GHOUSIA INSTITUTE OF TECHNOLOGY FOR WOMEN

NEAR DAIRY CIRCLE, HOSUR ROAD, BENGALURU-560029, KARNATAKA
AFFILIATED TO VTU., BELAGAVI, RECOGNIZED BY GOVERNMENT OF KARNATAKA & A.I.C.T.E., NEW DELHI
WWW.GITW.IN

Dr. NAVEED
Assistant Professor
GITW-Bengaluru

Project Management with Git-BCS358C

GitHub is a web-based platform used primarily for version control and collaborative software development. It is built around Git, an open-source version control system that tracks changes in files and allows multiple people to work on a project simultaneously without interfering with each other's work.GitHub is widely used by individual developers, teams, and large organizations to collaborate on projects, share code, and contribute to open-source software.

MORE INFORMATION
www.github.com

THIRD SEMESTER
B.E DEGREE 2024

GHOUSIA INSTITUTE OF TECHNOLOGY FOR WOMEN

Near Dairy Circle, Hosur Road, Bengaluru ,Karnataka 560029

Affiliated to VTU., Belagavi, Recognized by Government of Karnataka & A.I.C.T.E., New Delhi

PROJECT MANAGEMENT WITH GIT

(BCS358C)

As per 2022 Scheme Syllabus Prescribed by V.T.U.

For

THIRD SEMESTER

COMPUTER SCIENCE & ENGINEERING/INFORMATION SCIENCE & ENGINEERING

(Bachelor of Engineering)

Dr.NAVEED M.Tech., PhD.

Assistant Professor

Department of Computer Science & Engineering

GHOUSIA INSTITUTE OF TECHNOLOGY FOR WOMEN

Near Dairy Circle, Hosur Road, Bengaluru-560029, KARNATAKA
Affiliated to VTU., Belagavi, Recognized by Government of Karnataka & A.I.C.T.E., New Delhi

PROJECT MANAGEMENT WITH GIT / BCS358C / THIRD SEMESTER / B.E DEGREE

CERTIFICATE

This is to certify that Miss._____bearing USN_____of_____Branch completed the academic requirements for the practical course work titled "**PROJECT MANAGEMENT WITH GIT/ BCS358C"**ofTHIRD SEMESTER B.E, prescribed by Visvesvaraya Technological University , Belagavi, for the academic year_____. The details of Mark's obtained by the candidate is given below.

Sl.No	Particulars		Max.Marks (Execution+Record)	Marks Obtained		Page No	Staff Sign
1	Expt-01	Basic Setup and Creation of a New Repository	5+5 = 10			11	
2	Expt-02	To add README.md file into the Repository				15	
3	Expt-03	Creating and Managing Branches				19	
4	Expt-04	Merging of Feature-Branch into the Master Branch				26	
5	Expt-05	Updating of files in the feature-branch				28	
6	Expt-06	Light Weight and Annotated Tags				34	
7	Expt-07	Analyzing GIT History				39	
8	Expt-08	GIT Cherry-pick and Revert				44	
9	Expt-09	Git in VS Code				56	
10	Expt-10	VS Code and Github (cloning of repository)				65	
11	Expt-11	VS Code and Github (creating of new repository)				78	
12	Assignment Experiments		20+20 = 40			86	
Total Marks-A			150				
Test Marks-B			100				
Final Internal Assessment Mark's.			[(A*30)/150] + (B*20%) = **50**				

Internal Assessment Marks Awarded in Words: _____

Signature of Staff Incharge with Date:

Project Management with Git		Semester	3
Course Code	BCS358C	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0: 0 : 2: 0	SEE Marks	50
Credits	01	Exam Marks	100
Examination type (SEE)	Practical		

Course objectives:

- .To familiar with basic command of Git
- To create and manage branches
- To understand how to collaborate and work with Remote Repositories
- To familiar with virion controlling commands

Sl.NO	Experiments
1	**Setting Up and Basic Commands** Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.
2	**Creating and Managing Branches** Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."
3	**Creating and Managing Branches** Write the commands to stash your changes, switch branches, and then apply the stashed changes.
4	**Collaboration and Remote Repositories** Clone a remote Git repository to your local machine.
5	**Collaboration and Remote Repositories** Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.
6	**Collaboration and Remote Repositories** Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.
7	**Git Tags and Releases** Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.
8	**Advanced Git Operations**

	Write the command to cherry-pick a range of commits from "source-branch" to the current branch.
9	**Analysing and Changing Git History** Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?
10	**Analysing and Changing Git History** Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."
11	**Analysing and Changing Git History** Write the command to display the last five commits in the repository's history.
12	**Analysing and Changing Git History** Write the command to undo the changes introduced by the commit with the ID "abc123".

Course outcomes (Course Skill Set):

At the end of the course the student will be able to:

- Use the basics commands related to git repository
- Create and manage the branches
- Apply commands related to Collaboration and Remote Repositories
- Use the commands related to Git Tags, Releases and advanced git operations
- Analyse and change the git history

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

- Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.
- Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.
- Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).
- Weightage to be given for neatness and submission of record/write-up on time.
- Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.
- In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.
- The suitable rubrics can be designed to evaluate each student's performance and learning ability.
- The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

- SEE marks for the practical course are 50 Marks.
- SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.
- The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedule mentioned in the academic calendar of the University.

- All laboratory experiments are to be included for practical examination.
- (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners.
- Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.
- Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.

- General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)
- Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.

The minimum duration of SEE is 02 hours

Suggested Learning Resources:

- Version Control with Git, 3rd Edition, by Prem Kumar Ponuthorai, Jon Loeliger Released October 2022, Publisher(s): O'Reilly Media, Inc.
- Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, https://git-scm.com/book/en/v2
- https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013094443347369984278 2_shared/overview
- https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013301347121774592119 26_shared/overview

INRODUCTION TO GIT

Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It was created by Linus Torvalds in 2005 for the development of the Linux kernel.

1.0 About Version Control

Version control is a system that keeps track of changes made to files over time. It records every modification so that you can go back and look at older versions whenever needed. Think of it as a "time machine" for your project files.

Although version control is commonly used by software developers to manage source code, it can actually be used with almost any type of file — documents, images, designs, and more.

For example, if you are a web designer and want to save every version of a webpage layout you create, a Version Control System (VCS) is the perfect tool.

A VCS allows you to:

- ✓ Revert a single file to an earlier version if something breaks.
- ✓ Restore the entire project to a previous state if needed.
- ✓ Compare changes made over time and see what has been added or removed.
- ✓ Identify who made a change, when it was made, and why — which is helpful when debugging issues.
- ✓ Recover files if they are accidentally deleted or corrupted.

The best part is that version control does all this with very little extra effort once set up.

Example: Imagine you are working on a C++ project with three teammates. Each of you is writing different functions for a single program. Without version control, you might send code files back and forth through email or chat, leading to confusion about which file is the latest. Someone might accidentally overwrite another person's work, or you might lose an important piece of code after a mistake.

With a VCS like Git, all of you can store your project in a shared repository (e.g., GitHub). Every time you make a change, you "commit" it with a message explaining what was modified. If a new change causes a bug, you can easily roll back to the previous version. You can also see exactly which teammate introduced the bug, making it faster to fix.

This makes teamwork more organized, reduces mistakes, and ensures that no work is lost — even if someone's computer crashes.

1.1 Local Version Control Systems

The figure shows how a Local Version Control System (VCS) works on a computer.

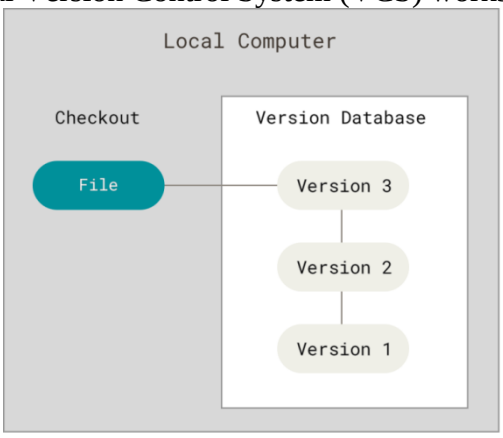


Figure1.Local version control diagram

On the left side, there is a file that you are working on. Instead of saving it normally or creating multiple copies in different folders, the Local VCS stores it in a Version Database (shown on the right side of the figure).

Inside this database, every time you save changes through the VCS, a new version is created — Version 1, Version 2, Version 3, and so on — forming a history of your work. If you ever need an earlier version, you can easily "check out" that version and restore it. This way, no work is permanently lost, and you can always review what changed over time.

Before VCS tools were common, people simply copied their files into new folders with different names (for example, project_v1, project_v2). This was simple but very error-prone — you could easily overwrite the wrong file or lose track of which version was the latest.

Local VCS tools solved this by keeping a centralized database of file changes and automatically managing versions for you.

Example: Imagine you are writing a Java program.

- ✓ Version 1 prints "Hello World."
- ✓ Version 2 adds user input functionality.
- ✓ Version 3 adds error handling.

If Version 3 has a bug, you can quickly go back to Version 2, see the differences, and fix the issue.

One of the earliest and most popular local VCS tools was RCS (Revision Control System). It is still available on many computers today.

- How it works: RCS stores *patch sets* — the differences between each version — instead of storing full copies of every file.
- When you want an older version, RCS applies these patches step by step to reconstruct the exact version of the file.
- This makes it efficient in terms of storage space, since only the changes are saved. RCS was widely used by programmers before more advanced tools like Git and Subversion (SVN) became popular.

1.2 Centralized Version Control Systems

A Centralized Version Control System (CVCS) is a version control setup where a single, shared repository (central server) stores all the versioned files and the complete history of a project. Developers connect to this central repository to check out files (download a copy) and commit their changes (upload updates). This approach became popular with tools like CVS, Subversion (SVN), and Perforce, as it solved the problem of collaboration that existed with local version control systems.

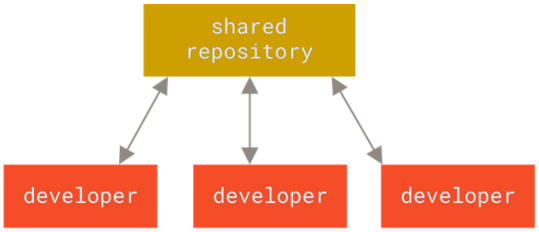


Figure2.Centralized version control diagram

The figure shows how a Centralized Version Control System (CVCS) works. There is a shared repository at the top, which stores all the project files and their history. Multiple developers connect to this shared repository — they download the latest version of the files (checkout) and upload their changes (commit) so that everyone stays in sync.

One major advantage of CVCS is that it keeps all developers in sync because everyone is working with the same central source of truth. This makes it easy for team members to know what others are working on. Administrators also benefit because they can control who has access to read, write, or modify code, and maintaining one central database is easier than managing multiple local databases.

However, centralized systems have a few drawbacks. The biggest issue is the single point of failure — if the server goes down, developers cannot collaborate, commit changes, or even access the latest version of the project until the server is back online. Worse, if the central database becomes corrupted and no proper backups are kept, the entire project history may be lost.

Another limitation is that CVCS relies on a constant network connection, which means collaboration slows down or stops completely if the internet or network connection is poor.

For example: Imagine a team of three developers working on a Python web application using SVN. Each developer downloads the latest code from the shared repository, works on their assigned module, and then commits their changes back to the central server. Other developers can then update their local copies to stay up-to-date. But if the central server crashes, no one can commit new code or get updates until it is fixed.

1.3 Distributed Version Control Systems:

A Distributed Version Control System (DVCS) is a more advanced form of version control where every developer's computer contains a full copy of the repository, including its entire history. Tools like Git, Mercurial, and Darcs follow this model.

Unlike centralized systems where the server holds the only complete version history, in DVCS every clone (copy) of the repository acts as a complete backup. This means that if the main server fails, any developer can restore the entire project by simply sharing their local repository.

Another major benefit of DVCS is its flexibility. It allows developers to collaborate in multiple ways — not just through a single central server. You can have several remote repositories and set up different workflows, such as hierarchical or peer-to-peer collaboration. This makes teamwork faster, safer, and more resilient.

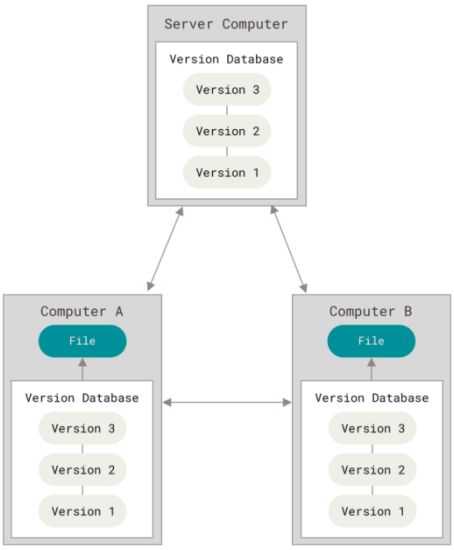


Figure3.Distributed version control diagram

The figure shows a server computer at the top and two developer computers (Computer A and Computer B) at the bottom.

- ✓ Each computer has its own full version database, containing Version 1, Version 2, and Version 3 of the project.

- ✓ Changes can be shared not just with the server but also directly between developers (peer-to-peer).
- ✓ If the server is lost, any developer can restore the entire history because they have a complete copy.

1.4 A Short History of Git

Like many great innovations, Git was born out of necessity, conflict, and creativity.

The Linux kernel is the core part of the Linux operating system — it manages the computer's hardware (CPU, memory, storage, devices) and allows software applications to run. It is one of the largest and most important open-source projects in the world, with thousands of developers contributing from across the globe.

In its early years (1991–2002), Linux kernel developers used to share changes by exchanging patches and archived files manually — a slow and error-prone process. In 2002, the Linux community adopted a proprietary distributed version control system called BitKeeper, which made collaboration much easier.

However, in 2005, the free-of-charge status of BitKeeper was revoked after a dispute between the Linux kernel community and the company that developed BitKeeper. This created a major problem — developers suddenly had no reliable tool to manage their code.

In response, Linus Torvalds (the creator of Linux) decided to build a completely new version control system from scratch. His goals were ambitious:

- ✓ *It had to be fast and efficient.*
- ✓ *It should have a simple design so developers could understand it easily.*
- ✓ *It must support non-linear development — meaning thousands of parallel branches could exist at the same time.*
- ✓ *It needed to be fully distributed, so every developer had a complete copy of the project.*
- ✓ *It had to handle very large projects like the Linux kernel without slowing down.*

The result was Git, released in 2005. Over time, Git has matured into one of the most popular and powerful version control systems in the world. It is incredibly fast, can handle huge projects, and has a powerful branching and merging system that makes collaborative software development smooth and efficient.

Example: Imagine you and 10 other students are working on a large C++ project for a university assignment. Without Git, you would have to manually send updated files over email or chat, merge everyone's changes by hand, and keep track of which file version is the latest — a nightmare for big teams.

With Git, every student can work on their own copy of the project, create separate branches for new features, and later merge them back into the main branch. If two people make changes at the same time, Git can intelligently combine the changes or alert you to resolve conflicts. Even if the main server fails, any student's repository can be used to restore the entire project history.

This is exactly why Git became the backbone of huge projects like the Linux kernel — it makes collaboration on complex software much faster, safer, and more organized.

1.5 What is Git?

Git is a version control system (VCS) — a tool that helps developers track and manage changes in their code over time. If you understand what Git is and how it works at a basic level, you will find it much easier to use effectively.

Many people compare Git with older systems like CVS, Subversion (SVN), or Perforce. While Git looks similar on the surface, it actually works very differently. To use Git confidently, try to forget how those older systems work and focus on Git's unique approach.

1.6 Snapshots, Not Differences

Most older version control systems store changes as a list of edits to files over time — like keeping a record of what changed line by line (called delta-based version control).

Git does not work this way. Instead, Git treats your project like a series of snapshots of the entire project at different points in time.

- ✓ *Every time you commit in Git, it takes a snapshot (picture) of all your files and remembers it.*
- ✓ *If a file has not changed since the last commit, Git does not save it again — it just links to the previous copy to save space.*

This design makes Git faster and more reliable. You can think of Git as a mini file system with powerful tools built on top of it.

Example: Imagine you are building a Python web application with your team. Every time you reach a stable state — say you complete the user login feature — you take a commit. Git saves a snapshot of the entire project at that moment. Later, if something breaks, you can go back to this exact snapshot and recover your code.

1.7 Nearly Every Operation is Local

One of Git's biggest advantages is that almost everything happens on your own computer, not on a remote server.

- ✓ *You don't need to be online to view history, check differences between files, or commit new changes.*
- ✓ *All the project's history is stored on your local machine, so Git can perform operations almost instantly.*

Example: You are working on a C++ project while traveling on a train with no internet. With Git, you can still commit your changes locally. When you get an internet connection later, you can push those commits to a shared repository like GitHub so your teammates can see them.

1.8 Git Has Strong Data Integrity

Every file and commit in Git is identified by a SHA-1 hash — a unique 40-character code calculated from the file's contents.

- ✓ *This means if a file changes by even a single character, Git will notice.*
- ✓ *It is nearly impossible to lose data silently — Git can detect corruption or accidental changes.*

Example: If a teammate accidentally modifies a configuration file in your Java project, Git will immediately flag that the file has changed. You can compare the old and new versions and decide whether to keep the change.

1.9 Git Generally Only Adds Data

Git rarely deletes data — most actions just add new data to the project history.

- ✓ This makes Git very safe: once you commit, your snapshot is stored permanently.
- ✓ Even if you make a mistake later, you can usually recover older versions.

Example: If you experiment with a new machine learning model and it fails, you can always return to your last working commit without losing anything.

1.10 The Three States of Git

Git organizes your files into three states:

1. Modified – You have changed the file, but it is not yet marked for saving in Git's history.
2. Staged – You have marked the modified file to be included in your next commit.
3. Committed – Your changes are permanently stored in Git's local database.

These states are tied to three main parts of a Git project:

- ✓ Working Tree: The actual files on your computer that you can see and edit.
- ✓ Staging Area (Index): A temporary space where you prepare changes before committing.
- ✓ Git Directory: The hidden .git folder where Git stores all snapshots, branches, and history.

1.11 Typical Git Workflow

1. Modify files in the working tree.
2. Stage changes you want to save using git add.
3. Commit them using git commit, which creates a permanent snapshot in the Git directory.

Example: Suppose you are developing a JavaScript website:

You edit index.html and style.css to improve the UI (modified).You decide only index.html should be saved for now, so you run git add index.html (staged).You run git commit -m "Updated homepage layout" (committed).If you later realize your CSS was wrong, you can fix it and commit it separately. This gives you a clean, well-organized history.

1.12 Uses of Git:

Use of Git	Explanation	Example
1. Collaboration	Multiple developers can work on the same project at the same time without overwriting each other's work. It is very popular for open-source projects where developers worldwide contribute.	Team of developers building a mobile app together; open-source projects like Linux kernel or ReactJS.
2. Tracking Changes	Git keeps a complete history of every change made to the code. You can review past versions, undo mistakes, and see who made which change and when.	Rolling back to an earlier version when a new code change introduces a bug.
3. Branching & Merging	Developers can create separate branches for new features, bug fixes, or experiments. After testing, they can merge the changes back into the main code safely.	Creating a branch feature-login for a new login system and merging it into main after review.
4. Backup & Restore	Each user has a full copy of the repository on their computer. Remote repositories (GitHub, GitLab, Bitbucket) serve as additional backups.	Recovering project files from your local Git repository even if the server is down.
5. CI/CD (Automation)	Git integrates with Continuous Integration/Continuous Deployment (CI/CD) tools to automatically test, build, and deploy code when changes are pushed.	Automatic unit testing and deployment of a Django web app when code is pushed to GitHub.
6. Documentation	Git repositories include README files and wikis that explain the project, setup steps, and contribution guidelines.	A repository containing a README.md file that describes how to run the project locally.

1.13 Installing Git

Before you can start using Git, you need to install it on your computer. Even if Git is already installed, it's a good idea to update it to the latest version to get new features and bug fixes. You can install Git in three ways — by downloading it as a package, using an installer, or by downloading the source code and compiling it yourself.

1.14 Installing on Windows

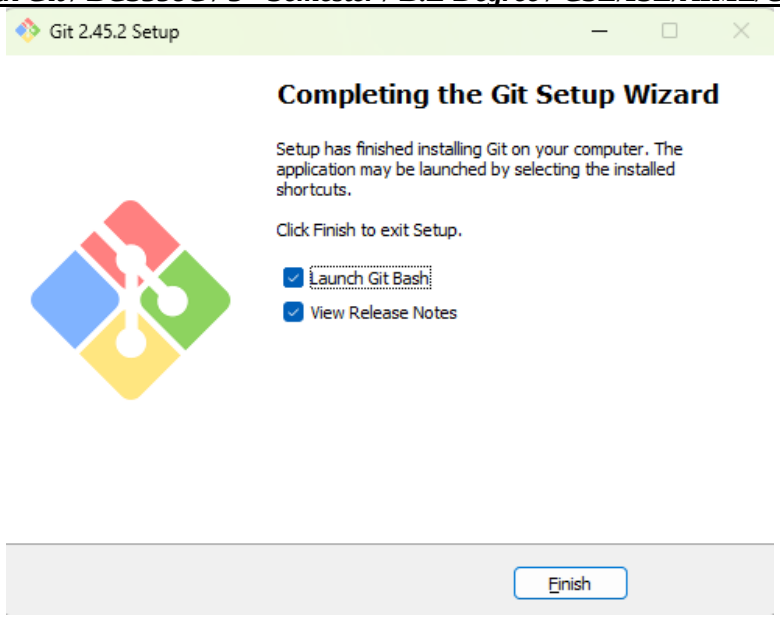
On Windows, the easiest way to install Git is to download the official installer. Just visit https://git-scm.com/download/win, and the download will start automatically. This installer is part of a project called Git for Windows, which provides extra features to make Git work well on Windows systems. You can learn more about this project at https://gitforwindows.org.

If you prefer an automated way, you can use the Chocolatey package manager to install Git. Keep in mind that the Chocolatey package is community-maintained, so it may not always be as up to date as the official installer.

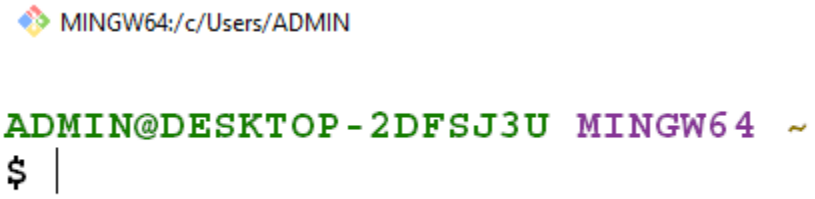
OR to Download click below link:

https://drive.google.com/file/d/1H9ZMW2lZnqNngLv-PTXSbUUXPas56IVw/view?usp=sharing

Go through the installation process by clicking Next for all the steps at the last click on Launch GitBash



Click on Finish.



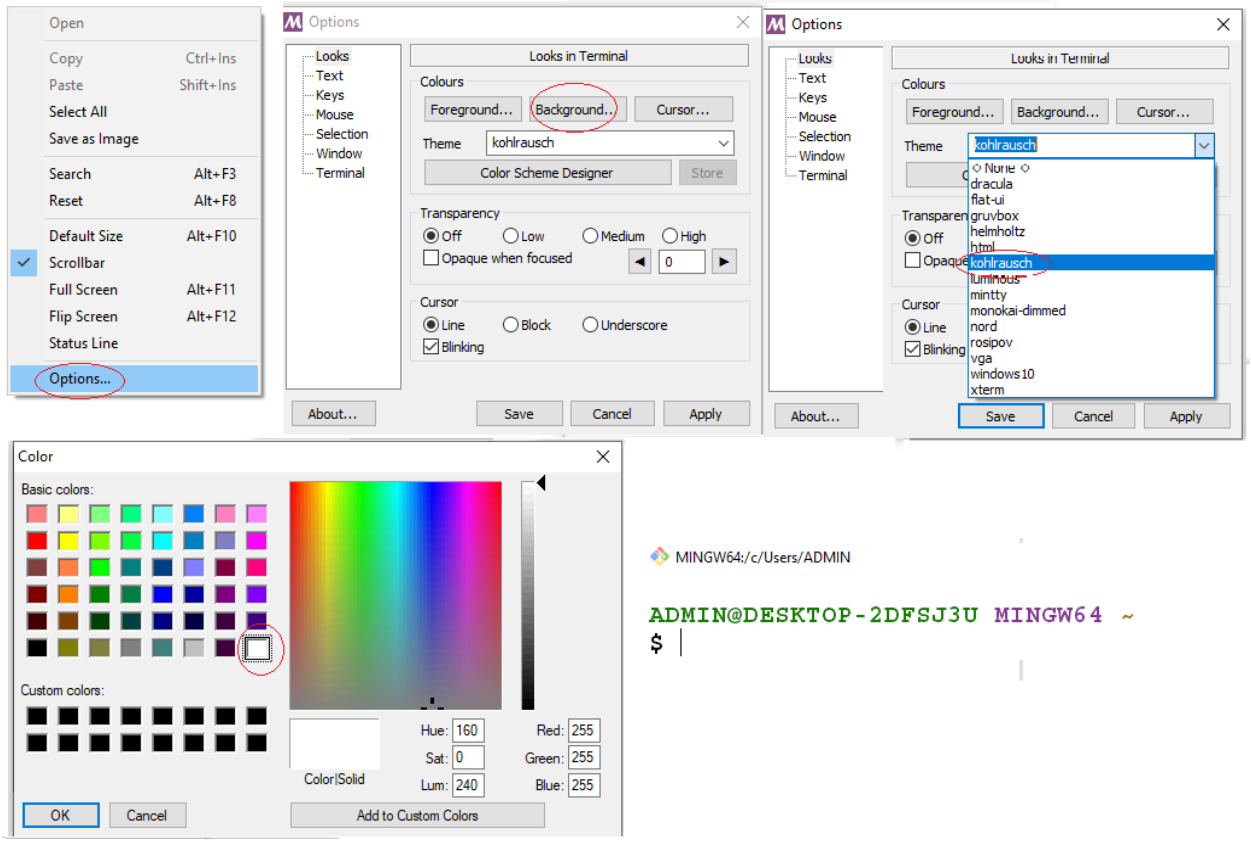
EXPERIMENT-01 Basic Setup and Creation of a New Repository

Aim:

1. To create a new repository "1WT23CS000" under any disc drive such as Z drive and also a sub repository "aboutMyself".
2. To make the default branch as master branch in some systems it is main branch.
3. To launch the git and enter the user configurations like name, email ID.

First-TimeGitSetup

To make the background color white and to make the theme Kohlrausch.



Configuring Git After Installation

Once Git is installed on your computer, the next step is to configure your Git environment. These settings help personalize Git for you and are usually set only once per computer. They will remain in place even after you update Git, but you can change them anytime by running the same commands again.

Setting Your Identity

The first thing you should configure is your **username** and **email address**. This is important because every Git commit you make will be permanently linked with this information.

You can set your global username and email with the following commands:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Using the `--global` option means Git will use this name and email for all repositories on your system.

If you want to use a **different name or email address** for a specific project, navigate to that project folder and run the commands **without** the `--global` option. This will override the global settings for that particular project only.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config --global user.name "Dr.NAVEED"

ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config --global user.email naveed.gce@gmail.com
```

Checking Your Git Settings

After setting up your username and email, you may want to confirm that your configuration is correct. You can do this by running the following command:

`git config --list`

This command will display all the configuration settings that Git is currently using on your system — including your username, email, and other preferences.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config --list
```

It will show:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Dr.NAVEED
user.email=naveed.gce@gmail.com
```

It will show all details and at the end user name and email ID will be highlighted. If only user name is required then git config user.name will be used.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config user.name
Dr.NAVEED
```

To check email of the user use git config user.email

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ git config user.email
naveed.gce@gmail.com
```

To clear the screen, use clear. Alternatively, you can use the keyboard shortcut Ctrl + L which

also clears the screen in most terminal emulators, including Git Bash.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ clear
```

To Create a New Repository (Folder): By name "1WT23CS000".
First specify in which drive of your computer you want to create the new repository for Example 'Z' Drive

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ cd /z
```

In the next line you can see that Z drive is highlighted.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z
$
```

To create a new repository by name 1WT23CS000

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z
$ mkdir 1WT23CS000
```

To switch to the new repository

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z
$ cd 1WT23CS000
```

You can see now

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000
$
```

To create a sub folder " aboutMyself" inside the main folder

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000
$ mkdir aboutMyself
```

To shift to the subfolder

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000
$ cd aboutMyself
```

Now you can see

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself
$
```

Once the repository is created initialize the git:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself
$ git init
```

It will make master branch as default branch.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself
$ git init
Initialized empty Git repository in Z:/1WT23CS000/aboutMyself/.git/

ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$
```

OUTPUT:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git config user.name
Dr.NAVEED

ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git config user.email
naveed.gce@gmail.com
```

EXPERIMENT No: 02 To add README.md file into the Repository

Aim:

1. To add README.md file into the repository /z/1WT23CS000/aboutMyself under master branch.
2. To add the contents given below:

```
Title: Dr.
Full Name: Naveed
USN: 1WT23CS000
Semester: Third
Section: A
Subject Name: Project Management with GIT
Subject Code: BCS358C
Academic Year: 2024-25
Mobile No: 9620483405
Email ID: naveed.gce@gmail.com
```

3. To commit with a message "Contents updated successfully"

First get back to the repository by using:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 ~
$ cd /z/1WT23CS000/aboutMyself
```

Now the repository is ready:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$
```

To Add README.md file inside the sub folder

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git add README.md
```

To check the file added:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git ls-files
```

It will show:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git ls-files
README.md
```

Note:Some times if it shows some error like Z drive is unsafe directory then to make it safe use:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git config --global --add safe.directory z:/
```

If git add command shows some error, then use:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ echo > README.md
```

This command creates a file named README.md but it will be untraced. To make it traced use

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git add README.md
```

To open the README.md file.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ nano README.md
```

An empty file will open. Type the below sample data:

MINGW64:/z/1WT23CS000/aboutMyself

```
  GNU nano 8.1                                          README.md
Title: Dr.
Full Name: Naveed
USN: 1WT23CS000
Semester: Third
Section: A
Subject Name: Project Management with GIT
Subject Code: BCS358C
Academic Year: 2024-25
Mobile No: 9620483405
Email ID: naveed.gce@gmail.com
```

To save:Ctrl+S
To Exit: Ctrl+X
To check the contents of the README.md File

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ cat README.md
```

It will highlight the contents of the file

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ cat README.md
Title: Dr.
Full Name: Naveed
USN: 1WT23CS000
Semester: Third
Section: A
Subject Name: Project Management with GIT
Subject Code: BCS358C
Academic Year: 2024-25
Mobile No: 9620483405
Email ID: naveed.gce@gmail.com
```

To edit or modify the contents of the README.md file the same above procedure may be adopted.
The data will be updated. But it will not be committed in the git. If we check the status:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git status
```

It will show changes to be committed with a new file README.md highlighted in green color.

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md

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:   README.md
```

To commit the above with a message "Added README.md file with basic data successfully"

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git add .
```

And then use

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git commit -m "Added README.md file with basic data successfully"
```

It will show you the message:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git commit -m "Added README.md file with basic data successfully"
[master (root-commit) 4dda3ea] Added README.md file with basic data successfully
 1 file changed, 11 insertions(+)
 create mode 100644 README.md
```

If you want to check the status:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git status
```

It will show you nothing to commit, working tree clean:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git status
On branch master
nothing to commit, working tree clean
```

`git add .` is used in Git to stage all changes in the current directory and its subdirectories for the next commit. When you make changes to files in your Git repository, these changes are initially considered "unstaged" or "untracked." Staging files with `git add .` prepares these changes to be included in the next commit snapshot of your project.

Here's a breakdown of what `git add .` does:

Staging Changes: It adds all modified (tracked) files and all new files (untracked) to the staging area.

Recursive Operation: The `.` represents the current directory and its subdirectories, so `git add .` recursively adds changes from all directories and subdirectories.

Efficiency: It's a convenient shortcut to stage multiple files and changes quickly without specifying each file individually.

After staging changes with `git add .`, you typically follow up with `git commit` to permanently store those changes in the Git repository history.

OUTPUT:

```
ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ cat README.md
Title: Dr.
Full Name: Naveed
USN: 1WT23CS000
Semester: Third
Section: A
Subject Name: Project Management with GIT
Subject Code: BCS358C
Academic Year: 2024-25
Mobile No: 9620483405
Email ID: naveed.gce@gmail.com

ADMIN@DESKTOP-2DFSJ3U MINGW64 /z/1WT23CS000/aboutMyself (master)
$ git status
On branch master
nothing to commit, working tree clean
```

VIVA-VOCE

1. **What is Git?**
   Git is a distributed version control system used to manage project code and track changes over time.
2. **Who developed Git and why?**
   Linus Torvalds developed Git in 2005 for managing Linux kernel development after the community stopped using BitKeeper.
3. **What are the three states in Git?**
   Modified, Staged, and Committed – representing a file's current workflow stage.
4. **What is a repository in Git?**
   A repository is a project directory tracked by Git, containing all version history and files.
5. **How do you initialize a Git repository?**
   By using the command git init inside the project folder.
6. **What is a staging area in Git?**
   A temporary area where changes are prepared (staged) before committing to the repository.
7. **What is a commit in Git?**
   A commit saves the current staged changes to the Git repository permanently with a message.
8. **What command is used to set user identity in Git?**
   git config --global user.name "Name" and git config --global user.email "email@example.com"
9. **What is the command to view current Git configuration?**
   git config --list
10. **How to clear the Git Bash screen?**
    Use clear or press Ctrl + L.
11. **What does git add . do?**
    It stages all the modified and new files in the current directory for the next commit.
12. **How to create a new file and add content?**
    Use nano filename or touch filename, then edit and save.
13. **What is the purpose of a README.md file?**
    It provides basic documentation and information about the repository.
14. **How to check the status of your working directory?**
    git status shows changes, untracked files, and staged files.
15. **How do you create a new branch?**
    git branch feature-branch creates a new branch named “feature-branch”.
16. **How to switch branches in Git?**
    Use git checkout branch-name.
17. **What is merging in Git?**
    Merging combines changes from one branch into another.
18. **How to merge a feature branch into master?**
    First switch to master using git checkout master, then use git merge feature-branch.
19. **What happens if you modify a file in a feature branch?**
    Changes remain local to that branch until merged into master.
20. **How do you delete a branch?**
    Use git branch -d branch-name.
21. **What are Git tags?**
    Tags mark specific points in Git history, often used for releases.
22. **Difference between lightweight and annotated tags?**
    Lightweight tags are simple pointers; annotated tags store metadata like author and date.
23. **How to create a lightweight tag?**
    git tag v1.0
24. **How to create an annotated tag?**
    git tag -a v2.0 -m "Tag message"

25. **How to list all tags in a repository?**
    Use git tag
26. **Can tags be pushed to remote repositories?**
    Yes, using git push origin tagname
27. **Can you delete a tag?**
    Yes, use git tag -d tagname
28. **How to view details of an annotated tag?**
    Use git show tagname
29. **Are tags part of branches?**
    No, tags are separate and point to commits, not branches.
30. **Why use tags in Git?**
    To mark release versions or important milestones in the codebase.

31. **What is git log used for?**
    To view the commit history with author, date, and message.
32. **How to see commits by a specific author?**
    git log --author="Author Name"
33. **How to view commit logs in brief?**
    git log --oneline
34. **How to view commits between dates?**
    git log --author="Name" --since="YYYY-MM-DD" --until="YYYY-MM-DD"
35. **How to view a specific commit using ID?**
    Use git show commit-ID
36. **What does git diff do?**
    Shows the difference between files or commits.
37. **Can Git log be used to see changes across branches?**
    Yes, using options like git log branch-name.
38. **How to view the last 5 commits?**
    git log -n 5
39. **Can we search commit messages?**
    Yes, with git log --grep="message"
40. **What is git blame used for?**
    To find who last modified a particular line of code.
41. **What is git cherry-pick?**
    It applies a specific commit from another branch into the current branch.
42. **Use case of cherry-pick?**
    Useful when you want only specific changes from a feature branch.
43. **How to undo a commit?**
    Use git revert commit-ID
44. **How to amend a previous commit message?**
    Use git commit --amend
45. **What is git reflog?**
    It records updates to the tip of branches, helping you recover lost commits.
46. **Can git cherry-pick cause conflicts?**
    Yes, conflicts may arise if the same content was modified.
47. **How to resolve cherry-pick conflicts?**
    Edit the file, resolve the conflict, then commit.
48. **How to change a commit message in history?**
    Use git rebase -i and modify messages carefully.
49. **How to check file history using Git?**
    Use git log filename
50. **What is git revert vs git reset?**
    git revert creates a new commit that undoes changes; git reset changes commit history.

51. **What is GitHub?**
    A platform for hosting Git repositories and collaborating on projects online.
52. **How to clone a GitHub repository to VS Code?**
    Use git clone URL and open it in VS Code.
53. **How to push local changes to GitHub?**
    Use git push origin branch-name

54. **What is git remote add origin?**
    It links the local repository to a remote GitHub repo.
55. **How to configure Git in VS Code?**
    Set Git path in settings and enable Git extension.
56. **What is a pull request?**
    A request to merge changes from one branch to another in GitHub.
57. **How to create a GitHub repository?**
    Click on "New" in GitHub, name the repository, and set options.
58. **How to view commit history in GitHub?**
    Navigate to the "Commits" tab in the repository.
59. **What is GitHub Actions?**
    A CI/CD feature to automate workflows in GitHub.
60. **What are forks in GitHub?**
    A personal copy of someone else's repository to contribute.

61. **How to rename a file in Git?**
    git mv oldname newname
62. **How to remove a file from Git?**
    git rm filename
63. **What is .gitignore?**
    A file that specifies which files Git should ignore.
64. **How to list all branches?**
    git branch
65. **What is HEAD in Git?**
    It refers to the current commit your working directory is based on.
66. **What is a detached HEAD state?**
    When HEAD points directly to a commit, not a branch.
67. **Can Git track empty directories?**
    No, Git only tracks files.
68. **How to revert to a previous commit?**
    Use git checkout commit-ID
69. **How to reset staging area?**
    git reset unstages files.
70. **How to discard changes in working directory?**
    git checkout -- filename

71. **Why use branches in projects?**
    To work on features independently without affecting the main codebase.
72. **When should you use tags in a project?**
    During stable releases or version marking.
73. **Why is Git preferred in large projects?**
    Due to its distributed architecture, speed, and branching efficiency.
74. **What is the role of commits in collaboration?**
    They provide checkpoints for tracking changes and debugging.
75. **How does Git help in team development?**
    Enables parallel development, tracking contributions, and resolving conflicts.
76. **How often should you commit changes?**
    Frequently, with meaningful commit messages.
77. **Why commit messages are important?**
    They describe changes clearly for future reference.
78. **How can Git help in backups?**
    Local and remote repositories serve as backups.
79. **Why use Git over traditional file copy methods?**
    Git provides version control, history, and collaboration support.
80. **What is the advantage of using VS Code with Git?**
    Integrated terminal, Git GUI, and ease of code editing.

81. **What is version control?**
    A system for tracking changes in files over time.

82. **Difference between Git and GitHub?**
    Git is the version control system; GitHub is a hosting platform for Git repositories.
83. **What is a commit hash?**
    A unique ID for each commit, used to reference it.
84. **Why is Git distributed?**
    Each user has a complete copy of the repository.
85. **Can you work offline with Git?**
    Yes, most operations are local and can sync later.
86. **Why use git init?**
    To start tracking a new project with Git.
87. **Is Git case-sensitive?**
    Yes, file names in Git are case-sensitive on case-sensitive systems.
88. **What happens when two users edit the same file?**
    Git highlights conflicts during merge.
89. **What is a conflict in Git?**
    When Git cannot automatically resolve differences between changes.
90. **How to resolve conflicts?**
    Manually edit the file and commit resolved version.
91. **What is the use of .md files in GitHub?**
    Markdown files for documentation and project info.
92. **What command is used to edit files in terminal?**
    nano filename
93. **How to stage only specific files?**
    git add filename1 filename2
94. **What is the role of git push?**
    Uploads local commits to the remote repository.
95. **What command sets the default branch name in newer Git versions?**
    git config --global init.defaultBranch main
96. **What is the importance of lab record in Git course?**
    Essential for tracking learning progress, assessed during CIE.

ASSIGNMENT

Q1.(EXP-01)BasicSetupandCreationofaNewRepository Aim:

1. Tocreateanewrepository“1WT23CS000”under anydiscdrivesuch asZ drive andalso a sub repository “aboutMyCollege”.
2. Tomakethe defaultbranch as masterbranchin some systemsit is main branch.
3. Tolaunchthegit andentertheuserconfigurationslikename,emailID.

(EXP-02)Toadd README.mdfileintotheRepository

Aim:

1. ToaddREADME.mdfileintotherepository/z/1WT23CS000/aboutMyCollegeunder master branch.
2. Toaddthe contentsgiven below:

```
Title: GITW
Full Name: Ghousia Institute of Technology for Women
College Code: WT
Affiliation: VTU, Belagavi
Year of Establishment: 2023
No. of Branches: 03
Departments: CS, IS and EC
Mobile No: 080-25536527
Email ID: principalgitw@gmail.com
Address: DRC post, near Dairy Circle Hosur Road Bangalore-560029
```

3. Tocommitwithamessage“Contentsupdatedsuccessfully”

(Exp-03)Creating andManagingBranches

Aim:

1. Tocreateanewbranchnamed "feature-branch”.
2. To add a text file “aboutMyCollege.txt” into the repository /1WT23CS000/aboutMyCollege.
3. Toaddthe contentsinto thetext filegiven below:

```
Title: GITW
Full Name: Ghousia Institute of Technology for Women
College Code: WT
Affiliation: VTU, Belagavi
Year of Establishment: 2023
No. of Branches: 03
Departments: CS, IS and EC
Mobile No: 080-25536527
Email ID: principalgitw@gmail.com
Address: DRC post, near Dairy Circle Hosur Road Bangalore-560029
```

4. Tocommitamessage“AddedtextfileaboutMyCollege.txt successfully”.

(Exp-04)MergingofFeature-BranchintotheMasterBranch: Aim:
Tomergefeature-branchintothemasterbranch
Experiment-05: UpdatingoffilesintheFeature-Branch
Aim:

1. ToaddtheBriefReviewofGITWinthefile“aboutMyCollege.txt”ofthefeature- branch:

GITW was established in the year 2023, affiliated with Visvesvaraya Technological University (VTU), Belagavi, Karnataka. Recognized by AICTE, New Delhi, and the Government of Karnataka, ranking first in women's minority education in the state. The college provides hostel facilities and organizes diverse programs enhancing students' overall personality. It offers B.E Programs in:

- ComputerScience&Engineering
- InformationScience&Engineering
- Electronics&CommunicationEngineering

Tocommit theabovewitha message“AddedbriefreviewofGITW Successfully”
Tomergethefeature-branchwiththemasterbranchandcommitwithamessage“Mergedthe feature-branch Successfully and updated the file with review Details”.

Q.5(Exp-06)LightWeightandAnnotatedTags Aim:

1. To update the file “README.md” and add “Date of Joining to GITW: 1st Oct-2023” to it under the repository /1WT23CS000/aboutMyCollege under master branch.
2. Tocommitwith amessage“My Date ofJoining to GITWUpdatedsuccessfully”
3. Toadd alight wighttag v1.0 into file“README.md”
4. ToaddanAnnotatedtag v2.0withamessage“My DateofJoiningtoGITW”intothe same file

Q.6(Exp-09)GitinVSCode Aim:

1. Toclonetherepository/z/1WT23CS000/aboutMyCollege fromGit-BashtoVS code:
2. ToaddanewfilevsFile.txtunderVScode.Addthefollowingdata: Vs

File details:

- Version:2.1
- DateofInstallation: 30/07/2024
- AuthorName:Dr.Naveed

andcommitamessage “AddedvsFile.txt successfully”

SUMMARY OF CODING

EXPERIMENT-01

```
// To start the git and to make master branch as default branch
$ git init
// To register your name (author name) and Email address for the
   new project
$ git config --global user.name “Dr.NAVEED”
$ git config --global user.email naveed.gce@gmail.com
// To check the author name , email address and other data:
$ git config --list
// To check author name of the project
$ git config user.name
// To check email address of the author
$ git config user.email
// To clear the screen of the terminal (Ctrl+L)
$ clear
// T get into the Z drive of the computer hard disk. It can be
   any symbol
$ cd /z
// To create a repository “1WT23CS000”
$ mkdir 1WT23CS000
// To get back to the repository
$ cd 1WT23CS000
// To create another sub-repository “aboutMyself”
$ mkdir aboutMyself
// To get back to aboutMyself
$ cd aboutMyself
// To get back in single shot
$ cd /z/1WT23CS000/aboutMyself
```

EXPERIMENT-02

```
//To make any drive like z drive as a safe directory
$ git config --global --add safe.directory z:/
// To add a new file like README.md as untracked file
$ echo > README.md
// To track the file like README.md
$ git add .
//To commit with a message like Added README.md file successfully
$ git commit -m “committed message”
// To edit the file like README.md in an open screen
$ nano README.md
// To read the contents of a file like README.md under terminal
$ cat README.md
//To check the current status
$ git status
```

EXPERIMENT-03

```
//To check the current branch as well to check out number

  of branches available
$ git branch
// To create a new branch such as feature-branch
$ git branch feature-branch
// To shift the directory to the new branch such as feature-branch

$ git checkout feature-branch

// To shift back the directory to the default master branch

$ git checkout master

// To checkout number of files available in the repository

$ git ls-files
```

EXPERIMENT-04

```
// To merge feature-branch into master branch which will add all
   files of feature-branch into master branch
$ git merge feature-branch
```

EXPERIMENT-05

```
// To delete a branch such as feature-branch
$ git branch -d feature-branch
// To get reference code of deleted branch
$ git reflog
// To restore the deleted branch such as feature-branch with
   the given ref.code
$ git checkout -b feature-branch ref.code
(in updated version ref.code is not needed)
```

EXPERIMENT-06

```
// To add a light weight tag such as v1.0
$ git tag v1.0
// To check the number of tags available

$ git tag

// To check a particular tag such as v1.0

$ git show v1.0

// To add an annotated tag such as v2.0

$ git tag -a v2.0 -m “committed message”
```

EXPERIMENT-07

```
// To check the committed messages in master branch with full details
$ git log master

// To check the committed messages in master branch in brief with just one line
$ git log master --oneline
```

```
//To check committed message of a particular task with the given reference code ID

$ git show ref.code ID

// To check commits of a particular author from date to desired date with full details

$ git log --author=”authorName” --since=”YEAR-MONTH-DAY” until=”YEAR-MONTH-DAY”

// To check commits of a particular author from date to desired date in brief with just one line

$ git log --author=”authorName” --since=”YEAR-MONTH-DAY” until=”YEAR-MONTH-DAY” --oneline
// To check last 5 commits made in master branch with complete details
$ git log master -n 5
// To check last 5 commits made in master branch in brief with just one line.
$ git log master -n 5 --oneline
Note: Press Q to get back to the coding.
```

EXPERIMENT-08

```
// To check the committed messages in feature- branch
$ git log feature-branch --oneline
// To check committed message of feature-branch with ref.ID in master branch
$ git cherry-pick ref.ID
// To open the conflicting file such as aboutMyself.txt

$ vim aboutMyself.txt

//To delete a file such as data1.txt

$ git rm data1.txt

//To modify the committed message with Ref.ID

$ git revert Ref.ID

//To display the committed message with Ref.ID

$ git show Ref.ID
```

EXPERIMENT-09

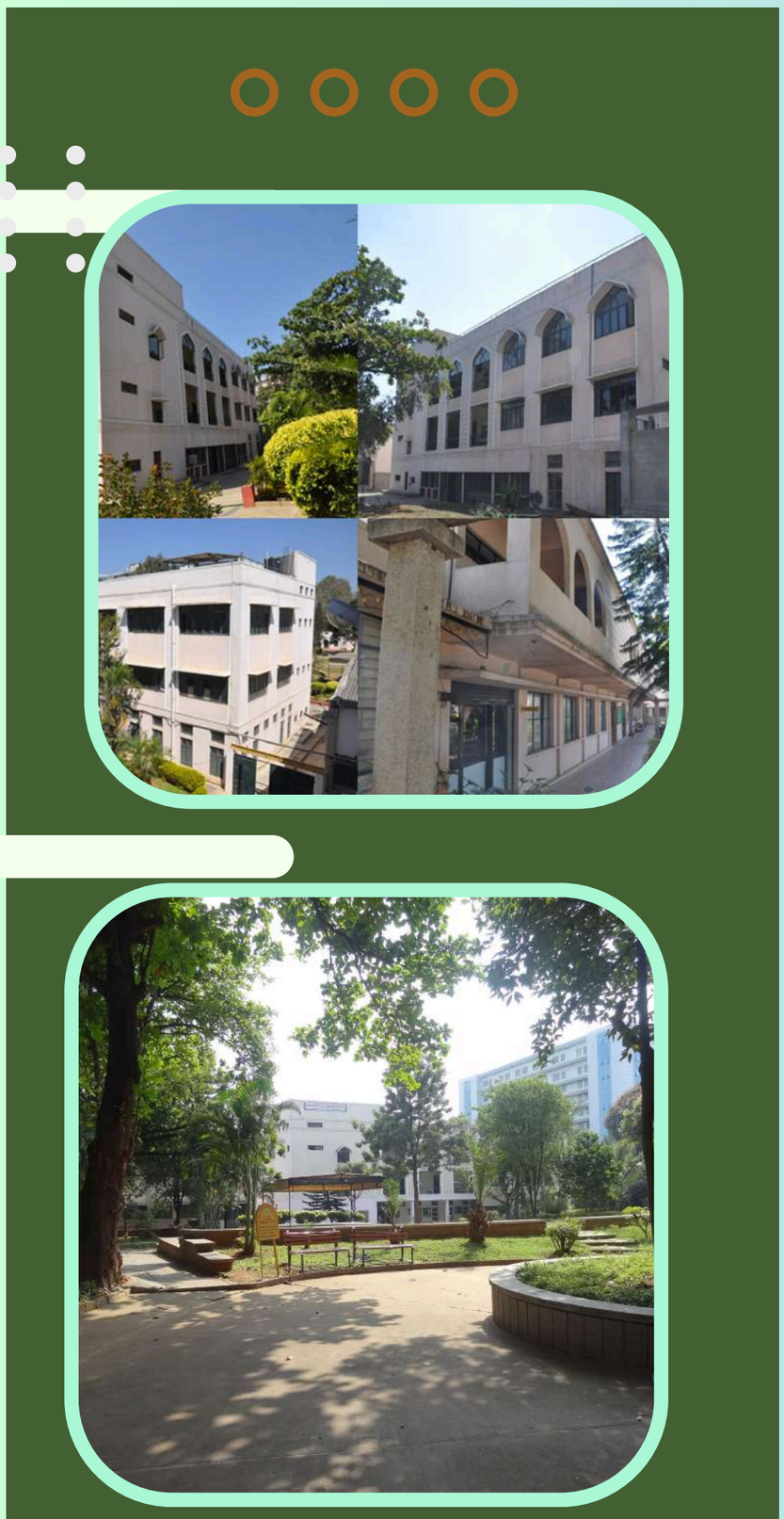
```
// To get back to the current repository in Z drive such as 1WT23CS000/aboutMyself
$ cd /z/1WT23CS000/aboutMyself
// To clone the above repository in VS code after getting back to the current repository
$ code .
```

EXPERIMENT-10

```
// To clone repository in Z drive such as 1WT23CS000/aboutMyself into the GitHub account
$ git remote add origin URLcodeOfGitHubRepository
//To push the repository into the GitHub account if it is master branch
$ git push -u origin master
//To push the repository into the GitHub account if it is main branch
$ git push -u origin main
//To add a new branch such as feature-branch into GitHub account
$ git push -u origin feature-branch
```

GHOUSIA INSTITUTE OF TECHNOLOGY FOR WOMEN

Near Dairy Circle, Hosur Road, Bengaluru-560029, KARNATAKA
Affiliated to VTU., Belagavi, Recognized by Government of Karnataka & A.I.C.T.E., New Delhi



B.E Programs Offered

- Computer Science & Engineering
- Information Science & Engineering
- Electronics & Communication Engineering.

Contact

9986343109 / 9845954481
080 – 25536527

www.gitw.in

It was established in the year 2023, affiliated with Visvesvaraya Technological University (VTU), Belagavi, Karnataka. Recognized by AICTE, New Delhi, and the Government of Karnataka. It is one among the two engineering colleges for women in the state. The college provides hostel facilities and organizes diverse programs enhancing students' overall personality.