12:20 first_script.sh

```

We can see with `ls -l` that user was themis and was changed to root. Notice that in order to change the owner of a file, we must have superuser privileges. To do this, our example employed the `sudo` command to execute `chown`. `chown` works the same way on directories as it does on files.

E. Changing group permission

The `chown` command changes the owner of a file, and we can use the `chgrp` command to change the group. On Linux, only root can use `chown` for changing ownership of a file, but any user can change the group to another group he belongs to. We will try changing the group owner for `first_script.sh` to root also :

```

themis@Ahmed-Belhaj:~/course$ sudo chgrp root first_script.sh
themis@Ahmed-Belhaj:~/course$ ls -l first_script.sh
-rwxrwxrwx 1 root root 0 Jan 12 12:20 first_script.sh

```

You can also change group permission recursively using the `-R` option. Changing directory group permission is the same.

```

themis@Ahmed-Belhaj:~/course$ sudo chgrp -R root bash/
themis@Ahmed-Belhaj:~/course$ ls -l
total 12
drwxr-xr-x 2 themis root 4096 Jan 12 12:32 bash
-rwxrwxrwx 1 root root 0 Jan 12 12:20 first_script.sh
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
themis@Ahmed-Belhaj:~/course$ cd bash/
themis@Ahmed-Belhaj:~/course/bash$ ls -l
total 0
-rw-r--r-- 1 themis root 0 Jan 11 16:01 git_setup

```

F. Groups and user

id command in Linux is used to find out user and group names and numeric ID's (UID or group ID) of the current user or any other user in the server. This command is useful to find out the following information as listed below:

- User name and real user id.
- Find out the specific Users UID.
- Show the UID and all groups associated with a user.
- List out all the groups a user belongs to.
- Display security context of the current user.

```
themis@Ahmed-Belhaj:~/course/bash$ id
uid=1000(themis) gid=1000(themis)
groups=1000(themis),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),117(netdev),1001(docker)
```

As well you can use groups to list them

```
themis@Ahmed-Belhaj:~/course/bash$ groups
themis adm dialout cdrom floppy sudo audio dip video plugdev netdev docker
themis@Ahmed-Belhaj:~/course/bash$ groups themis
themis : themis adm dialout cdrom floppy sudo audio dip video plugdev
netdev docker
themis@Ahmed-Belhaj:~/course/bash$
```

And you can pass a user to it and it will display the group which the user belongs to

Also we can use the command whoami to display the current user

```
themis@Ahmed-Belhaj:~/course/bash$ whoami
themis
```

G. Adding new user and group

When of other common command when you are working as system administrator or devops filed it will be adding users and groups

The adduser command creates a new user and additional information about the user, directories, and a password. Depending on the Command Line options and the given parameters, additional elements can be added. Its syntax is given below:

```
$ adduser -- options arguments
```

Only root user can add user without asking for permission for other you need to supply it with sudo

```
themis@Ahmed-Belhaj:~/course/bash$ adduser new_user

themis@Ahmed-Belhaj:~/course/bash$ sudo adduser new_user
Adding user `new_user' ...
Adding new group `new_user' (1002) ...
Adding new user `new_user' (1001) with group `new_user' ...
Creating home directory `/home/new_user' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for new_user
Enter the new value, or press ENTER for the default
    Full Name []: Ahmed Belhaj
    Room Number []: 2
    Work Phone []:
    Home Phone []:
    Other []:
Is the information correct? [Y/n] Y
```

And we also can add new group by addgroup command

```
themis@Ahmed-Belhaj:~/course/bash$ sudo addgroup new_group
Adding group `new_group' (GID 1003) ...
Done.
```

3. I/O Redirection and Filters

In this lesson, we will explore a powerful feature used by command line programs called input/output redirection. As we have seen, many commands such as ls print their output on the display. This does not have to be the case, however. By using some special notations we can redirect the output of many commands to files, devices, and even to the input of other commands.

A. Standard Output

Most command line programs that display their results do so by sending their results to a facility called standard output. By default, standard output directs its contents to the display. To redirect standard output to a file, the ">" character is used like this:

```
themis@Ahmed-Belhaj:~/course$ ls > listing.txt
themis@Ahmed-Belhaj:~/course$ cat listing.txt
bash
first_script.sh
git
github
listing.txt
```

In this example, the `ls` command is executed and the results are written in a file named `listing.txt`. Since the output of `ls` was redirected to the file, no results appear on the display

We can see when we are concatenating the file using `cat` command we can see the result saved in `listing.txt`

Each time the command above is repeated, `file_list.txt` is overwritten from the beginning with the output of the command `ls`. To have the new results appended to the file instead, we use `">>"` like this:

```
themis@Ahmed-Belhaj:~/course$ ls -l >> listing.txt
themis@Ahmed-Belhaj:~/course$ cat listing.txt
bash
first_script.sh
git
github
listing.txt
total 16
drwxr-xr-x 2 themis root 4096 Jan 12 12:32 bash
-rwxrwxrwx 1 root root 0 Jan 12 12:20 first_script.sh
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
-rw-r--r-- 1 themis themis 44 Jan 13 14:08 listing.txt
```

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated. If the file does not exist when we attempt to append the redirected output, the file will be created.

B. Standard Input

Many commands can accept input from a facility called standard input. By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected. To redirect standard input from a file instead of the keyboard, the `"<"` character is used like this:

```

themis@Ahmed-Belhaj:~/course$ sort < listing.txt
-rw-r--r-- 1 themis themis  44 Jan 13 14:08 listing.txt
-rwxrwxrwx 1 root    root    0 Jan 12 12:20 first_script.sh
bash
drwxr-xr-x 2 themis root    4096 Jan 12 12:32 bash
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
first_script.sh
git
github
listing.txt
total 16

```

In the example above, we used the sort command to process the contents of listing.txt. The results are output on the display since the standard output was not redirected. We could redirect standard output to another file like this:

```

themis@Ahmed-Belhaj:~/course$ sort < listing.txt > sorted_listing.txt
themis@Ahmed-Belhaj:~/course$ cat sorted_listing.txt
-rw-r--r-- 1 themis themis  44 Jan 13 14:08 listing.txt
-rwxrwxrwx 1 root    root    0 Jan 12 12:20 first_script.sh
bash
drwxr-xr-x 2 themis root    4096 Jan 12 12:32 bash
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
first_script.sh
git
github
listing.txt
total 16

```

As we can see, a command can have both its input and output redirected. Be aware that the order of the redirection does not matter. The only requirement is that the redirection operators (the "<" and ">") must appear after the other options and arguments in the command.

A. Filters

One kind of program frequently used in pipelines is called a filter. Filters take standard input and perform an operation upon it and send the results to standard output. In this way, they can be combined to process information in powerful ways. Here are some of the common programs that can act as filters

→ <<grep>> command

Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.

grep examples:

Display lines containing the pattern “themis” from the file listing.txt

```
themis@Ahmed-Belhaj:~/course$ grep "themis" listing.txt
drwxr-xr-x 2 themis root 4096 Jan 12 12:32 bash
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
-rw-r--r-- 1 themis themis 44 Jan 13 14:08 listing.txt
```

Display the number of lines that contain the pattern “bin” in the file /etc/passwd :

```
themis@Ahmed-Belhaj:~/course$ grep "bin" /etc/passwd | wc -l
33
```

PS : we will discuss how to multiple commands using pipes later

wc stands for word count. As the name implies, it is mainly used for counting purposes.

Resources:

- **wc Command**
- **cat command**

. Display lines containing the pattern “root” and 3 lines after them in the file /etc/passwd:

```
themis@Ahmed-Belhaj:~/course$ grep -A 3 "root" /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Display all the lines in the file /etc/passwd that do not contain the pattern “less”:

```
themis@Ahmed-Belhaj:~/course$ grep -v 'less' /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```

```
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network
Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd
Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:104:systemd Time
Synchronization,,,:/run/systemd:/usr/sbin/nologin
```

→ <<head>> command

Outputs the first few lines of its input. Useful for getting the header of a file.

By default, 'head' command reads the first 10 lines of the file. If you want to read more or less than 10 lines from the beginning of the file then you have to use the '-n' option with 'head' command. head [option] [filename]...[filename] Using option in 'head' command is optional. You can apply the 'head' command for one or more files. 1.

Display the first 10 lines of /etc/passwd :

```
themis@Ahmed-Belhaj:~/course$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Head Examples :

Display the first 5 lines of /etc/passwd :

```
themis@Ahmed-Belhaj:~/course$ head -n 5 /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

‘-n’ option with 5 is used in the following ‘head’ command. The first five lines of the /etc/passwd file will be shown in the output. 3. Omit some lines: You can use negative value with ‘-n’ option in ‘head’ command if you want to omit some lines from the file. The following command will omit the last 7 lines from /etc/passwd .

```
themis@Ahmed-Belhaj:~/course$ head -n -7 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
...
```

→ <<tail>> command

Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file

By default, ‘tail’ command reads the last 10 lines of the file. If you want to read more or less than 10 lines from the ending of the file then you have to use the ‘-n’ option with the ‘tail’ command. tail [option] [filename]...[filename] Like ‘head’ command ‘tail’ command is also applicable for multiple files and using option is optional for ‘tail’ command.

```
themis@Ahmed-Belhaj:~/course$ tail /etc/passwd
_apt:x:105:65534:./nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:112:./run/uuidd:/usr/sbin/nologin
tcpdump:x:108:113:./nonexistent:/usr/sbin/nologin
sshd:x:109:65534:./run/sshd:/usr/sbin/nologin
landscape:x:110:115:./var/lib/landscape:/usr/sbin/nologin
pollinate:x:111:1:./var/cache/pollinate:/bin/false
themis:x:1000:1000:,,,:/home/themis:/bin/bash
mysql:x:112:119:MySQL Server,,,:/nonexistent:/bin/false
new_user:x:1001:1002:Ahmed Belhaj,2,,:/home/new_user:/bin/bash
```

Tail Examples

Display the last 2 lines of /etc/passwd

```
themis@Ahmed-Belhaj:~/course$ tail -n 2 /etc/passwd
mysql:x:112:119:MySQL Server,,,:/nonexistent:/bin/false
new_user:x:1001:1002:Ahmed Belhaj,2,,:/home/new_user:/bin/bash
```

When you want to read particular lines from the ending of the file then you have to use ‘-n’ option with positive value. The following command will display the last 2 lines of /etc/passwd file.

Omit some lines:

If you want to omit the specific lines from the beginning then you have to use ‘-n’ option with negative value in ‘tail’ command. The following command will display the content of /etc/passwd file by omitting 3 lines from the beginning.

```
themis@Ahmed-Belhaj:~/course$ tail -n -3 /etc/passwd
themis:x:1000:1000:,,,:/home/themis:/bin/bash
mysql:x:112:119:MySQL Server,,,:/nonexistent:/bin/false
new_user:x:1001:1002:Ahmed Belhaj,2,,:/home/new_user:/bin/bash
```

Using ‘head’ and ‘tail’ commands together : If you want to read the content from the middle of any file then only ‘head’ or ‘tail’ command can’t solve this problem. You have to use both ‘head’ and ‘tail’ commands together to solve this problem. The following command will read lines from 2 to 6 of /etc/passwd file. At first, ‘head’ command will retrieve first 6 lines by omitting the last 5 lines for negative value and ‘tail’ command will retrieve the last 5 line from the output of ‘head’ command.

```
themis@Ahmed-Belhaj:~/course$ head -n 5 /etc/passwd | tail -n 5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Some of the commands that you need to look into :

A. echo - display a line of text

```
themis@Ahmed-Belhaj:~/course$ echo "hello"
hello
```

B. cat - concatenate files and print on the standard output

```
themis@Ahmed-Belhaj:~/course$ cat listing.txt
bash
```

```

first_script.sh
git
github
listing.txt
total 16
drwxr-xr-x 2 themis root 4096 Jan 12 12:32 bash
-rwxrwxrwx 1 root root 0 Jan 12 12:20 first_script.sh
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
-rw-r--r-- 1 themis themis 44 Jan 13 14:08 listing.txt

```

C. find - search for files in a directory hierarchy

```

themis@Ahmed-Belhaj:~/course$ find /etc/
/etc/
/etc/services
/etc/lsb-release
/etc/vulkan
/etc/vulkan/explicit_layer.d
/etc/vulkan/icd.d
/etc/vulkan/implicit_layer.d
/etc/protocols
/etc/rsyslog.conf
...

```

D. wc - print newline, word, and byte counts for each file

```

themis@Ahmed-Belhaj:~/course$ wc listing.txt
11 52 322 listing.txt

```

E. sort - sort lines of text files

```

themis@Ahmed-Belhaj:~/course$ sort listing.txt
-rw-r--r-- 1 themis themis 44 Jan 13 14:08 listing.txt
-rwxrwxrwx 1 root root 0 Jan 12 12:20 first_script.sh
bash
drwxr-xr-x 2 themis root 4096 Jan 12 12:32 bash
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
first_script.sh
git
github
listing.txt
total 16

```

F. uniq - report or omit repeated lines

```
themis@Ahmed-Belhaj:~/course$ uniq /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
```

G. tr - translate or delete characters

```
themis@Ahmed-Belhaj:~/course$ echo "hello" | tr "h" "x"
xello
themis@Ahmed-Belhaj:~/course$ echo "hello" | tr -d "h"
ello
```

H. rev - reverse lines character wise

```
themis@Ahmed-Belhaj:~/course$ echo "hello" | rev
olleh
```

Resources :

B. Pipes

Pipe is a form of redirection (transfer of standard output to some other destination) that is used in linux and unix alike operating system to send output of one command/program/process to another

Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen.

Pipes are unidirectional i.e data flows from left to right through the pipeline.

```
command_1 | command_2 | command_3 | .... | command_N
```

Example:

```
themis@Ahmed-Belhaj:~/course$ cat sorted_listing.txt | grep "themis"
-rw-r--r-- 1 themis themis  44 Jan 13 14:08 listing.txt
drwxr-xr-x 2 themis root   4096 Jan 12 12:32 bash
drwxr-xr-x 2 themis themis 4096 Jan 12 12:14 github
drwxr-xr-x 2 themis themis 4096 Jan 12 12:33 git
```

A pipeline is a sequence of one or more commands separated by one of the control operators '|' or '|&'.
The output of each command in the pipeline is connected via a pipe to the input of the next command. That is, each command reads the previous command's output. This connection is performed before any redirections specified by the command.

If '|&' is used, command1's standard error, in addition to its standard output, is connected to command2's standard input through the pipe; it is shorthand for 2>&1 |. This implicit redirection of the standard error to the standard output is performed after any redirections specified by the command.

```
themis@Ahmed-Belhaj:~/course$ ls ; echo "hello"
bash first_script.sh git github listing.txt sorted_listing.txt
hello
```

Resources:

<https://www.gnu.org/software/bash/manual/bash.html#Pipelines>

4. Shell init files , variable and expansion

A. Alias

Do you often find yourself typing a long command on the command line or searching the bash history for a previously typed command? If your answer to any of those questions is yes, then you will find bash aliases handy. Bash aliases allow you to set a memorable shortcut command for a longer command.

Thats where alias comme in

→ *Creating Alias*

Creating aliases in bash is very straight forward. The syntax is as follows:

```
$ alias alias_name="command_to_run"
```

An alias declaration starts with the alias keyword followed by the alias name, an equal sign and the command you want to run when you type the alias. The command needs to be enclosed in quotes and with no spacing around the equal sign. Each alias needs to be declared on a new line.

The ls command is probably one of the most used commands on the Linux command line. I usually use this command with the -la switch to list out all files and directories, including the hidden ones in long list format.

Let's create a simple bash alias named ll which will be a shortcut for the ls -la command. To do so type open a terminal window and type:

```
themis@Ahmed-Belhaj:~/course$ alias ll="ls -la"
themis@Ahmed-Belhaj:~/course$ ll
total 24
drwxr-xr-x  5 themis themis 4096 Jan 13 16:01 .
drwxr-xr-x 18 themis themis 4096 Jan 13 15:27 ..
drwxr-xr-x  2 themis root   4096 Jan 12 12:32 bash
-rwxrwxrwx  1 root  root    0 Jan 12 12:20 first_script.sh
drwxr-xr-x  2 themis themis 4096 Jan 12 12:33 git
drwxr-xr-x  2 themis themis 4096 Jan 12 12:14 github
-rw-r--r--  1 themis themis  0 Jan 13 15:02 listing.txt
-rw-r--r--  1 themis themis 322 Jan 13 14:17 sorted_listing.txt
```

the ll alias will be available only in the current shell session. If you exit the session or open a new session from another terminal, the alias will not be available.

To make the alias persistent you need to declare it in the ~/.bash_profile or ~/.bashrc file.

~/.bashrc file

```
# Aliases
# alias alias_name="command_to_run"

# Long format list
```

```
alias ll="ls -la"

# Print my public IP
alias myip='curl ipinfo.io/ip'
```

B. Expansion

Each time we type a command line and press the enter key, bash performs several processes upon the text before it carries out our command. We have seen a couple of cases of how a simple character sequence, for example “*”, can have a lot of meaning to the shell. The process that makes this happen is called expansion. With expansion, we type something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the [echo](#) command. echo is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```
themis@Ahmed-Belhaj:~/course$ echo *
bash first_script.sh git github listing.txt sorted_listing.txt
```

→ Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allow us to use the shell prompt as a calculator:

```
themis@Ahmed-Belhaj:~/course$ echo $((2 + 2))
4
```

Arithmetic expansion uses the form:

```
$((expression))
```

where expression is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations.

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply five squared by three:

```
themis@Ahmed-Belhaj:~/course$ echo $((($((5**2)) * 3))
75
```

→ *Brace Expansion*

Perhaps the strangest expansion is called brace expansion. With it, we can create multiple text strings from a pattern containing braces. Here's an example:

```
themis@Ahmed-Belhaj:~/course$ echo Front-{A,B,C}-Back  
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a preamble and a trailing portion called a postscript. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
themis@Ahmed-Belhaj:~/course$ echo Number_{1..5}  
Number_1 Number_2 Number_3 Number_4 Number_5
```

→ *Parameter Expansion*

We're only going to touch briefly on parameter expansion in this lesson, but we'll be covering it more later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called variables, are available for our examination. For example, the variable named "USER" contains our user name. To invoke parameter expansion and reveal the contents of USER we would do this:

```
themis@Ahmed-Belhaj:~/course$ echo $USER  
themis
```

To see a list of available variables, try this:

```
themis@Ahmed-Belhaj:~/course$ printenv | less
```

With other types of expansion, if we mistype a pattern, the expansion will not take place and the echo command will simply display the mistyped pattern. With parameter expansion, if we misspell the name of a variable, the expansion will still take place, but will result in an empty string:

```
themis@Ahmed-Belhaj:~/course$ echo $SUER
```

Resources:

https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html

Advanced Shell Scripting

1. Variable Assignment

Variables are named symbols that represent either a string or numeric value. When you use them in commands and expressions, they are treated as if you had typed the value they hold instead of the name of the variable. To create a variable, you just provide a name and value for it. Your variable names should be descriptive and remind you of the value they hold. A variable name cannot start with a number, nor can it contain spaces. It can, however, start with an underscore. Apart from that, you can use any mix of upper- and lowercase alphanumeric characters. Here, we'll create five variables. The format is to type the name, the equals sign =, and the value. Note there isn't a space before or after the equals sign. Giving a variable a value is often referred to as assigning a value to the variable. We'll create four string variables and one numeric variable:

```
themis@Ahmed-Belhaj:~/course$ name="ahmed"
themis@Ahmed-Belhaj:~/course$ echo $name
ahmed
```

To see the value held in a variable, use the echo command. You must precede the variable name with a dollar sign \$ whenever you reference the value it contains, as shown above

We'll talk about quoting variables later. For now, here are some things to remember:

- A variable in single quotes ' is treated as a literal string, and not as a variable.
- Variables in quotation marks " are treated as variables.
- To get the value held in a variable, you have to provide the dollar sign \$.
- A variable without the dollar sign \$ only provides the name of the variable.

```
$ echo '$name'
$my_name
$ echo "$name"
ahmed
```



```
$ echo name
my_name $ echo $my_name
ahmed
```

Bash Arithmetic Operators The Bash shell has a large list of supported arithmetic operators to do math calculations, as shown below:

Arithmetic Operator	Description
id++, id--	variable post-increment, post-decrement
++id, --id	variable pre-increment, pre-decrement
-, +	unary minus, plus
!, ~	logical and bitwise negation
**	exponentiation
*, /, %	multiplication, division, remainder (modulo)
+, -	addition, subtraction
«, »	left and right bitwise shifts
<=, >=, <, >	comparison
=, !=	equality, inequality
&	bitwise AND
^	bitwise XOR
	bitwise OR
&&	logical AND
	logical OR
expression ? expression : expression	conditional operator
=, *=, /=, %=, +=, -=, «=, »=, &=, ^=, =	assignment

A. Get inputs from user

So let's say that we will create a bash script file names script.sh to be executed. We must create an empty file and then we make it executable and add the shebang at the first line of the file and then you can start write your code.

```
themis@Ahmed-Belhaj:~/course$ touch script.sh #creating an empty file
themis@Ahmed-Belhaj:~/course$ chmod +x script.sh #make the file executable
```

You can now open the file using a text editor or IDE to add the shebang and your bash script. I will use nano to open the file and add the shebang and save.

```
themis@Ahmed-Belhaj:~/course$ nano script.sh
```

Shebang: #!/bin/bash

It tells the computer which type of interpreter to use for the script. So the content of the file script.sh will be like this:

```
themis@Ahmed-Belhaj:~/course$ cat script.sh
#!/bin/bash
{your bash script here}
```

We can execute our script using this command:

```
themis@Ahmed-Belhaj:~/course$ ./script.sh
```

Now let's back to the main idea of getting input from user, there is two ways to get input from user:

- Command line arguments
- Interactive input

B. Command line arguments

Command line arguments are commonly used and easy to work with so they are a good place to start. When we run a program on the command line you would be familiar with supplying arguments after it to control its behaviour. For instance we could run the command `ls -l /etc`. `-l` and `/etc` are both command line arguments to the command `ls`. We can do similar with our bash scripts. To do this we use the variables `$1` to represent the first command line argument, `$2` to represent the second command line argument and so on. These are automatically set by the system when we run our script so all we need to do is refer to them. Let's look at an example.

```
#!/bin/bash
echo This bash script make the addition of two variables
echo we will add the value $1 to the value $2
sum1=$(( $1 + $2 ))
sum2=`expr $1 + $2`
echo The sum is $sum1 we didn't use expr command
echo The sum is $sum2 using expr command
```

let's execute this script and see the output that we will get we must add two arguments exactly because we used \$1 and \$2 in our script:

```
themis@Ahmed-Belhaj:~/course$ ./script.sh 5 7
This bash script make the addition of two variables
we will add the value 5 to the value 7
The sum is 12 we didn't use expr command
The sum is 12 using expr command
```

It is important to note that in the example above we used the command echo simply because it is a convenient way to demonstrate that the variables have actually been set. echo is not needed to make use of variables and is only used when you wish to print a specific message to the screen

C. Interactive input

We looked at one form of user input (command line arguments) in the previous section. Now we would like to introduce other ways the user may provide input to the Bash script. Following this we'll have a discussion on when and where is best to use each method. After the mammoth previous section this one is much easier to get through. If we would like to ask the user for input then we use a command called read. This command takes the input and will save it into a variable. Let's make the same example using the interactive way:

```
#!/bin/bash
echo This bash script make the addition of two variables
read -p `enter the value number 1` var1
read -p `enter the value number 2` var2
echo we will add the value $var1 to the value $var2
sum1=$(( $var1 + $var2 ))
sum2=`expr $var1 + $var2`
echo The sum is $sum1 we didn't use expr command
echo The sum is $sum2 using expr command
```

let's execute this script and see the output that we will get

```
themis@Ahmed-Belhaj:~/course$ ./script.sh
This bash script make the addition of two variables
enter the value number 1: 5
enter the value number 2: 7
we will add the value 5 to the value 7
The sum is 12 we didn't use expr command
The sum is 12 using expr command$ ./script.sh
This bash script make the addition of two variables
```

```
enter the value number 1: 5
enter the value number 2: 7
we will add the value 5 to the value 7
The sum is 12 we didn't use expr command
The sum is 12 using expr command
```

The general mechanism is that you can supply several variable names to read. Read will then take your input and split it on whitespace. The first item will then be assigned to the first variable name, the second item to the second variable name and so on. If there are more items than variable names then the remaining items will all be added to the last variable name. If there are less items than variable names then the remaining variable names will be set to blank or null

2. Shell Conditions

A. If Statement

Bash if statements are very useful. In this section of our Bash Scripting course you will learn the ways you may use if statements in your Bash scripts to help automate tasks.

If statements (and, closely related, case statements) allow us to make decisions in our Bash scripts. They allow us to decide whether or not to run a piece of code based upon conditions that we may set. If statements, combined with loops (which we'll look at in the next section) allow us to make much more complex scripts which may solve larger tasks.

A basic if statement effectively says, if a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. It follows the format below:

```
if [ <some test> ]
then
  <command>
fi
```

Anything between then and fi (if backwards) will be executed only if the test (between the square brackets) is true. Let's look at a simple example:

```
themis@Ahmed-Belhaj:~/course$ ./script.sh 10
That's Great
themis@Ahmed-Belhaj:~/course$ ./script.sh 100
```

```
The number 100 is greater than 10
/home/themis/course
That's Great
themis@Ahmed-Belhaj:~/course$
```

And now let's break it down:

```
if [ $1 -gt 10 ] # see if the first command line argument is greater than
10
```

every command in between the then and fi will only get run if statement returns true. You can have as many commands here as you like. fi signals the end of the if statement. All commands after this will be run as normal.

```
echo That\'s Great
```

Because this command is outside the if statement it will be run regardless of the outcome of the if statement. The backslash (\) in front of the single quote (') is needed as the single quote has a special meaning for bash and we don't want that special meaning. The backslash escapes the special meaning to make it a normal plain single quote again. The square brackets ([]) in the if statement above are actually a reference to the command test. This means that all of the operators that test allows may be used here as well. Look up the man page for test to see all of the possible operators (there are quite a few) but some of the more common ones are listed below

Operator	Description
<code>! EXPRESSION</code>	The <code>EXPRESSION</code> is false.
<code>-n STRING</code>	The length of <code>STRING</code> is greater than zero.
<code>-z STRING</code>	The length of <code>STRING</code> is zero (ie it is empty).
<code>STRING1 = STRING2</code> <code>STRING1 != STRING2</code>	<code>STRING1</code> is equal to <code>STRING2</code> <code>STRING1</code> is not equal to <code>STRING2</code>
<code>INTEGER1 -eq INTEGER2</code>	<code>INTEGER1</code> is numerically equal to <code>INTEGER2</code>
<code>INTEGER1 -gt INTEGER2</code>	<code>INTEGER1</code> is numerically greater than <code>INTEGER2</code>
<code>INTEGER1 -lt INTEGER2</code>	<code>INTEGER1</code> is numerically less than <code>INTEGER2</code>
<code>-d FILE</code>	<code>FILE</code> exists and is a directory.
<code>-e FILE</code>	<code>FILE</code> exists.
<code>-r FILE</code>	<code>FILE</code> exists and the read permission is granted.
<code>-s FILE</code>	<code>FILE</code> exists and its size is greater than zero (ie. it is not empty).
<code>-w FILE</code>	<code>FILE</code> exists and the write permission is granted.
<code>-x FILE</code>	<code>FILE</code> exists and the execute permission is granted.

You'll notice that in the if statement above we indented the commands that were run if the statement was true. This is referred to as indenting and is an important part of writing good, clean code (in any language, not just Bash scripts). The aim is to improve readability and make it harder for us to make simple, silly mistakes. There aren't any rules regarding indenting in Bash so you may indent or not indent however you like and your scripts will still run exactly the same. I would highly recommend you do indent your code however (especially as your scripts get larger) otherwise you will find it increasingly difficult to see the structure in your scripts. Talking of indenting. Here's a perfect example of when it makes life easier for you. You may have as many if statements as necessary inside your script. It is also possible to have an if statement inside of another if statement. For example, we may want to analyse a number given on the command line like so:

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

```

if [ $1 -gt 10 ]
then
    echo The number $1 is greater than 10
    if (( $1 % 2 == 0 ))
    then
        echo And is also even number.
    fi
fi
echo That\'s Great

```

Lets run it

```

themis@Ahmed-Belhaj:~$ ./script.sh 21
The number 21 is greater than 10
That's Great
themis@Ahmed-Belhaj:~$ ./script.sh 12
The number 12 is greater than 10
And is also even number.
That's Great

```

You can nest as many if statements as you like but as a general rule of thumb if you need to nest more than 3 levels deep you should probably have a think about reorganising your logic

B. If else Statements

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the else mechanism.

```

if [ <some test> ]
then
    <commands>
else
    <other commands>
fi

```

Sometimes we may have a series of conditions that may lead to different paths.

```

if [ <some test> ]
then
    <commands>

```

```
elif [ <some test> ]
then
    <different commands>
else
    <other commands>
fi
```

Let's make 2 scripts examples using these two structures:

```
#!/bin/bash
if (( $1 % 2 == 0 ))
then
    echo $1 is an even number.
else
    echo $1 is an odd number.
fi
```

```
themis@Ahmed-Belhaj:~$ ./script.sh 21
21 is an odd number.
themis@Ahmed-Belhaj:~$ ./script.sh 22
22 is an even number.
```

```
#!/bin/bash
if [ $1 -gt 10 ]
then
    echo $1 is greater than 10.
elif [ $1 -lt 10 ]
then
    echo $1 is less than 10.
else
    echo $1 is equal to 10
fi
```

```
themis@Ahmed-Belhaj:~$ ./script.sh 21
21 is greater than 10.
```



```
themis@Ahmed-Belhaj:~$./script.sh 5
5 is less than 10.
themis@Ahmed-Belhaj:~$./script.sh 10
10 is equal to 10
```

Sometimes we only want to do something if multiple conditions are met. Other times we would like to perform the action if one of several condition is met. We can accommodate these with boolean operators. • and - && • or - ||

In the format below the statement will be True if the two conditions are Ture:

```
if [ <some test> ] && [ <some other test> ]
then
  <commands>
fi
```

In the format below the statement will be True if one of the two conditions is Ture:

```
if [ <some test> ] || [ <some other test> ]
then
  <command>
fi
```

C. Case Statement

Sometimes we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of if and elif statements but that would soon grow to be unwieldy. Fortunately there is a case statement which can make things cleaner. It's a little hard to explain so here are some examples to illustrate:

Here is a basic example:

```
#!/bin/bash
case $1 in
  start)
    echo starting
    ;;
  stop)
    echo stoping
    ;;
  restart)
    echo restarting
    ;;
  *)
```

```
echo don\'t know
;;
esac
```

Let's try to execute our script with different inputs:

```
themis@Ahmed-Belhaj:~$ ./script.sh start
starting
themis@Ahmed-Belhaj:~$ ./script.sh restart
restarting
themis@Ahmed-Belhaj:~$ ./script.sh stop
stopping
themis@Ahmed-Belhaj:~$ ./script.sh blah
don't know
```

Summary

- if

Perform a set of commands if a test is true.

- else

If the test is not true then perform a different set of commands.

- elif

If the previous test returned false then try this one.

- &&

Perform the and operation.

- ||

Perform the or operation.

- case

Choose a set of commands to execute depending on a string matching a particular pattern.

- Indenting

Indenting makes your code much easier to read. It gets increasingly important as your Bash scripts get longer.

- Planning

Now that your scripts are getting a little more complex you will probably want to spend a little bit of time thinking about how you structure them before diving in.

3. Shell Loops

Bash loops are very useful. In this section of our Bash Scripting course we'll look at the different loop formats available to us as well as discuss when and why you may want to

use each of them. Loops allow us to take a series of commands and keep re-running them until a particular situation is reached. They are useful for automating repetitive tasks. There are 3 basic loop structures in Bash scripting which we'll look at below. There are also a few statements which we can use to control the loop operation.

A. While loops

One of the easiest loops to work with is while loops. They say, while an expression is true, keep executing these lines of code. They have the following format:

```
While [ <some condition> ]  
do  
    <commands>  
done
```

In the example bellow we will print the number 1 through to 10:

```
#!/bin/bash  
  
counter=1  
while [ $counter -le 10 ]  
do  
    echo $counter  
    ((counter++))  
done  
echo All done
```

Let's try to execute our script

```
themis@Ahmed-Belhaj:~$ ./script.sh  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
All done
```

A common mistake is what's called an off by one error. In the example above we could have put -lt as opposed to -le (less than as opposed to less than or equal). Had we done

this it would have printed up until 9. These mistakes are easy to make but also easy to fix once you've identified it so don't worry too much if you make this error.

B. Until loops

The until loop is fairly similar to the while loop. The difference is that it will execute the commands within it until the test becomes true.

```
until [ <some condition> ]
do
  <command>
done
```

As you can see in the example above, the syntax is almost exactly the same as the while loop (just replace while with until). We can also create a script that does exactly the same as the while example above just by changing the test accordingly.

```
#!/bin/bash
counter=1
until [ $counter -gt 10 ]
do
  echo $counter
  ((counter++))
done
echo All done
```

C. For loops

The for loop is a little bit different to the previous two loops. What it does is say for each of the items in a given list, perform the given set of commands. It has the following syntax.

```
for var in <list>
do
  <command>
done
```

Here is a simple example to illustrate:

```
#!/bin/bash
names=`Stan Kyle Cartman`
for name in $names
do
  echo $name
```

```
done
echo All done
```

The for loop will take each item in the list (in order, one after the other), assign that item as the value of the variable var, execute the commands between do and done then go back to the top, grab the next item in the list and repeat over. The list is defined as a series of strings, separated by spaces. Let's see our output:

```
themis@Ahmed-Belhaj:~$ ./script.sh
Stan
Kyle
Cartman
All done
```

We can also process a series of numbers

```
#!/bin/bash
for value in {1..5}
do
echo $value
done
echo All done
```

Let's execute our script:

```
themis@Ahmed-Belhaj:~$ ./script.sh
1
2
3
4
5
All done
```

It's important when specifying a range like this that there are no spaces present between the curly brackets { }. If there are then it will not be seen as a range but as a list of items. When specifying a range you may specify any number you like for both the starting value and ending value. The first value may also be larger than the second in which case it will count down. It is also possible to specify a value to increase or decrease by each time. You do this by adding another two dots (..) and the value to step by.

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

```
for value in {10..0..2}
do
echo $value
done
echo All done
```

And now the execution

```
themis@Ahmed-Belhaj:~$ ./script.sh
10
8
6
4
2
0
All done
```

D. Controlling loops: Break and Continue

Most of the time your loops are going to through in a smooth and orderly manner. Sometimes however we may need to intervene and alter their running slightly. There are two statements we may issue to do this.

→ Break

The break statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well. For instance, maybe we are copying files but if the free disk space get's below a certain level we should stop copying.

In the script below, the execution of the while loop will be interrupted once the current iterated item is equal to 2:

```
#!/bin/bash
i=0
while [[ $i -lt 5]]
do
echo $i
((i++))
if [[ $i -eq 2]] then
break
```

```
fi
done
echo All done
```

The output of this script will be:

```
themis@Ahmed-Belhaj:~$ ./script.sh
0
1
All done
$
```

→ *Continue*

The continue statement tells Bash to stop running through this iteration of the loop and begin the next iteration. Sometimes there are circumstances that stop us from going any further. For instance, maybe we are using the loop to process a series of files but if we happen upon a file which we don't have the read permission for we should not try to process it.

In the example below, once the current iterated item is equal to 2, the continue statement will cause execution to return to the beginning of the loop and to continue with the next iteration.

```
#!/bin/bash
i=0
while [[ $i -lt 5]]
do
((i++))
if [[ $i -eq 2]] then
continue
fi
echo $i
done
echo All done
```

The output of this script will be:

```
themis@Ahmed-Belhaj:~$ ./script.sh
1
3
```

4

5

All done

Git - Getting Started

1. About Version Control

This chapter will be about getting started with Git. We will begin by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it set up to start working with. At the end of this chapter you should understand why Git is around .

Version control is at the center of any file based project. Whether you're a software developer, project manager, team member, student, or anyone who works on file based projects, keeping track of changes is essential to creating a great product. In this course, Getting Started with Git, you'll learn the popular version control system Git and why it plays a significant role in creating better projects. First, you'll learn the basics of understanding Git. Next, you'll start your journey exploring how Git elevates in a project in a fun and easy step-by-step experience. Finally, you'll discover common and extended commands used in Git everyday. When you're finished with this course, you'll have a working knowledge of Git as a version control system for your project.

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this curriculum , you will use some source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

2. What is Git?

Git is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems).

For example, you could be working on a website's landing page and discover that you do not like the navigation bar. But at the same time, you might not want to start altering its components because it might get worse.

With Git, you can create an identical copy of that file and play around with the navigation bar. Then, when you are satisfied with your changes, you can merge the copy to the original file. You are not limited to using Git just for source code files – you can also use it to keep track of text files or even images. This means that Git is not just for developers – anyone can find it helpful.

3. Installing Git

In order to use Git, you have to install it on your computer. To do this, you can download the latest version on the [official website](#). You can download for your operating system from the options given.

You can also install Git using the command line, but since the commands vary with each operating system, we'll focus on the more general approach.

4. First-Time Git Setup

I will assume that at this point you have installed Git. To verify this, you can run this command on the command line: `git --version`. This shows you the current version installed on your PC. The next thing you'll need to do is to set your username and email address. Git will use this information to identify who made specific changes to files.

To set your username, type and execute these commands: `git config --global user.name "YOUR_USERNAME"` and `git config --global user.email "YOUR_EMAIL"`. Just make sure to replace "YOUR_USERNAME" and "YOUR_EMAIL" with the values you choose.

```
themis@Ahmed-Belhaj:~$ git --version
git version 2.25.1
themis@Ahmed-Belhaj:~$ git config --global user.name "Theemiss"
themis@Ahmed-Belhaj:~$ git config --global user.email
"test@gmail.com"
themis@Ahmed-Belhaj:~$
```

A quick aside: git and GitHub are not the same thing. Git is an open-source, version control tool created in 2005 by developers working on the Linux operating system; GitHub is a company founded in 2008 that makes tools which integrate with git. You do not need GitHub to use git, but you cannot use GitHub without using git. There are many other alternatives to GitHub, such as GitLab, BitBucket, and “host-your-own” solutions such as gogs and gittea. All of these are referred to in git-speak as “remotes”, and all are completely optional. You do not need to use a remote to use git, but it will make sharing your code with others easier.

- Resources :
 - **What is Github**
 - **Version Control**

- **Git vs Github**
- **Github Alternative**
- **Why Git**

Git Basics

1. Using git with GitHub

To use git with github first you need to create an account here, [Github](#), github use repository .In revision control systems, a repository is a data structure that stores metadata for a set of files or directory structure

You need to create a repo (Repository). Let's start by creating a public repository , First_lessons.it should be public , without readme and no gitignore(we will talk about them later).a public repository is viewed by everyone, and he will able to clone and use it , but he can't add something on your repository unless you give permission.a private repository is accessed and viewed only by you , unless you added people .

Visite this and let's start working [New Repository GitHub](#) , or head up to github website and find the create new repository button

Owner ^{*} Theemiss / Repository name ^{*} first_lessons ✓

Great repository names are short and memorable. Need inspiration? How about **sturdy-guacamole**?

Description (optional)
first git lesson

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

- ☐ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)
- ☐ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)
- ☐ **Choose a license**
A license tells others what they can and can't do with your code. [Learn more.](#)

Grant your Marketplace apps access to this repository
You are subscribed to 1 Marketplace app

- ☒ **CommitCheck**
CommitCheck ensures your commit messages are consistent and contain all required information

Create repository

Fill in the repository name, and the description and click on the green button “Create repository”. Do not initialize this repository with a README.
After that you should see this :

Quick setup — if you've done this kind of thing before

☒ Set up in Desktop
 or
 ☐ HTTPS
 ☐ SSH
 https://github.com/Theemiss/first_lessons.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# first_lessons" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/Theemiss/first_lessons.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/Theemiss/first_lessons.git
git branch -M main
git push -u origin main
```

...or import code from another repository
You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Activate Windows
Go to Settings to activate

2. Getting a Git Repository

After that let link the repository to your local let's head up to your pc and start by creating a directory with the same name "first_lessons" and cd into it .

```
themis@Ahmed-Belhaj:~$ mkdir first_lessons
themis@Ahmed-Belhaj:~$ cd first_lessons/
themis@Ahmed-Belhaj:~/first_lessons$
```

We will try to push a README file

Then let's follow those steps in the github page before ,start with git init which used to initialize a local remote , after that we will use git add to save the changes to local and prepare them for staging using git commit. It's used to move files from the staging area to a commit.

After that The first command git remote add origin

https://github.com/<username>/first_lessons.git

creates a connection between your local repo and the remote repo on Github.

The URL for your remote project should be entirely different from the one above. So to follow along, make sure you are following the steps and working with your own remote repo. You won't usually get a response after executing this command but make sure you have an internet connection.

The second command git branch -M main changes your main branch's name to "main". The default branch might be created as "master", but "main" is the standard name for this repo now. There is usually no response here.

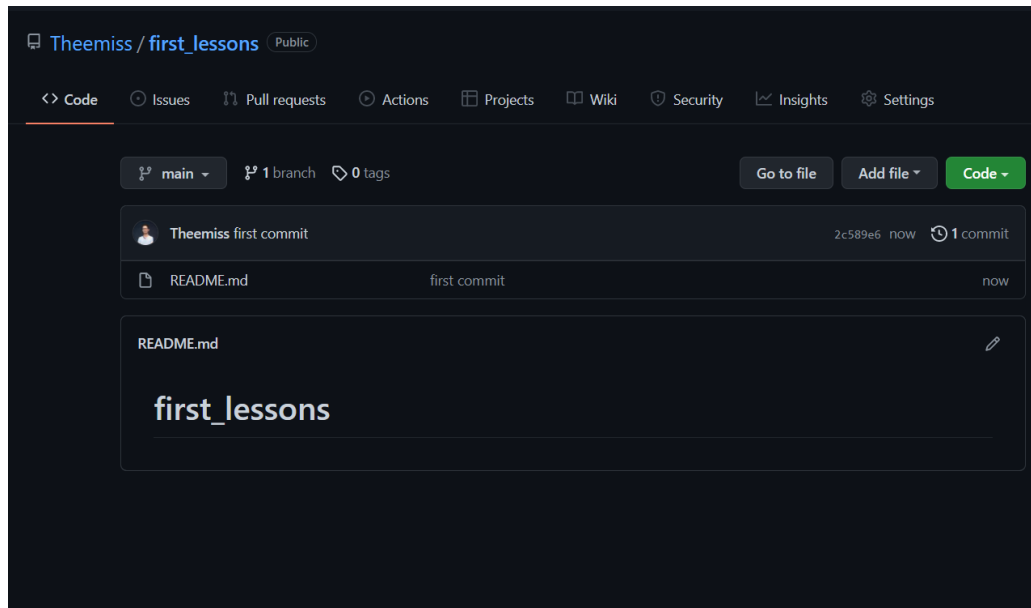
The last command git push -u origin main pushes your repo from your local device to GitHub. You should get a response similar to this:

```

themis@Ahmed-Belhaj:~/first_lessons$ echo "# first_lessons" >>
README.md
mainthemis@Ahmed-Belhaj:~/first_lessons$ git init
Initialized empty Git repository in /home/themis/first_lessons/.git/
themis@Ahmed-Belhaj:~/first_lessons$ git add README.md
themis@Ahmed-Belhaj:~/first_lessons$ git commit -m "first commit"
[master (root-commit) 9884e17] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
themis@Ahmed-Belhaj:~/first_lessons$ git branch -M main
themis@Ahmed-Belhaj:~/first_lessons$ git remote add origin
https://github.com/Theemiss/first_lessons.git
themis@Ahmed-Belhaj:~/first_lessons$ git push -u origin main
Username for 'https://github.com': test@gmail.com
Password for 'https://test@gmail.com@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 338 bytes | 169.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:Theemiss/first_lessons.git
 604e617..e61d8be  main -> main

```

If you check your repository it should look something like this :



Now you successfully initialized a repository and pushed a readme file to the “main” branch.

3. Initializing a repository

The `git init` command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

```
$ git init
```

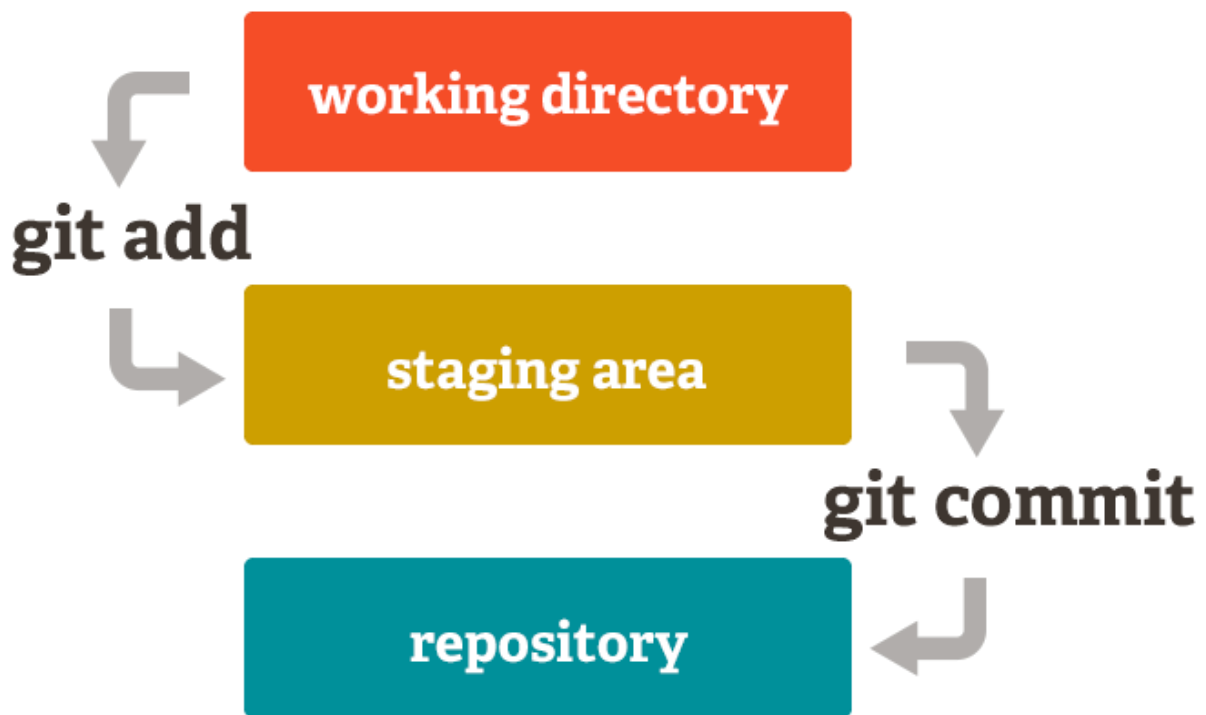
4. Cloning an existing repository

If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is `git clone`. Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run `git clone`. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there).

```
$ git clone <URL>
$ git clone https://github.com/Theemiss/first_lessons.git #using URL
$ git clone git@github.com:Theemiss/first_lessons.git #using SSH
```

5. Remote and recording Changes

There are three core areas to git. These are the Working Tree, the Staging Area (also known as Index), and the Local Repository. When working in a git repository files and modifications will travel from the Working Tree to the Staging Area and finish at the Local Repository. Thus we will talk about these three areas in that order.



- **Working Tree**

The Working Tree is the area where you are currently working. It is where your files live. This area is also known as the “untracked” area of git. Any changes to files will be marked and seen in the Working Tree. Here if you make changes and do not explicitly save them to git, you will lose the changes made to your files. This loss of changes occurs because git is not aware of the files or changes in the Working Tree until you tell it to pay attention to them. If you make changes to files in your working tree git will recognize that they are modified, but until you tell git “Hey pay attention to these files,” it won’t save anything that goes on in them.

How can you see what is in your Working Tree? Run the command **git status**. This command will show you two things: The files in your Working Tree and the files in your Staging Area. It will look something like the image below if you don’t have anything in your Staging Area.

```
$ git status
```

- **Staging Area (git add)**

The Staging Area is when git starts tracking and saving changes that occur in files. These saved changes reflect in the .git directory. That is about it when it comes to the Staging Area. You tell git that I want to track these specific files, then git says okay and moves them from your Working Tree to the Staging Area and says “Cool, I know about this file in its entirety.” However, if you make any more additional changes after adding a file to the Staging Area, git will not know about those specific changes until you tell it to see them. You explicitly have to tell git to notice the edits in your files.

git add is used to record changes from local to the staging area

```
$ git add README.md #adding a single file
$ git add * #adding all files in Current directory
$ git add README.md git/hello.txt # adding multiple files
```

Example:

```
$ git add README.md #adding a single file
$ git add * #adding all files in current directory
$ git add README.md git/hello.txt #adding multiple files
```

Note: Empty directory can't be added

- **The Local Repository**

is everything in your .git directory. Mainly what you will see in your Local Repository are all of your checkpoints or commits. It is the area that saves everything (so don't delete it). That's it. How do you add items from your Staging Area to your Local Repository? The git command **git commit** takes all changes in the Staging Area, wraps them together and puts them in your Local Repository. A commit is simply a checkpoint telling git to track all changes that have occurred up to this point using our last commit as a comparison. After committing, your Staging Area will be empty.

```
$ git commit <flag> "<commit Description>"
```

Example:


```
$ git commit -m "first commit"
$ git commit -a # The -a flag, which stands for all, allows you to
automatically stage all modified files to be committed.
$ git commit --amend -m "added check position to app.py"
```

How can you see what is in your Local Repository? There are a few commands that you can do that show different things.

```
$ git ls-tree --full-tree -r HEAD
```

Example:

```
themis@Ahmed-Belhaj:~/first_lessons$ git ls-tree --full-tree -r HEAD
100644 blob d3be9581c56b208e382f5cd62ac0fddbf1658e1 README.md
100644 blob 748955e7110e7e0c31f8bf712659057dbca1c677 initialize.sh
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 script.py
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 script.sh
```

This command shows all files within your git repo that it's tracking. I don't use this one that much because I rarely need to see every file that is being tracked by git.

git log

I use this command a lot more as it brings up a log of all previous checkpoints in my repository. If I want to see more information about a specific commit, then I run the command **git show** #commit# to see what was changed at that specific checkpoint.

```
themis@Ahmed-Belhaj:~/first_lessons$ git show
commit d729fac0d6da779ef02def5589f7a1b2a87562f4 (HEAD -> main, origin/main,
origin/HEAD)
Author: Theemiss <midinfotn401@gmail.com>
Date: Thu Dec 30 19:12:28 2021 +0100

    update

diff --git a/script.py b/script.py
new file mode 100644
index 0000000..e69de29
diff --git a/script.sh b/script.sh
new file mode 100644
index 0000000..e69de29
```

Learn more about git commit and how to create a commit message here :

- [Git Commit Guide](#)
- [How to Write commit Messages](#)

- **Remote Repository**

The git push command is used to upload local repository content to a remote repository.

```
$ git push <remote> <branch>
```

Example :

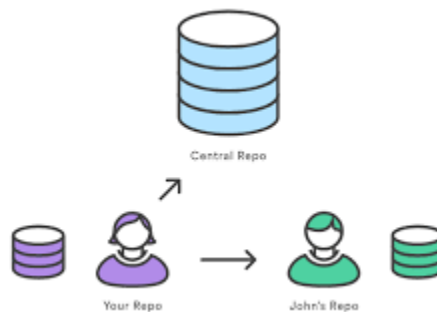
```
$ git push origin main  
$ git push # push to default branch  
$ git push origin DevBranch
```

→ Git remote:

The git remote command lets you create , view and delete connections to other repositories .

Remote connections are more like bookmarks rather than direct link into the repository. Instead of providing a real-time access they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer's repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



Let's check our remote

```
$ git remote #display the remote
Origin
$ git remote -v # same as before but include the url of each connection
origin git@github.com:Theemiss/first_lessons.git (fetch)
origin git@github.com:Theemiss/first_lessons.git (push)
```

You also can change the remote, remove ,add, or rename it
Using the following :

```
$ git remote add <name> <url>
$ git remote rm <name>
$ git remote rename <old-name> <new-name>
```

→ The origin remote:

When you **clone** (Git command line utility which is used to target an existing repository and create a clone) a repository, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most git based projects call their central repository origin.

Basic Usage of git :

```
$ git clone <repo>
$ touch test
$ git add test
$ git commit -m "Initial commit"
$ git push origin main
```

6. Checking the Commit History

After you created a several commit , or if you have cloned a repository with an existing commits history , you'll probably want to check what happened , the most basic command is git log

```

themis@Ahmed-Belhaj:~$ git log
commit f5d8de84650cbc75484f543716c40d055b7914ab (HEAD -> main,
origin/main)
Author: Theemiss <test@gmail.com>
Date: Thu Dec 30 13:48:19 2021 +0100

    Added new endpoint to user.py
: ...
:

```

One of the more helpful options is -p or --patch, which shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

```

themis@Ahmed-Belhaj:~$ git log -p -2
commit f5d8de84650cbc75484f543716c40d055b7914ab (HEAD -> main,
origin/main)
Author: Theemiss <test@gmail.com>
Date: Thu Dec 30 13:48:19 2021 +0100

    update

diff --git
a/0x0D-csharp-text_based_interface/Tests/InventoryManagement.csproj
b/0x0D-csharp-text_based_interface/InventoryManagement.Tests/Inventory
Management.csproj
j
similarity index 100%
rename from
0x0D-csharp-text_based_interface/Tests/InventoryManagement.csproj
rename to
0x0D-csharp-text_based_interface/InventoryManagement.Tests/InventoryM
anagement.csproj
diff --git a/0x0D-csharp-text_based_interface/Tests/UnitTest1.cs
b/0x0D-csharp-text_based_interface/InventoryManagement.Tests/UnitTest
1.cs
similarity index 100%

```

Until this point we already come across a lot of tricks and how to use basic git , lets dive more into the changing in repository and tracking changes and more the collab theme .

Let's get back to the repo we created at the start of this course

```
themis@Ahmed-Belhaj:$ git clone
https://github.com/<username>/first_lessons.git
themis@Ahmed-Belhaj:$ cd first_lessons/
themis@Ahmed-Belhaj:/first_lessons$ ls -a
.  ..  .git  README.md
themis@Ahmed-Belhaj:/first_lessons$
```

Whenever you clone or init a repo in the local there will be file you cloned and also the git config inside a hidden folder called `.git` learn more about it here [StackOverflow](#)

Let's start by creating a new file called `initialize.sh` which contain how to initialize a repository from github

```
themis@Ahmed-Belhaj:/first_lessons$ touch initialize.sh
themis@Ahmed-Belhaj:/first_lessons$ emacs initialize.sh
themis@Ahmed-Belhaj:/first_lessons$ cat initialize.sh
git init
echo "#hello" >> README.md
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/Theemiss/first_lessons.git
git push -u origin main
themis@Ahmed-Belhaj:/first_lessons$ git status #used to display the
state of the repository and staging
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
initialize.sh
```

```
nothing added to commit but untracked files present (use "git add" to track)
themis@Ahmed-Belhaj:/first_lessons$
```

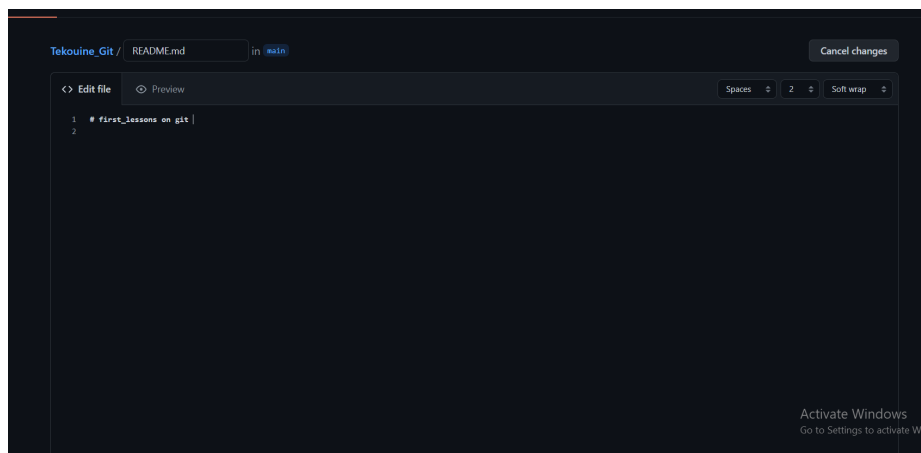
We used git status to check the untracked file `intialize.sh`, let's add

```
themis@Ahmed-Belhaj:/first_lessons$ git add initialize.sh
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   initialize.sh
themis@Ahmed-Belhaj:/first_lessons$ git commit -m "update"
[main 0a8e1a3] update
1 file changed, 7 insertions(+)
create mode 100644 initialize.sh
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
themis@Ahmed-Belhaj:/first_lessons$ git push
...
```

Let's head up to your remote repo on github and make some changes there



We will try change README.md from github

Next up to get the updated we made on github platform we need to pull the changes to our local ,

```
themis@Ahmed-Belhaj:/first_lessons$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 689 bytes | 29.00 KiB/s, done.
From github.com:Theemiss/first_lessons.git
   0a8e1a3..9d969d7  main      -> origin/main
Updating 0a8e1a3..9d969d7
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
themis@Ahmed-Belhaj:/first_lessons$
```

➤ Git pull

Git pull is used to to get the changes made on the remote to you local , let's take another practical example let's say there two Contributor working on this repo or a team , if one of them made changes and pushed it to repo , you need to get those changes and work on them or add some other code to it . and pulling before pushing is good practice cause at some point you came across conflict or you both did the same thing or worked on the same code . By default git pull do two things Updates the current local working branch (currently checked out branch) Updates the remote tracking branches for all other branches

```
$ git pull [<options>] [<repository> [<refspec>...]]
```

➤ Git fetch

We've seen git pull and now we will see git fetch , there are a lot of people who don't know the difference between git pull and git fetch .

That said, to keep your clone updated with whatever changes may have been applied to the original, you'll need to bring those to your clone.

That's where fetch and pull come in.

git fetch is the command that tells your local git to retrieve the latest meta-data info from the original (yet doesn't do any file transferring. It's more like just checking to see if there are any changes available).

git pull on the other hand does that AND brings (copy) those changes from the remote repository.

You can use git fetch to know the changes done in the remote repo/branch since your last pull. This is useful to allow for checking before doing an actual pull, which could change files in your current branch and working copy (and potentially lose your changes, etc).

```
$ git fetch [<options>] [<repository> [<refspec>...]]
$ git fetch [<options>] <group>
$ git fetch --multiple [<options>] [(<repository> | <group>)...]
$ git fetch --all [<options>]
```

More into checking changes see

- [git diff](#)

7. Undoing Things

Let's say you were working on a project and you made a lot of changes and they are not working and you should go back to the previous checkpoint .

At every stage you may want to undo things here is some commands which will help you

One of the most common undo happen when you commit to early and possibly forget something , or you mess up the commit messages you want redo that commit , then make the changes stage them using git add and commit using **-amend** option

```
$ git commit -amend
```

Example

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```


- **Unstaging unstaged files**

The next phase will show us how to work with your staging area and working directory changes . for example let's say you've changed two files

And want to commit them as two separate changes , but you accidentally you typed `git *` and staged them both

Use this Two files `script.py` , `script.sh`

```
themis@Ahmed-Belhaj:/first_lessons$ ls
README.md  initialize.sh  script.py  script.sh
themis@Ahmed-Belhaj:/first_lessons$ git add *
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   script.py
        modified:   script.sh
themis@Ahmed-Belhaj:/first_lessons$
```

Right bellow you can see changes to be committed , and it's says `git restore --staged <file>` to unstage , let's follow him

```
themis@Ahmed-Belhaj:/first_lessons$ git restore --staged script.py
script.sh
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   script.py
```

```
modified:  script.sh

no changes added to commit (use "git add" and/or "git commit -a")

themis@Ahmed-Belhaj:/first_lessons$
```

```
$ git restore <filename>
```

The "restore" command helps to unstage or even discard uncommitted local changes. On the other hand, the restore command can also be used to discard local changes in a file, thereby restoring its last committed state.

You can do the same thing with git reset as follow

```
themis@Ahmed-Belhaj:/first_lessons$ git reset HEAD script.py
script.sh
Unstaged changes after reset:
    script.py
    script.sh
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
    modified:   script.py
    modified:   script.sh

no changes added to commit (use "git add" and/or "git commit -a")
```

- **Unmodifying a modified file :**

What if you realize that you don't want to keep your changes to the script.py file? How can you easily unmodify it — revert it back to what it looked like when you last committed? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   script.py
        modified:   script.sh
```

It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

```
themis@Ahmed-Belhaj:/first_lessons$ git restore script.py
themis@Ahmed-Belhaj:/first_lessons$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   script.sh

no changes added to commit (use "git add" and/or "git commit -a")
```

You can see that the changes have been reverted.

You can do the same git checkout

```
$ git checkout -- script.py
$ git status
On branch master
Changes to be committed:
  (use "git restore <file>..." to discard changes in working
director)
        modified:   script.sh

no changes added to commit (use "git add" and/or "git commit -a")
```

Look more into [git reset here](#)

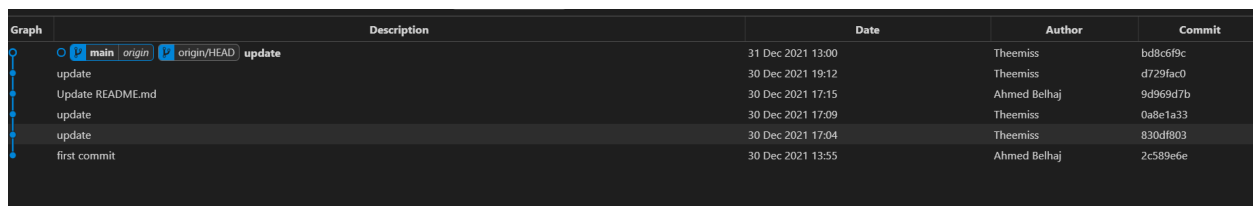
Branching

1. Branches in Nutshell

Nearly every version control system (VCS) has some sort of branching support and git also use it, branching means you diverge from the main line of the development and continue working without messing the main line .In many **VCS** tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

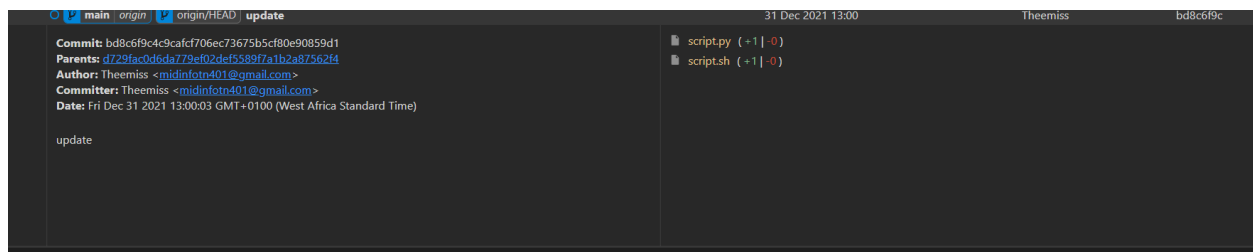
Some people refer to Git's branching model as its “killer feature,” and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

To really understand how git branching work you need to understand how git store data
You can check [What is Git?](#), Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots.



Graph	Description	Date	Author	Commit
main origin	update	31 Dec 2021 13:00	Theemiss	bd8c6f9c
origin/HEAD update	update	30 Dec 2021 19:12	Theemiss	d729fac0
	Update README.md	30 Dec 2021 17:15	Ahmed Belhaj	9d969d7b
	update	30 Dec 2021 17:09	Theemiss	0a8e1a33
	update	30 Dec 2021 17:04	Theemiss	830df803
	first commit	30 Dec 2021 13:55	Ahmed Belhaj	2c589e6e

Example of Simple Graph



	31 Dec 2021 13:00	Theemiss	bd8c6f9c
Commit: bd8c6f9c4c9cafc7706ec73675b5cf80e90859d1	script.py (+1 -0)		
Parents: d729fac0d6da779ef02def5589f7a1b2a875624	script.sh (+1 -0)		
Author: Theemiss <midinfotn401@gmail.com>			
Committer: Theemiss <midinfotn401@gmail.com>			
Date: Fri Dec 31 2021 13:00:03 GMT+0100 (West Africa Standard Time)			
update			

Example of Snapshot

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches. To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one, stores that version of the file in the Git repository (Git refers to them as **blobs**), and adds that checksum to the staging area

```
themis@Ahmed-Belhaj:/first_lessons$ touch .gitignore LICENSE test.sh
themis@Ahmed-Belhaj:/first_lessons$ cat .gitignore
~
.vs
test.sh
themis@Ahmed-Belhaj:/first_lessons$ git add .gitignore LICENSE
themis@Ahmed-Belhaj:/first_lessons$ git commit -m "added .gitignore and
LICENSE"
g .gitignore and LICENSE'
[main 29fcce1] adding .gitignore and LICENSE
2 files changed, 3 insertions(+)
create mode 100644 .gitignore
create mode 100644 LICENSE
```

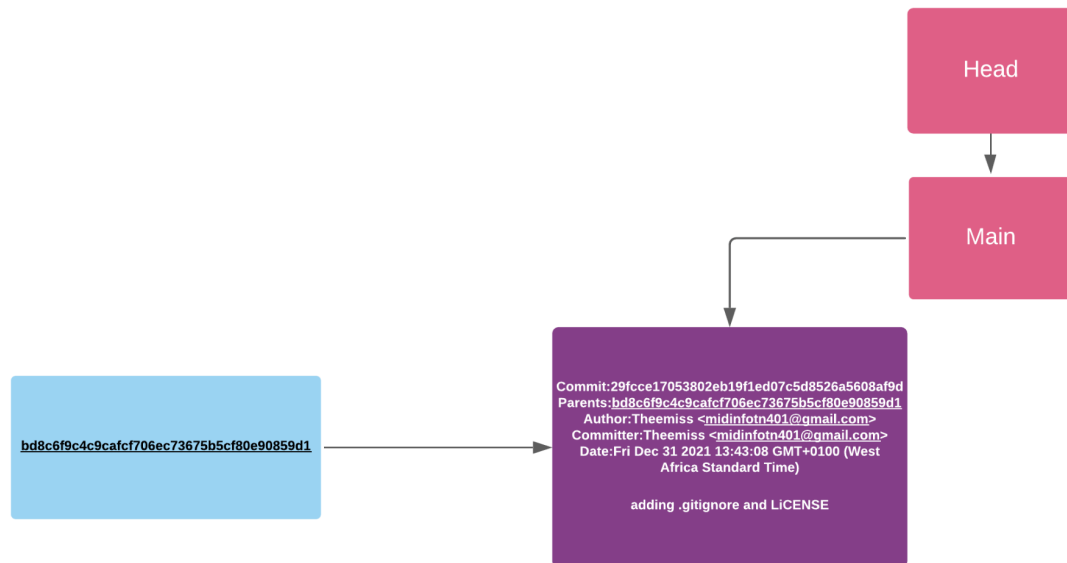
When you create the commit by running `git commit`, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can recreate that snapshot when needed.

Your Git repository now contains four objects: two blobs (each representing the contents of one of the two files), one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.



If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is main. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the main branch pointer moves forward automatically.



2. Basic Branching and Merging

- *Creating a new branch*

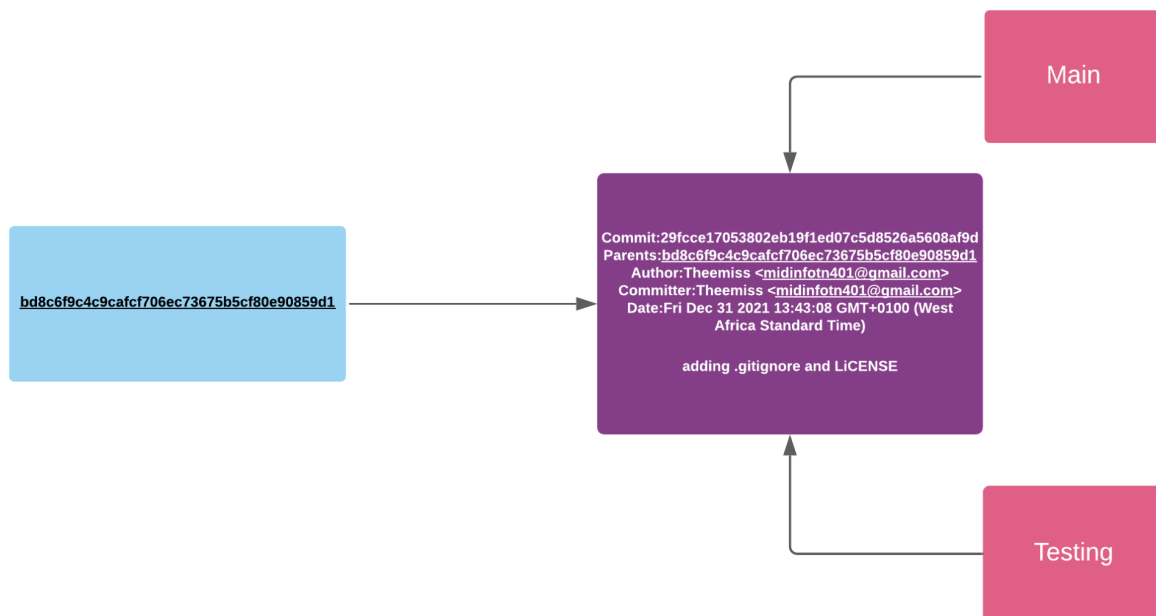
What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you want to create a new branch called **testing**. You do this with the git branch command

```
$ git branch <branchname>
```

Let's start by creating our new branch

```
themis@Ahmed-Belhaj:/first_lessons$ git branch testing  
  
$themis@Ahmed-Belhaj:/first_lessons$
```

This creates a new pointer to the same commit you're currently on.



How does Git know what branch you're currently on? It keeps a special pointer called HEAD. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on main. The git branch command only created a new branch — it didn't switch to that branch. You can use git branch only to check the current working branch or git log with --decorate option

```
themis@Ahmed-Belhaj:/first_lessons$ git branch  
* main  
  testing  
themis@Ahmed-Belhaj:/first_lessons$ git log --oneline --decorate  
29fcae1 (HEAD -> main, origin/main, origin/HEAD, testing) adding .gitignore  
and LICENSE  
bd8c6f9 update  
d729fac update
```

- **Switching branches**

To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

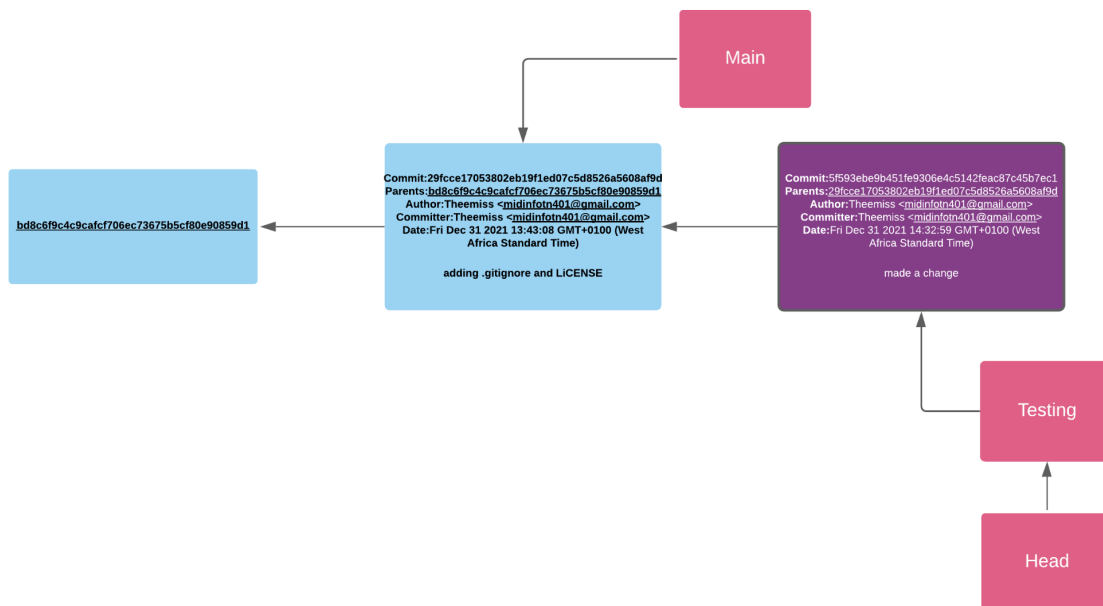
```
themis@Ahmed-Belhaj:/first_lessons$ git checkout testing

Switched to branch 'testing'
themis@Ahmed-Belhaj:/first_lessons$ git branch
  main
* testing
```

This moves **HEAD** to point to the **testing** branch.

Now lets see what happen when we work on another branch lets alter README.md file

```
themis@Ahmed-Belhaj:/first_lessons$ nano README.md
themis@Ahmed-Belhaj:/first_lessons$ git commit -a -m 'made a change'
[testing 5f593eb] made a change
 1 file changed, 1 insertion(+)
themis@Ahmed-Belhaj:/first_lessons$ git push
```



This is interesting, because now your testing branch has moved forward, but your main branch still points to the commit you were on when you ran git checkout to switch branches.

HEAD moves when you checkout

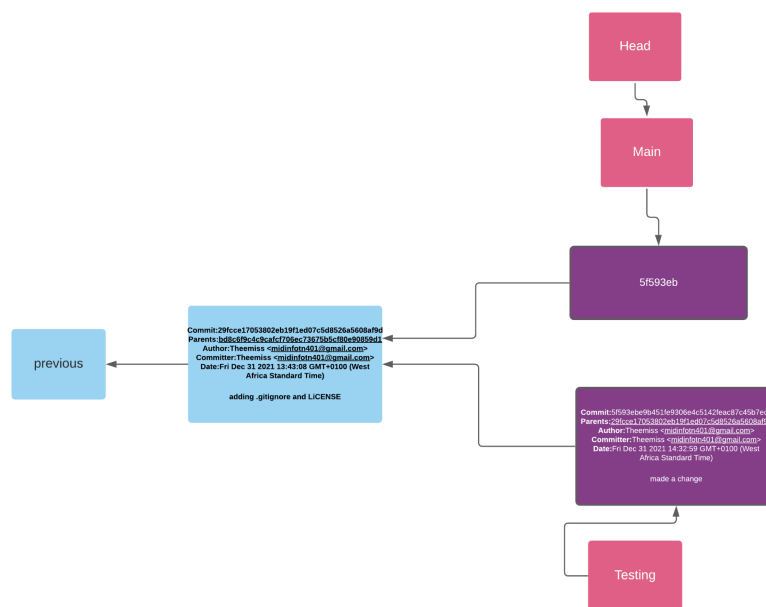
That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.

This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your testing branch so you can go in a different direction.

Lets make some changes again and alter another file

```
themis@Ahmed-Belhaj:/first_lessons$ git checkout main
...
themis@Ahmed-Belhaj:/first_lessons$ nano script.py
themis@Ahmed-Belhaj:/first_lessons$ git add script.py
themis@Ahmed-Belhaj:/first_lessons$ git commit -m "modefied script.py"
themis@Ahmed-Belhaj:/first_lessons$ git push
```

Now your project history has diverged. You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple branch, checkout, and commit ,add commands.



You can check using `git log` command for the changes

```
themis@Ahmed-Belhaj:/first_lessons$ git log --oneline --decorate --graph --all
* 877773b (HEAD -> main) modefied script.py
| * 5f593eb (testing) made a change
|/
* 29fcce1 (origin/main, origin/HEAD) adding .gitignore and LiCENSE
* bd8c6f9 update
* d729fac update
* 9d969d7 Update README.md
* 0a8e1a3 update
* 830df80 update
* 2c589e6 first commit
themis@Ahmed-Belhaj:/first_lessons$
```

More into Creating and switching

- Switch to an existing branch: `git switch testing-branch`.
- Create a new branch and switch to it: `git switch -c new-branch`. The `-c` flag stands for create, you can also use the full flag: `--create`.
- Return to your previously checked out branch: `git switch -`.

Creating a new branch and switching to it at the same time

It's typical to create a new branch and want to switch to that new branch at the same time — this can be done in one operation with `git checkout -b <newbranchname>`.

- ***Basic Branching and Merging***

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do some work on a project.
2. Create a branch for a Feature you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

3. Switch to your production branch.
4. Create a branch to add the hotfix.
5. After it's tested, merge the hotfix branch, and push to production.
6. Switch back to your original Feature branch

We will use this repo from here you are going to fork it from here [Branching Repo](#)

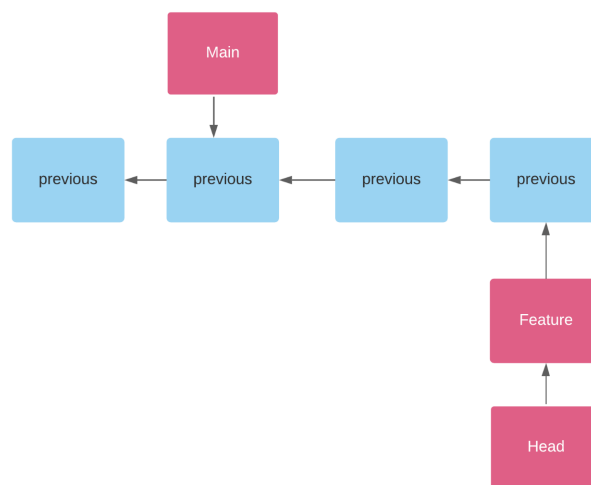
```
themis@Ahmed-Belhaj:$ git clone https://github.com/<username>/Branching.git
themis@Ahmed-Belhaj:$ cd Branching
themis@Ahmed-Belhaj:/Branching$
```

Use the fork button on the top right corner.in this project your feature branch is ahead of your main branch

You decided to work on some Feature

To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

```
themis@Ahmed-Belhaj:/Branching$ git checkout -b Feature
Switched to a branch 'Feature'
themis@Ahmed-Belhaj:/Branching$ nano Feature.py
themis@Ahmed-Belhaj:/Branching$ nano app.py
themis@Ahmed-Belhaj:/Branching$ git commit -m 'added feature[Feature]'
```



Now you get the call that there is an issue with the app and you need to fix it immediately. With Git, you don't have to deploy your fix along with the Feature changes you've made, and you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your main branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) **Stashing and Cleaning**. For now, let's assume you've committed all your changes, so you can switch back to your main branch

```
$ git checkout main
Switched to branch 'master'
```

At this point, your project working directory is exactly the way it was before you started working on Feature, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you

committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a hotfix branch on which to work until it's completed

```
themis@Ahmed-Belhaj:/Branching$ git checkout -b hotfix
Switched to a new branch 'hotfix'
themis@Ahmed-Belhaj:/Branching$ nano app.py
$ git commit -a -m 'Fix broken calculator'
[hotfix 1fb7853]Fix broken calculator
1 file changed, 2 insertions(+)
```

You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your main branch to deploy to production. You do this with the git merge command

```
themis@Ahmed-Belhaj:/Branching$ git checkout main
themis@Ahmed-Belhaj:/Branching$ git merge hotfix
Updating 0c39a79..4986937
Fast-forward
 app.py | 5 +++++
1 file changed, 5 insertions(+)
```

you'll notice the phrase "fast-forward" in that merge. Because the commit 0c39a79 pointed to by the branch hotfix you merged in was directly ahead of commit the main branch you're on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast-forward."

To delete the hotfix branch after merging you can use

```
themis@Ahmed-Belhaj:/Branching$ git branch -d hotfix
Deleted branch Hotfix (was 4986937).
```

Now you can switch back to your feature branch and continue working on

It's worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch. If you need to pull it in, you can merge your main branch into your Feature branch by running `git merge main`, or you can wait to integrate those changes until you decide to pull the Feature branch back into main later.

- **Basic Merging**

Suppose you've decided that your Feature work is complete and ready to be merged into your master branch. In order to do that, you'll merge your Feature branch into master, much like you

merged your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command

```
themis@Ahmed-Belhaj:/Branching$ git checkout main
Switched to branch 'main'
themis@Ahmed-Belhaj:/Branching$ git merge Feature
Merge made by the 'recursive' strategy.
app.py | 1 +
1 file changed, 1 insertion(+)
```

Learn more about 'recursive' strategy [here](#)

- **Merging Conflicts**

Occasionally, this process doesn't go smoothly. If you change the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly. If your fix for Feature modified the same part of a file as the hotfix branch, you'll get a merge conflict that looks something like this

```
themis@Ahmed-Belhaj:/Branching$ git merge Feature
Auto-merging app.py
CONFLICT (content): Merge conflict in app.py
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status

```
themis@Ahmed-Belhaj:/Branching$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

   both modified:   app.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this

```
<<<<<< HEAD:app.py
Res = a +b
=====
Res = calculator(a,b)
>>>>>>Feature:app.py
```

This means the version in HEAD (your main branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =====), while the version in your Feature branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this

```
Res = calculator(a,b)
```

This resolution has a little of each section, and the <<<<<<, =====, and >>>>>> lines have been completely removed. After you've resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

Now you've seen how to work with branches and simple conflicts lets look more into some branching workflow

7. Branching Workflow

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate them into your own development cycle

You can look into those

- Long-Running Branches
- progressive-stability branching
- Topic Branches
- Multiple topic branches

Resources:

- [Git Branching - Branch Management](#)
- [\(deleting , rename ..\)](#)
- [Using branches](#)
- [git merge](#)
- [Rebasing](#)

- Remote branching

Summary

You should have a basic understanding of what Git is and how it's different from any centralized version control systems you may have been using previously. You should also now have a working version of Git on your system that's set up with your personal identity. It's now time to learn some Git basics.

At this point, you can do all the basic local Git operations — creating or cloning a repository, making changes, staging and committing those changes, and viewing the history of all the changes the repository has been through. Next, we'll cover Git's killer feature: its branching model.

We've covered basic branching and merging in Git. You should feel comfortable creating and switching to new branches, switching between branches and merging local branches together. You should also be able to share your branches by pushing them to a shared server, working with others on shared branches and rebasing your branches before they are shared. Next, we'll cover what you'll need to run your own Git repository-hosting server.

sdcfv)mm