WEEK 2 - PART 2

CODER'S TOOLKIT-

TERMINAL, FILE SYSTEMS & VERSION CONTROL

LINNAEUS
UNIVERSITY

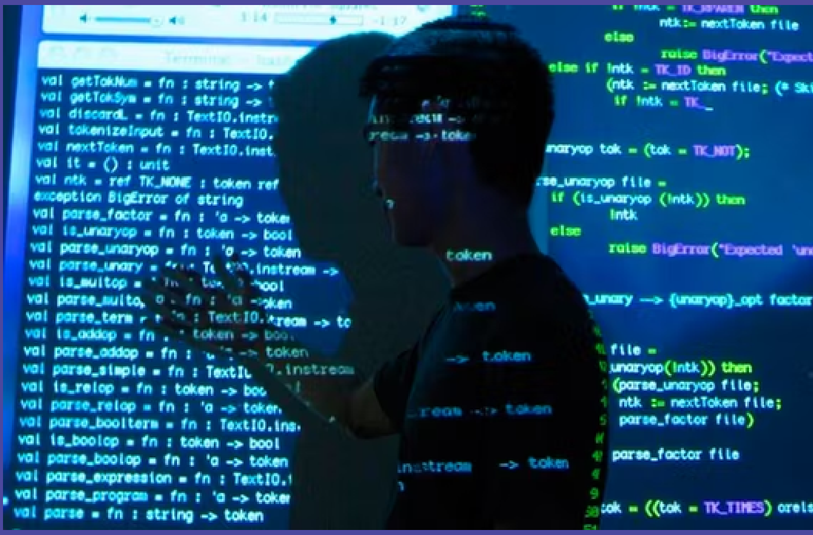
SUMMER 2024

TABLE OF CONTENTS

WHAT ARE
BASICS FOR CODERS

In the world of software development, proficiency with the terminal, a clear understanding of file systems, and mastery of version control systems are foundational skills.

The terminal provides a powerful interface for executing commands, automating tasks, and managing systems efficiently.



A solid grasp of file systems and directory structures is essential for programmers because it directly influences how they interact with files and resources within their projects.

Understanding file paths and how to navigate folders using the terminal allows developers to efficiently manage and access code, configuration files, and other assets.

This knowledge is crucial when linking file paths in code, setting up environment configurations, and deploying applications.
For instance, being able to reference files correctly in scripts and configuration files ensures that your code functions as intended, whether it's locating a library, loading data, or executing a build process.

Meanwhile, version control systems like Git are indispensable for tracking code changes, facilitating collaboration, and maintaining a historical record of modifications, ensuring that developers can manage and restore their work with ease.

These are fundamental skills for any programmer, as they provide the essential foundation for managing code, navigating project structures, and ensuring efficient development workflows.

ESSENTIALS - THE TERMINAL & SHELL

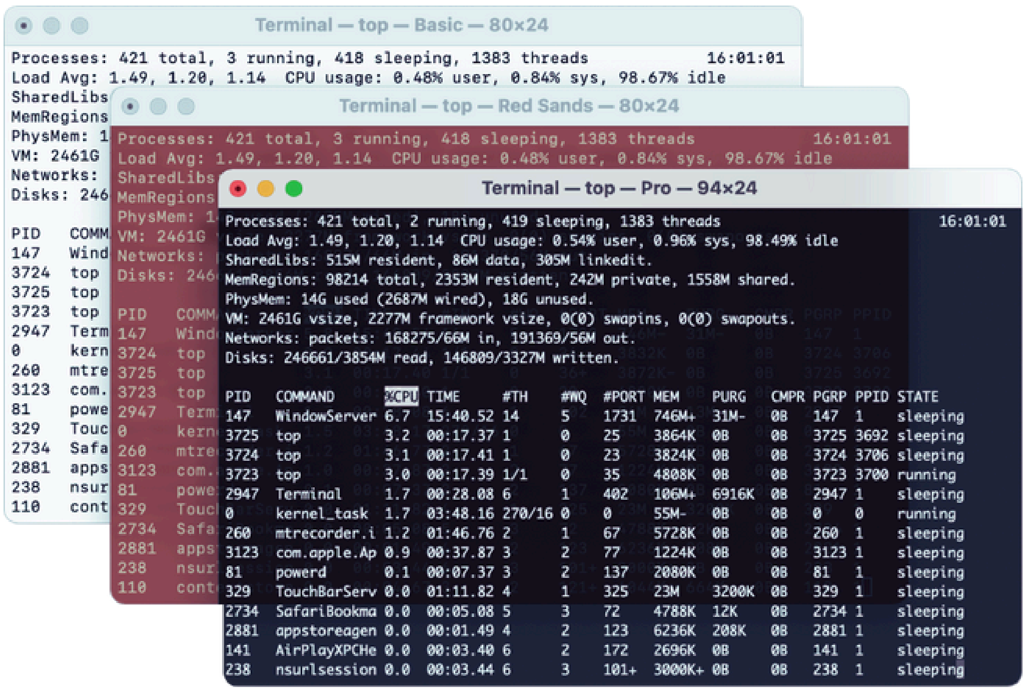
In the early days of computing, users interacted with their machines through plain, black screens, inputting commands in a text-based format.

The terminal is the graphical interface and environment where you input commands, while the shell is the underlying interpreter that processes these commands.

The terminal provides the UI - a window where users type commands, allowing users to access the shell.

A shell is a command interpreter that processes the commands typed into the terminal and communicates with the operating system to execute them. It manages the interpretation and execution of commands, handling tasks such as file manipulation, program execution, and system monitoring. The shell interacts directly with the operating system's kernel to perform these actions.

MacOS The Terminal


Terminal — top — Basic — 80×24
Terminal — top — Red Sands — 80×24
Terminal — top — Pro — 94×24

```
Processes: 421 total, 3 running, 418 sleeping, 1383 threads            16:01:01
Load Avg: 1.49, 1.20, 1.14  CPU usage: 0.48% user, 0.84% sys, 98.67% idle
SharedLibs
MemRegions
PhysMem: 1
VM: 2461G
Networks:
Disks: 246

PID   COMM
147   Wind
3724  top
3725  top
3723  top
2947  Term
0     kern
260   mtre
3123  com.
81    powe
329   Touc
2734  Safa
2881  apps
238   nsur
110   cont
```

```
Processes: 421 total, 3 running, 418 sleeping, 1383 threads            16:01:01
Load Avg: 1.49, 1.20, 1.14  CPU usage: 0.48% user, 0.84% sys, 98.67% idle
SharedLibs
MemRegions
PhysMem: 1
VM: 2461G
Networks:
Disks: 246

PID   COMM
147   Wind
3724  top
3725  top
3723  top
2947  Term
0     kern
260   mtre
3123  com.
81    powe
329   Touc
2734  Safa
2881  apps
238   nsur
110   cont
```

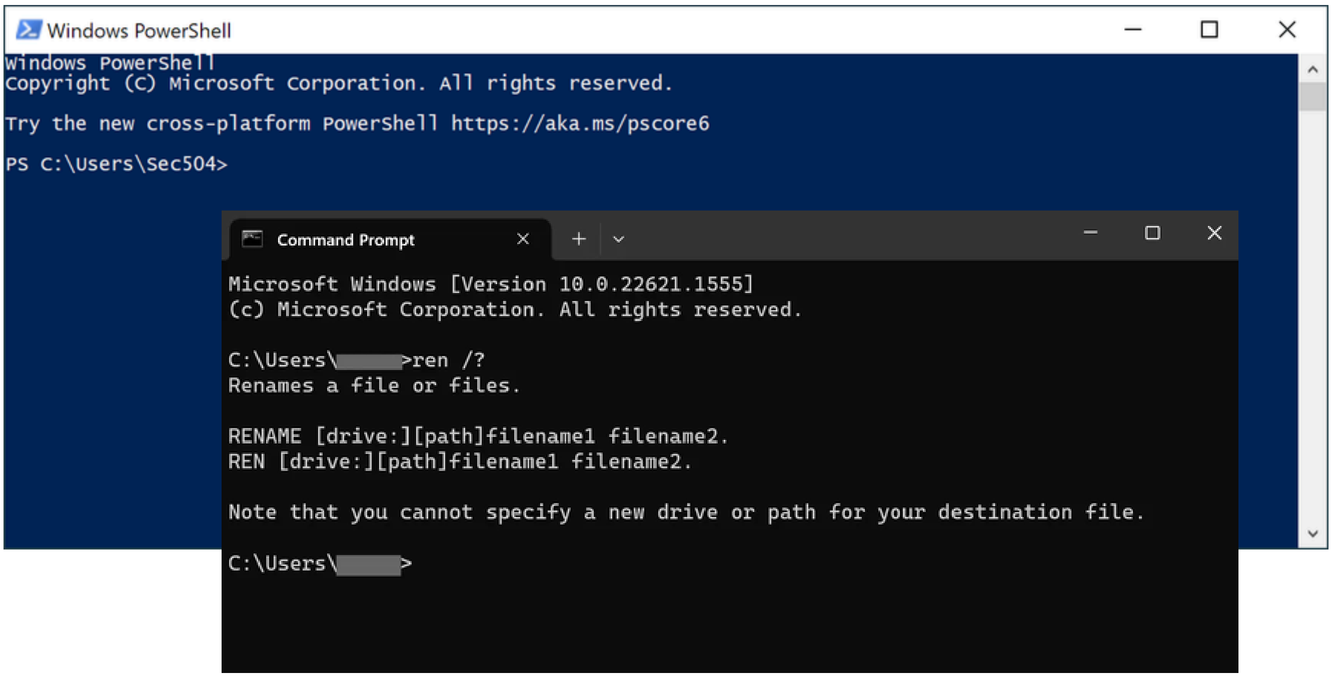
```
Processes: 421 total, 2 running, 419 sleeping, 1383 threads                          16:01:01
Load Avg: 1.49, 1.20, 1.14  CPU usage: 0.54% user, 0.96% sys, 98.49% idle
SharedLibs: 515M resident, 86M data, 305M linkedit.
MemRegions: 98214 total, 2353M resident, 242M private, 1558M shared.
PhysMem: 14G used (2687M wired), 18G unused.
VM: 2461G vsize, 2277M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 168275/66M in, 191369/56M out.
Disks: 246661/3854M read, 146809/3327M written.

PID   COMMAND      %CPU TIME     #TH    #WQ  #PORT MEM    PURG   CMPR PGRP PPID STATE
147   WindowServer 6.7  15:40.52 14     5    1731  746M+  31M-   0B   147  1    sleeping
3725  top          3.2  00:17.37 1      0    25    3864K  0B     0B   3725 3692 sleeping
3724  top          3.1  00:17.41 1      0    23    3824K  0B     0B   3724 3706 sleeping
3723  top          3.0  00:17.39 1/1    0    35    4808K  0B     0B   3723 3700 running
2947  Terminal     1.7  00:28.08 6      1    402   106M+  6916K  0B   2947 1    sleeping
0     kernel_task  1.7  03:48.16 270/16 0    0     55M-   0B     0B   0    0    running
260   mtrecorder.i 1.2  01:46.76 2      1    67    5728K  0B     0B   260  1    sleeping
3123  com.apple.Ap 0.9  00:37.87 3      2    77    1224K  0B     0B   3123 1    sleeping
81    powerd       0.1  00:07.37 3      2    137   2080K  0B     0B   81   1    sleeping
329   TouchBarServ 0.0  01:11.82 4      1    325   23M    3200K  0B   329  1    sleeping
2734  SafariBookma 0.0  00:05.08 5      3    72    4788K  12K    0B   2734 1    sleeping
2881  appstoreagen 0.0  00:01.49 4      2    123   6236K  208K   0B   2881 1    sleeping
141   AirPlayXPCHe 0.0  00:03.40 6      2    172   2696K  0B     0B   141  1    sleeping
238   nsurlsession 0.0  00:03.44 6      3    101+  3000K+ 0B     0B   238  1    sleeping
```

Essentially, the Terminal acts as a bridge between the user and the shell. Also known as a terminal emulator or console emulator, the terminal is a software program that provides a graphical interface for accessing a computer's command-line interface (CLI).

When you launch a terminal application like macOS's Terminal or Windows Command Prompt, you're opening a terminal emulator, which presents a text-based UI in a window, where you can enter your commands.
The shell interprets the text input you provide in the terminal, executes the corresponding instructions, and returns the output, which is then displayed in the terminal window. By entering commands into the terminal, you can access and modify files, install software, adjust system settings, and gather information about various system components.

It's important to remember that you can have multiple terminals open simultaneously, each working on different tasks. This flexibility allows you to manage various terminal sessions simultaneously, each tailored to different functions or tasks.

PowerShell & Command Prompt for Windows



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Sec504>
```

```
Microsoft Windows [Version 10.0.22621.1555]
(c) Microsoft Corporation. All rights reserved.

C:\Users\[redacted]>ren /?
Renames a file or files.

RENAME [drive:][path]filename1 filename2.
REN [drive:][path]filename1 filename2.

Note that you cannot specify a new drive or path for your destination file.

C:\Users\[redacted]>
```

TERMINAL AND SHELL APPLICATIONS

The terms "shell" and "terminal" are often used interchangeably, which can lead to confusion. While they are closely related, they refer to different aspects of command-line interfaces.

There are various terminals and shells, each offering different features and capabilities, which can further complicate their interchangeable use. Some applications combine both functionalities, such as terminal emulators with built-in shells, allowing users to seamlessly execute commands and interact with the operating system.

macOS's Terminal and PowerShell are often referred to as both terminal and shell because they provide the interface (Terminal) and house the command interpreter (Shell), serving distinct but complementary roles in user-system interaction.

macOS's Terminal serves as a windowed environment for command-line interactions on macOS. It can use either Bash or Zsh as its default shell, allowing users to run commands and scripts within macOS.

Windows' Command Prompt also provides a window for command-line interactions, but it uses the Command Prompt shell. This setup allows for the execution of commands and scripts specific to Windows systems.

PowerShell, on the other hand, functions as both a terminal and a shell on Windows. It integrates a powerful scripting environment with its own scripting language, offering advanced scripting and command execution capabilities.

COMPARING TERMINALS

Although Windows, macOS and Linux use different sets of commands and syntax, many core concepts remain consistent.

Command Prompt and PowerShell use different commands and syntax compared to macOS for performing similar tasks. For example, in Windows Command Prompt, you use `dir` to list folder contents and `copy` to duplicate files, while macOS uses `ls` for listing folder contents and `cp` for copying files. These variations highlight the need to understand the specific command-line conventions of each operating system to perform file management effectively.

macOS Terminal and Windows Command Prompt both provide a window for entering and executing commands tailored to their respective operating systems. macOS Terminal supports more advanced Unix-based commands and scripting through Bash, whereas Windows Command Prompt offers more basic functionality.

PowerShell provides a similar windowed interface for entering commands but extends beyond the capabilities of both macOS Terminal and Windows Command Prompt. It features advanced scripting capabilities, enabling more complex automation and scripting tasks in a powerful environment for Windows.

BASH

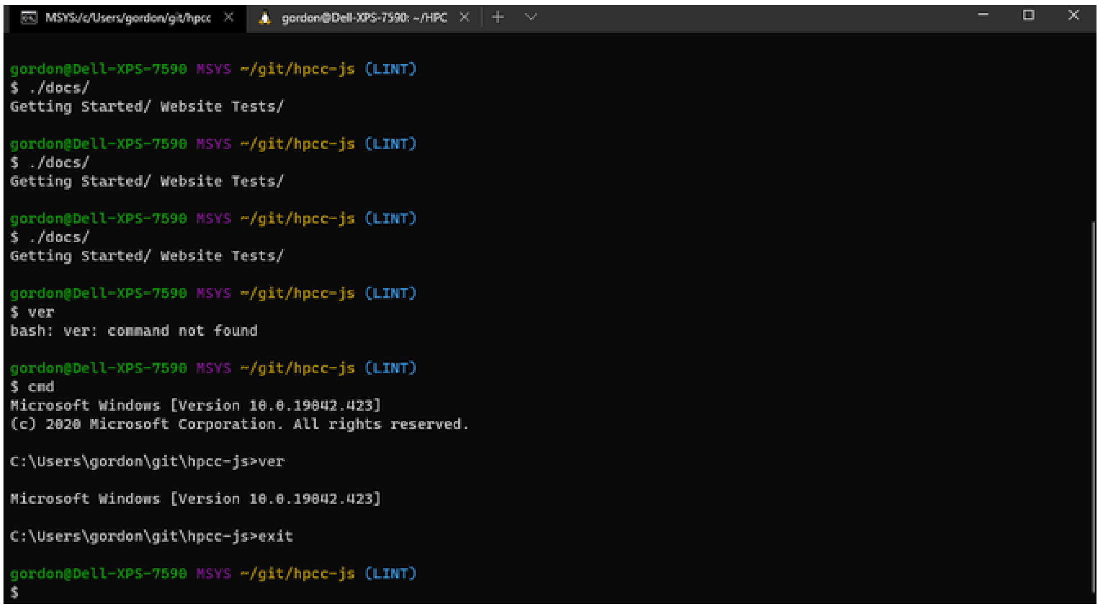
Bash is one of the most widely used shells and serves as the default shell on many **Unix-based operating systems**, including Linux and older versions of macOS. Known for its power and extensive features, Bash offers a broad range of capabilities. It includes various utilities for managing files and processes, as well as tools for automating tasks through scripts.

GIT BASH

**Git Bash is a terminal that mimics the command-line interface found on Unix-based systems, such as Linux and macOS.
Git Bash bridges the gap between macOS and Windows users, allowing developers regardless of their underlying hardware or operating system to work with the same commands and features.**

Git bash does this by emulating the Unix command-line environment, behaviors, and syntax on Windows, helping to provide a consistent terminal experience and streamlining work across different operating systems.

It also incorporates Git, a version control system.
Git Bash unifies the development experience and facilitates working with Git in a terminal environment that behaves similarly across all underlying system software.



```
gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$ ./docs/
Getting Started/ Website Tests/

gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$ ./docs/
Getting Started/ Website Tests/

gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$ ./docs/
Getting Started/ Website Tests/

gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$ ver
bash: ver: command not found

gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$ cmd
Microsoft Windows [Version 10.0.19042.423]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\gordon\git\hpcc-js>ver

Microsoft Windows [Version 10.0.19042.423]

C:\Users\gordon\git\hpcc-js>exit

gordon@Dell-XPS-7590 MSYS ~/git/hpcc-js (LINT)
$
```

FILES & FILE SYSTEMS A GUIDE

WHAT IS A FILE?

A file is a collection of data or information stored on a storage device, and it can **represent various types of content**, such as documents, images, videos, programs, or system configuration files. Each file is identified by a name and includes attributes such as its size, permissions (who can access or modify it), and creation or modification dates.

Files also have a file type, which refers to the specific format or category of data contained within the file, determining how the file is used and which programs can open or process it. The file type is often indicated by a file extension, a suffix at the end of the file name preceded by a dot (e.g., .txt, .jpg).

WHAT IS A DIRECTORY?

Directories, also known as folders, are used to organize and group related files. They provide a hierarchical structure for storing files and other directories, forming a tree-like arrangement that begins with a root directory (the top-most directory in a file system hierarchy) and branches out into subdirectories (subfolders).

Directories can contain a mix of files and subdirectories, facilitating efficient organization and easy navigation of stored data.

FILE SYSTEMS- ORGANIZING & STORING DATA

A file system is a structure used by an operating system to organize, store, and retrieve files on a storage device such as a hard drive or solid-state drive. It defines how files are named, arranged into directories (also known as folders), and stored on the storage media.

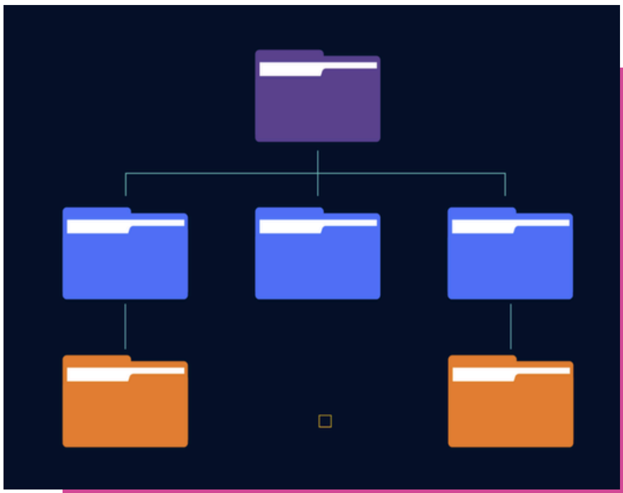
Directories are organized hierarchically, meaning they can contain other directories within them**, creating a "directory tree"** with a **nested structure** of folders and subfolders.

For example, within a top-level "Documents" folder, you might have created subfolders such as "Work," "Personal," and "Projects." Each of these subfolders can contain files related to their specific categories.
This hierarchical organization simplifies file management by allowing you to navigate through the levels of directories to locate specific files.

In addition to organizing files, **file systems manage file permissions**, for security reasons and to aid multi-user environments.

Different operating systems use different file systems: Windows commonly uses NTFS, macOS uses APFS, and Linux may use ext4. Each file system has unique features, benefits, and limitations designed to meet the needs of the operating system and the types of storage devices it supports.

In addition, files systems specify rules for naming files and grouping them within directories. This facilitates organized data storage, simplifies file retrieval, and provides an efficient way to manage files and directories.



FILE TYPES

Text files come in various formats, each suited to different purposes. For instance, **.txt files are plain text files** containing unformatted text, ideal for simple notes or documents and can be opened by any text editor. On the other hand, **docx files are Microsoft Word documents** that include formatted text, images, and other elements. They require word processing software like Microsoft Word for viewing and editing.

Image **files also come in multiple formats.** .jpg or **.jpeg files** are commonly used for **compressed images**, making them suitable for photographs and web images where a balance between quality and file size is needed. **.png files**, or Portable Network Graphics, are i**mage files that support lossless compression** and transparency, making them ideal for web graphics and high-quality images.

For **video content, .mp4 files** are widely used for their ability to provide high-quality video and audio while maintaining relatively small file sizes. This format is compatible with most devices and media players. In contrast, .avi files, or Audio Video Interleave, can store video and audio data in a high-quality format but often result in larger file sizes.

Audio files include formats like .mp3 and .wav. The .mp3 format is popular for its compression capabilities, which reduce file size while maintaining sound quality, making it ideal for music and podcasts. Conversely, .wav files offer high-fidelity sound without compression, though they result in larger file sizes.

Executable files are used to run programs and come in formats such as .e**xe for Windows** operating systems. Double-clicking an .exe file launches the associated application or script. **For macOS, .app** files serve a similar purpose, allowing users to launch and run software.

Compressed files, such as .zip and .rar, help reduce file sizes and facilitate easier transfer or storage by compressing
one or more files into a **single archive**.

FILES - METADATA & ATTRIBUTES

File Metadata refers to the descriptive information about a file that provides context regarding its characteristics.
File Attributes, on the other hand, control the permissions and restrictions applied to a file.

Metadata includes details such as the file name, which identifies the file; the file size, indicating how much space the file occupies on the storage device; the creation date, which shows when the file was originally created; and the modification date, reflecting the last time the file was altered. Additionally, the file type specifies the format or nature of the file, such as .txt for text files or .jpg for image files. Metadata is crucial for understanding and managing files, helping users locate and organize their data effectively.

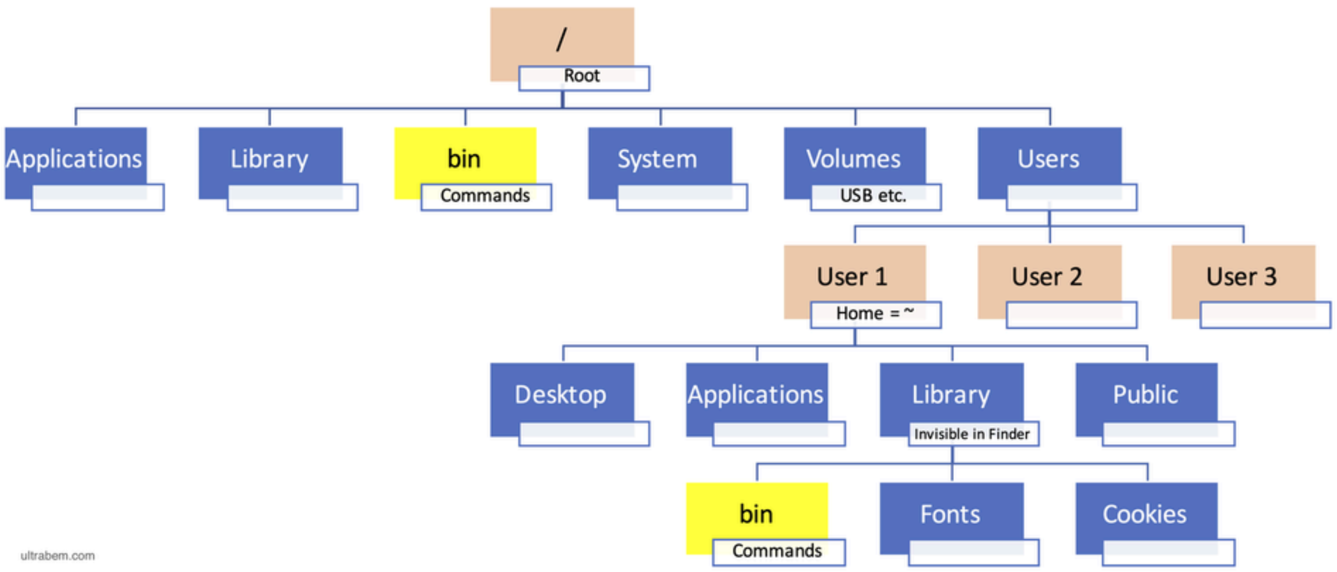
File attributes dictate how the file can be accessed, modified, or executed. Key attributes include read permission, which allows users to open and view the file; write permission, which permits users to modify or delete the file; and execute permission, which enables users to run the file as a program, relevant to executable files. Additionally, the file owner attribute identifies the user or group who owns the file, influencing who can change its attributes or permissions. File attributes are essential for enforcing access control and managing how files are interacted with within the operating system.

ROOT DIRECTORY & HIERARCHY

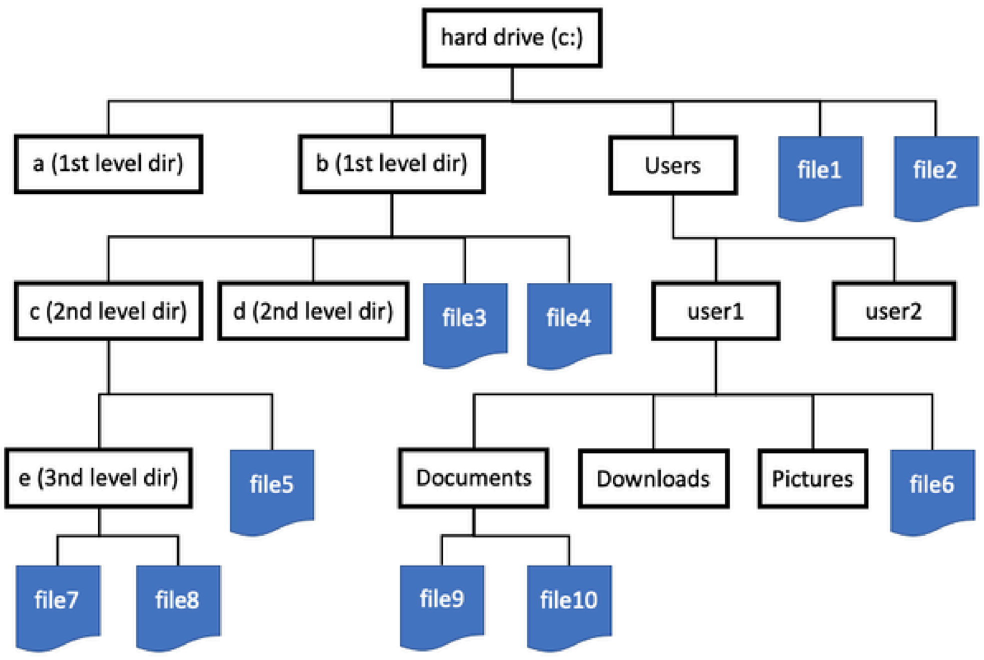
The root directory is the foundational directory at the top of a file system hierarchy. All directories and files are organized under this root - the root acts as the starting point for organizing and accessing all other files and directories on a storage device. This root directory forms the base of the directory tree structure, from which all other directories, subdirectories, and files branch out, creating a hierarchical organization. Directories are nested within one another, forming a tree-like arrangement that allows for structured data storage and retrieval.

The root directory encompasses the entire file system for a specific storage device. It includes critical directories and files necessary for both system operations and user data storage. This makes it a crucial component of the file system, containing essential system files and directories that are integral to the operating system's functionality.

In Linux and macOS, the root directory is commonly represented by a single forward slash (/).



In Windows systems, the root directory of a drive is denoted by the drive letter followed by a backslash (e.g., C:\).



These symbols - (/) and C:\ respectively, denote the top-most level of the file system hierarchy, with all other directories and files organized beneath it. For example, the root directory typically contains a directory named home, which includes subdirectories for each user.

Within a user's directory, such as user, there is a documents folder where personal files might be stored.

For instance, **C:\Users\John\Documents** indicates that "Documents" is a folder within "John's" user directory, which is under "Users" on the C: drive's root directory. Understanding the root directory is essential for efficient file system navigation, for locating files and directories accurately, and manage file paths effectively.
By knowing the structure and path representation of the root directory, users can better navigate their file systems and access data.

FILE PATHS

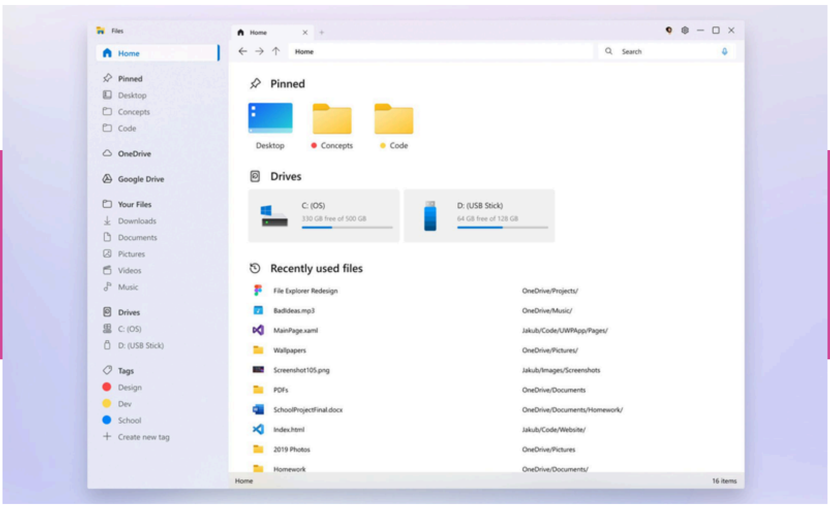
A file path is a unique identifier used to locate a file or directory within a file system. It specifies the exact location of a file or application within the file system.

It defines the sequence of directories you need to navigate through to access a specific file, providing a way to organize and manage files efficiently.

When you interact with your computer's interface graphically, such as opening folders or files, you are navigating through the underlying file system hierarchy. For instance, if you open the 'Downloads' folder on a Windows system, you are accessing the directory located at C:\Users\yourusername\Downloads.
This path reflects the hierarchical structure of the file system, starting from the root and moving through user-specific directories.

In graphical interfaces, the concept of the Current Working Directory (CWD) is represented by the folder or directory you are currently viewing. When you open a file explorer, it starts in a default directory, like your user's home folder.
Clicking through folders navigates you deeper into the directory structure, changing the view to show contents relative to your current location.

For instance, if you navigate to the Documents folder, your CWD is now Documents.

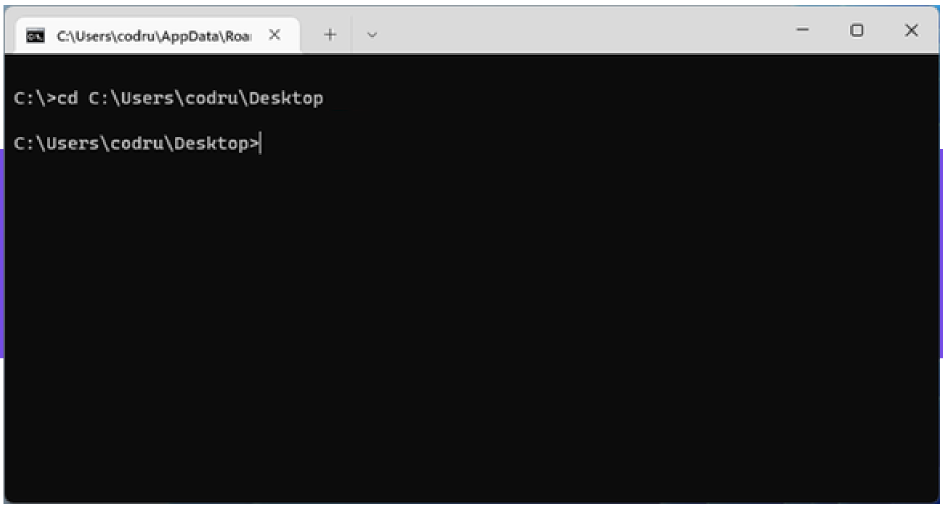


File Explorer Redesign	OneDrive/Projects/
BadIdeas.mp3	OneDrive/Music/
MainPage.xaml	Jakub/Code/UWPApp/Pages/
Wallpapers	OneDrive/Pictures/
Screenshot105.png	Jakub/Images/Screenshots
PDFs	OneDrive/Documents
SchoolProjectFinal.docx	OneDrive/Documents/Homework/
Index.html	Jakub/Code/Website/
2019 Photos	OneDrive/Pictures
Homework	OneDrive/Documents/

Opening different directories changes your current CWD, representing your position in the file system and showing you contents relative to your current location.

When you use the graphical interface, navigation is visual. **Clicking "Up" moves to the parent directory, while opening subdirectories dives deeper into the hierarchy.** Most file explorers display the current directory path somewhere in the interface, often in a **breadcrumb trail or address bar.** This shows your current position within the underlying directory structure.

When you open a terminal or command prompt, it starts in a default directory, usually your user's home directory.



```
C:\>cd C:\Users\codru\Desktop

C:\Users\codru\Desktop>
```

For example, on Unix-like systems such as Linux or macOS, the default might be **/home/yourusername**, while on Windows, it could be **C:\Users\yourusername**.

When operating the terminal, the **CWD will be displayed as part of the prompt**, indicating the directory you are currently operating in. This is the directory **where your terminal or command-line interface is currently focused.** When you enter commands, the CWD serves as your starting point for file operations and command execution. This means that any operations will be based on your current location in the directory structure.

In the command-line interface (CLI), navigation is text-based.
To navigate to different directories, you use the `` `***cd***` `` (change directory) command.
For example, if your CWD is **/home/yourusername**
and you want to move to **/home/yourusername/Documents**,
you would enter `` `***cd Documents***` ``.
After pressing enter and executing this command,
/home/yourusername/Documents becomes the new CWD.

Commands will affect the files and directories within the CWD unless you specify a different path.

TYPES OF FILE PATHS

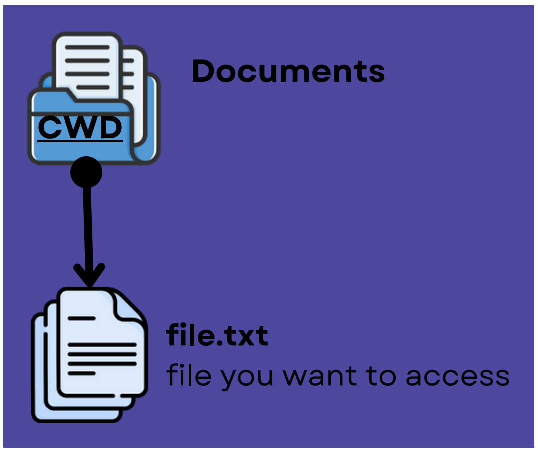
An absolute path provides the complete address of a file or directory from the root of the file system. It begins with the root directory and includes every directory in the hierarchy up to the target file or folder. This path tells the system exactly where to find a file or directory, regardless of your current location in the file system.

On a Windows system, an absolute path might look like **C:\Users\yourusername\Documents\file.txt.**

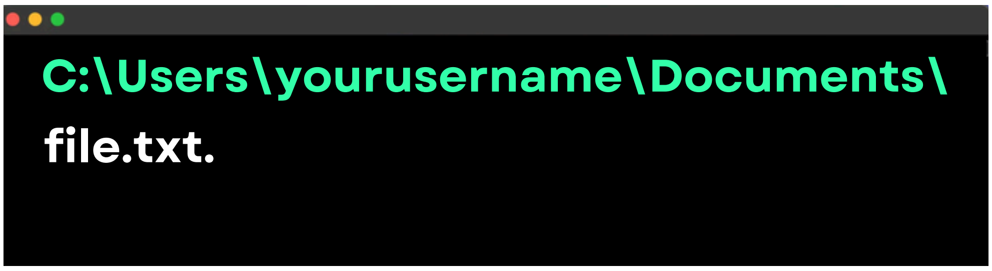
On a Unix-based system (like Linux or macOS), it would be **/home/yourusername/Documents/file.txt.**

These paths explicitly direct the system to the file's location, **no matter what your current working directory is.**

A relative path specifies a file or directory location in relation to the current working directory. It does not start from the root directory but from where you are currently navigating within the file system - current working directory.

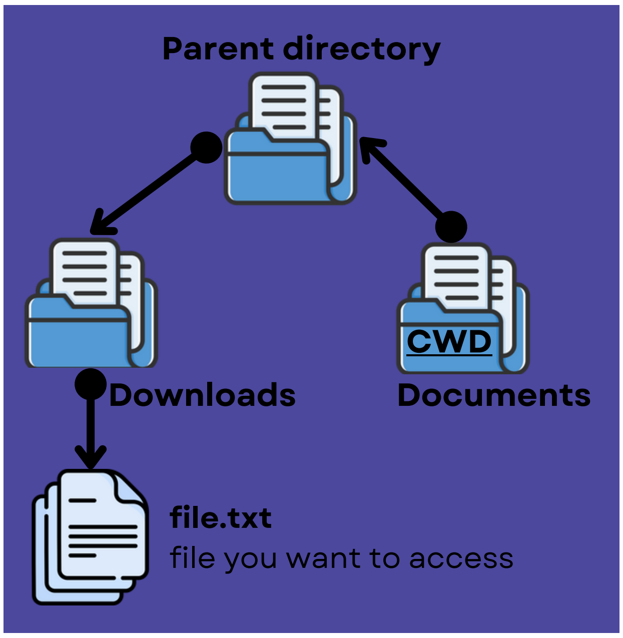


For example, if you are in the Documents directory and you want to access file.txt within that same directory, you can simply use the relative path *`file.txt.`*

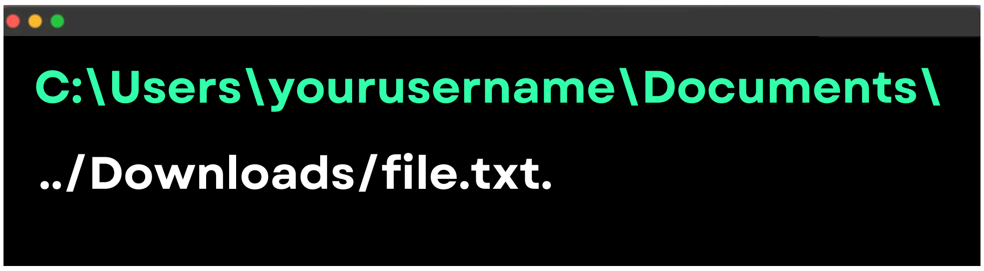


```
C:\Users\yourusername\Documents\
file.txt.
```

However, if file.txt is located in the Downloads directory, which is a sibling directory to Documents, you would use the relative path ***`../Downloads/file.txt.`***



Here `..` **represents moving up one level to the parent directory**, and then navigating **down into the Downloads** directory to access the file. This use of relative paths allows you to navigate the file system efficiently based on your current location.



```
C:\Users\yourusername\Documents\
../Downloads/file.txt.
```

RUNNING SCRIPTS & COMMANDS

When programming, file paths are used to read from, write to, or modify files. When you enter the file path for a program or executable file (like a .exe on Windows), you are directing the computer to that location and instructing it to launch the program.

For example, if you provide the path C:\Program Files\MyApp\myapp.exe and execute it, **the operating system locates this .exe file in the specified directory and starts the application.** The file path acts as a navigation tool, guiding the operating system to the correct location and then runs it with the same command.

When you **specify a .txt file** in the Command-Line Interface (CLI), such as by using a command like cat filename.txt (in Unix-like systems) or type filename.txt (in Windows), the **CLI reads and displays the contents of the file.** Text files are meant to be opened with text editors or viewed directly, and **the system simply processes and outputs the text contained within the file.**

For example, running `***cat document.txt***` in a Unix-based terminal will show the text contained in document.txt directly in the terminal window. It does not execute any commands or code.

When you specify a .exe file in the CLI, you are instructing the system to execute the file as a program. For instance, typing `***myapp.exe***` in the Command Prompt or `***./myapp.exe***` in a Unix-like terminal will start the executable program.

Executing a .exe file involves loading the program into memory and running it according to its coded instructions. This can result in a wide range of outcomes, such as opening an application window, performing system tasks, or generating output.

FILE OPERATIONS- READ, WRITE, EXECUTE, DELETE

Understanding file operations—read, write, execute, and delete—is essential for effective file management on a computer. Each operation serves a specific purpose.

Reading a file involves accessing its contents without making any changes. This action allows you to view or retrieve the information contained within the file. For example, opening a text file to read its content, such as a story or data in a log file, demonstrates reading a file.

Writing to a file means modifying its contents by adding new data or updating existing information. This changes what the file contains. For instance, saving changes to a document you've edited or appending new information to a configuration file involves writing to a file.

Executing a file means running it as a program or script. This action causes the file to perform its designated tasks or operations. For example, running an application by clicking its icon or executing a script to automate tasks demonstrates executing a file.

Deleting a file removes it from the storage device, making it no longer accessible through the file system. This action permanently eliminates the file's data unless it is recoverable from a backup or recycle bin. For instance, moving a file to the trash or recycle bin and then emptying it to permanently delete the file illustrates deleting a file.

These operations—read, write, execute, and delete—are fundamental for managing files and controlling how data is handled on a computer. Permissions govern who can perform these actions, ensuring proper access and security.

FILE OWNERSHIP

The file owner is the individual who creates or saves a file initially and holds special control over it.
Ownership confers specific rights and responsibilities.

Read
The owner can view the file's contents.

Write
The owner can modify or add to the file's contents.

Execute
For executable files or scripts, the owner can run the file.

Delete
The owner can remove the file from the system.

The file owner can also set or modify permissions for other users or groups, determining who else can read, write, execute, or delete the file.

For example, a file might be set to allow only the owner to read and write it, while others may only be able to read it. This system of ownership and permissions helps manage file security and organization, ensuring that only authorized users have access to or can modify important files.

ACCESS, PERMISSIONS & USER ROLES

File permissions control access to files on a computer, dictating which users or programs can read, write, execute, or delete files.

These permissions are typically set by the file owner or a system administrator and can be configured to suit various user roles and needs.

For instance, a file might be set to allow only its owner to read and write it, while others may only have read access. Proper management of these permissions is crucial for protecting sensitive data and ensuring files are used securely and efficiently.

User roles define the levels of access and permissions within a system or network, managing what actions users can perform on files and other resources. Common user roles include:

The administrator has comprehensive control over the system, including creating, modifying, deleting files, and altering system settings. Administrators can set permissions for other users and manage system-wide configurations.

The standard user typically has restricted access, able to read and write their own files but lacking the ability to modify or delete files owned by others or make system-wide changes.

Guests generally has very limited access, permitted only to read files or use certain applications. Guests cannot alter the system or access sensitive data.

The root directory is subject to strict access controls and permissions, which regulate who can view, modify, or manage its contents.

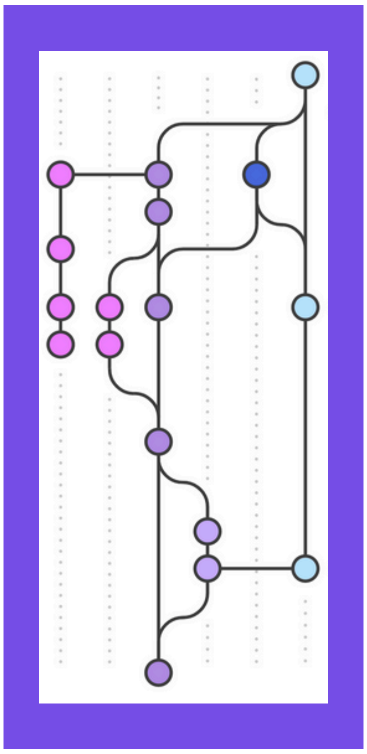
Permissions are set to ensure that only authorized users or systems can access or manipulate specific files and directories.

Privileges, or specific rights granted to users, determine the actions they can perform within the file system.

Proper management of these permissions is vital for maintaining system security and organization. System administrators or users with the appropriate privileges oversee the contents of the root directory and its subdirectories.

UNDERSTANDING VERSION CONTROL & GIT

WHAT IS VERSION CONTROL?



Git is a powerful version control tool that helps you manage and track changes to your files.

To use Git, you first **need to download and install it on your computer.** Unlike graphical applications with buttons and windows, Git is a command-line tool - you use terminals to work with Git.
This means **you interact with Git through a text-based interface** where you type commands to perform actions.
However, version control can also be integrated within an IDE for a more visual approach.
Additionally, it can be utilized through the integrated terminal in
VSCode, allowing for a combination of graphical and command-line interfaces.

Version control systems are essential for managing changes to projects over time. **They allow you to track modifications, maintain detailed history, and facilitate collaboration among multiple users.**

It helps you manage and track changes made to files and helps secure your projects by providing a comprehensive backup of your work.
Imagine working on an important document and accidentally deleting a crucial section or making an unwanted change. With version control, you can effortlessly **revert to earlier versions of your file**, undoing any mistakes and recovering lost content.

This feature is invaluable in situations such as a computer crash, accidental file deletion, or when you want to review the evolution of your document over time. Version control provides a safety net, allowing you to restore previous states and maintain the integrity of your work.

A key feature of version control systems is their ability to manage multiple contributions simultaneously. When different people work on separate parts of a document, version control systems can **seamlessly merge these changes into a single, cohesive file.**

This integration process ensures that all contributions are combined smoothly, resulting in a complete and up-to-date version of the document. By effectively handling these contributions, **version control systems prevent conflicts** and avoid overwriting each other's work, preserving the integrity of the document.

Each change is logged with an author and a timestamp, providing a clear record of the document's evolution and the context behind each update. This historical overview is essential for managing collaborative projects effectively, allowing teams to track progress, review contributions, and coordinate their efforts.

LOCAL OPERATION & VERSION CONTROL

Git operates locally on your computer.
This means all version control actions, such as branching, merging, and reverting changes, are done **directly on your machine.** You don't need a constant internet connection to use Git, as it maintains a complete history of your files right on your computer. This local framework provides a robust environment for managing your files and their history.

CLOUD- GITHUB & GITLAB

Once you have Git installed on your computer, you can use online platforms like GitLab and GitHub to improve your version control by storing your repositories in the cloud.

These platforms allow you to upload your local Git repositories to a remote server, which means all your project files, commit history, branches, and tags are saved online.

By uploading your repository to GitHub or GitLab, you ensure that your project's history is available from any computer, as long as you have the right access permissions. This makes it easy to switch computers or collaborate with others, as you can download your repository to any machine and continue working from where you left off. Additionally, these platforms provide URLs that you can share with colleagues or collaborators, granting them access to your codebase based on their permissions.

In professional settings, you'll regularly use GitHub, GitLab, or similar platforms. These tools are crucial for managing and sharing code, maintaining project history, and facilitating teamwork.

GITHUB **GITLAB**

VERSION CONTROL FEATURES

At the core of Git are several fundamental concepts that aid collaboration and maintain the integrity of the project's history.

REPOSITORIES

A repository, often abbreviated as "repo," serves as the central storage location for a project managed by a version control system.

Think of it as a directory that holds all the files related to a project, along with a comprehensive history of all modifications and updates.

Just like a folder contains various files and subfolders, a repository contains all project files and directories.

It also contains any **auxiliary files**, such as documentation, configuration files, and help files, which are crucial for understanding and managing the project.

This repository acts as a **container for the entire project**, including independent lines of development, ensuring that every change is meticulously tracked and documented - capturing the complete history of a project's evolution and storing all past and present states of the code.

The detailed record within a Git repository is comparable to metadata but far more extensive and functional.

While metadata on your computer typically includes basic information such as file size, type, and creation/modification dates, the history in a Git repo includes **in-depth details** about each change, such as, who made the change **(author and committer information)**, when the change was made **(timestamp)**, what was changed **(differences between versions)**, why the change was made **(commit messages providing context or reasons).**

COMMITS

A commit is a fundamental concept in version control, akin to a save in a word processor. It represents a specific set of changes made to the files in a repository, capturing the state of the project at a particular moment in time. Just like saving your work in a word processor, a commit ensures that your progress is recorded and can be revisited later.

Each commit includes details about the save, allowing you to track how the project evolves.
The commit history is a chronological record of all commits made in a repository, and serves as a detailed log of the project's development. Each entry in the commit history includes key information about the commit.

Commit Hash - A unique identifier for each commit.

Author Information - Who made the commit.

Date and Time - When the commit was made.

Commit Message - A description of the changes made in that commit.

BRANCHES

The main branch is the default branch created when you first create - ie "initialize" - a new repository. It represents the stable and most up-to-date version of your project. **This branch is typically where the final, production-ready code resides** - the one that goes live on a website, for example, which users interact with.
It is the branch that most developers consider the primary code base of development, where the most reliable and tested code should be stored. Developers and teams work to ensure that this branch remains free of errors and issues.

The default branch name in Git repositories can be either **main or master**, depending on the context and repository setup. Historically, master was the default branch name used by Git, but many projects and platforms have transitioned to using main as the default branch name to adopt more inclusive language. If you encounter a repository using main, and another using master, **they are functionally the same; only the name differs.**

- **Central Repository for Stable Code**
  The main branch serves as the repository for the stable and production-ready code. It is the branch that is often made available to end-users. By keeping the main branch stable and working, teams ensure that the code in this branch is functional and reliable.

- **Integration Point for Features and Fixes**
  When developers complete work –such as features or bug-fix –they add their changes to the main branch. This process integrates new features or fixes into the stable codebase, ensuring that these updates become part of the project's primary version.

- **Reference Point for New Development**
  The main branch acts as a reference point for creating new branches. Branching in Git is best described as creating a parallel version of the main branch. Think of it like this: If the main branch is like a timeline of your project, creating a new branch is like drawing a new line that starts from the same point but can diverge in different directions, where both lines can evolve independently.

Creating branches, or branching, is a powerful feature that allows developers to create **separate lines of development** within your project - this allows you to **diverge from the main codebase and use it as a starting point for new developments**, without altering the original codebase.

On the parallel version, you can work on specific features, bug fixes, or experiments without **affecting the main branch.**

For example, if you’re tasked with developing a new feature like a login page for a website, you might create a branch named feature-login-page.

This branch starts with a **complete copy of the main codebase**, allowing you to ensure that the new login feature **integrates seamlessly** with the rest of the website. You can test how the login feature interacts with existing functionality and make adjustments as needed.

Developers typically create feature branches or bug-fix branches off of the main branch. This means that work on new features or fixes **begins with t latest version of the stable code**, ensuring that new developments are bu on top of the most current codebase.

MERGING

Once work on a branch is complete, the changes need to be integrated into the main project. This process is known as merging. Merging combines the changes from one branch into another, typically incorporating them into the main branch.

For example, after finishing work on the "feature-login-page" branch, you would merge it into the main branch to add the login functionality to the project. The merge process ensures that both the code for the new feature and the code with the recent updates in the main branch are preserved.

For example, if the main branch has recent improvements to the website's layout or functionality (that were not present when you created your feature branch), these changes are preserved and combined with the login feature you've developed.

After merging, the main branch represents the updated codebase that includes both the **current existing functionality and the new changes from the feature branch.**
There is no longer a separate, isolated codebase for the feature; **its changes are now part of the main codebase.**

For most merges, **Git can automatically integrate changes** from one branch into another. This is the case when the changes do not overlap or conflict.

Git compares the differences between the branches and applies the changes to the target branch without any issues.

This comparison includes added, deleted, and modified lines of code.
Git then applies these differences to the **target branch (e.g., main).**
It takes the changes from the **source branch (feature-login-page)** and **integrates them with the existing code in the target branch.**

This means that new lines of code, changes to existing lines, and deletions in the feature branch are incorporated into the main branch.
If there are no overlapping changes—meaning that the feature branch and the main branch have modified different parts of the code or different files—**Git handles this integration seamlessly.**

The feature branch still exists separately and contains a copy of the codebase from the main branch at the point when it was created, plus any specific changes you made for the new feature, such as the login page. It remains as it was before the merge, which means it still contains the code developed for the login feature.
This branch can be kept for reference or deleted if it is no longer needed.
Merging changes into the main branch often involves **code reviews** - this integration typically follows a review process to ensure code quality.
Since the main branch is where all changes converge, it becomes the focal point for resolving any conflicts between different branches and reviewing code before it becomes part of the stable version.

CONFLICTS

Sometimes, changes in the feature-login-page branch might overlap with changes that were made to the main branch since the branch was created.

These conflicts occur when both branches have modifications in the same areas of code.

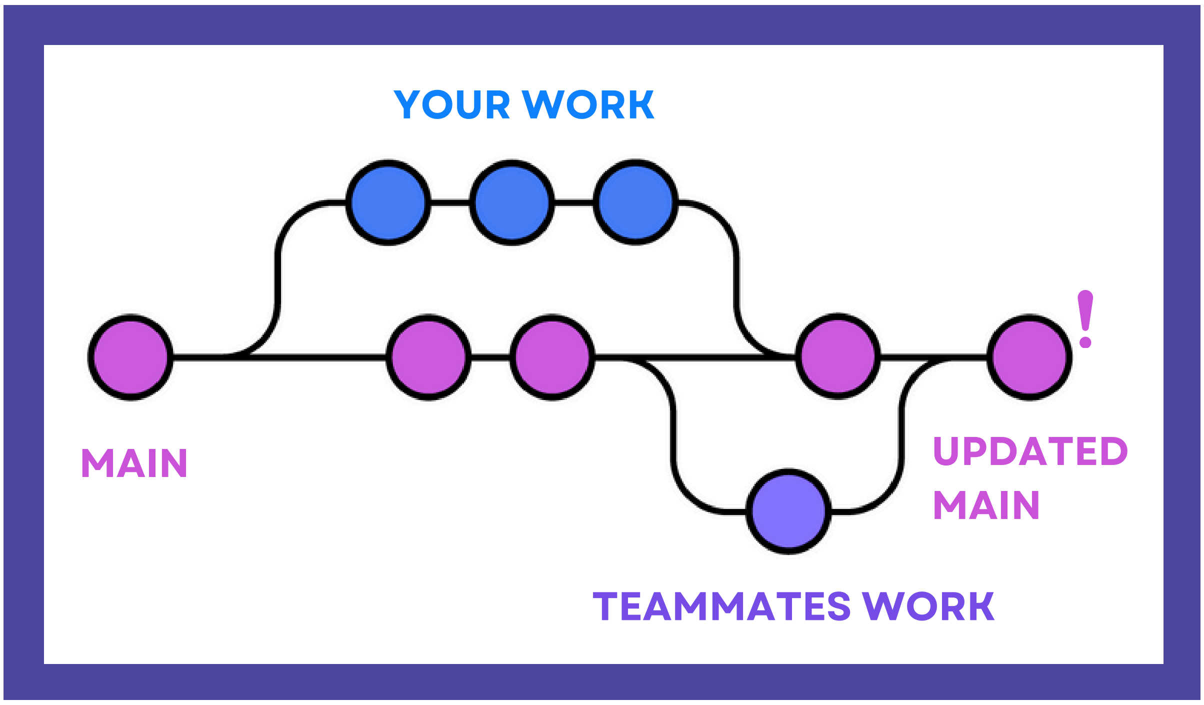
For instance, if someone updated the layout of the homepage on the main branch while you were working on the login feature, there might be conflicting changes.

If Git finds **conflicts—places where the changes from your feature branch and the main branch can’t be automatically merged—**it will ask you to resolve these conflicts.
This means you’ll need to manually review the code and decide how to combine the changes. You need to **open the files with conflicts and manually review the changes.**

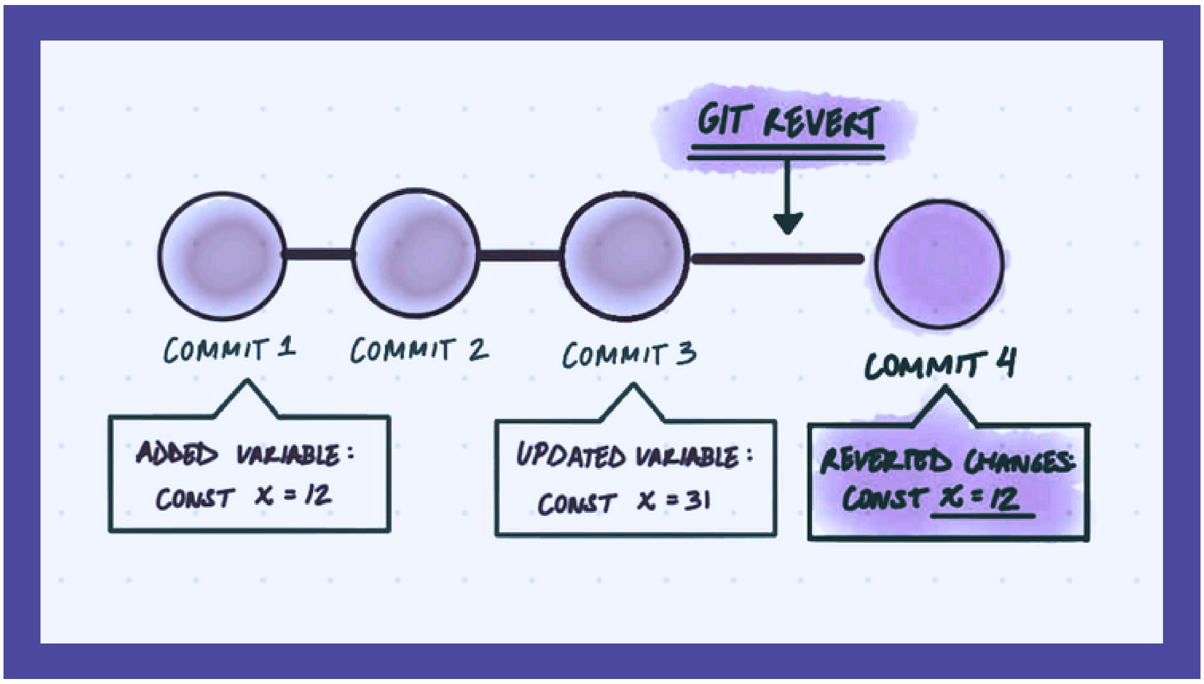
Git will show both versions of the conflicting code, allowing you to decide how to integrate them. For example, you might need to manually combine the layout changes with the login feature to ensure both are correctly represented.

GLOSSARY & VISUAL EXPLANATIONS

BRANCHING AND MERGING



REVERTS



GIT REVERT
COMMIT 1
COMMIT 2
COMMIT 3
COMMIT 4
ADDED VARIABLE: CONST X = 12
UPDATED VARIABLE: CONST X = 31
REVERTED CHANGES: CONST X = 12