

# **ATME COLLEGE OF ENGINEERING**

**13<sup>th</sup> KM Stone, Bannur Road, Mysore - 560 028**



# **A T M E**

**College of Engineering**

## **DEPARTMENT OF COMPUTER SCIENCE AND DESIGN**

**(ACADEMIC YEAR 2026-27)**

## **LABORATORY MANUAL**

**SUBJECT: PROJECT MANAGEMENT WITH GIT**

**SUB CODE: 1BCSL307A**

**SEMESTER: III**

**Outcome Based Education (OBE) and Choice Based Credit System (CBCS)**

**(Effective from the academic year 2023-24)**

**Composed By**

**Faculty Coordinator**

**Approved By**

**Mr. PRADEEP J**  
**Assistant Instructor**

**Mr. SHASHANK PATIL**  
**Assistant Professor**

**Dr. NASREEN FATHIMA**  
**HOD, CSE-CY**

# **INSTITUTIONAL MISSION AND VISION**

## **Objectives**

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

## **Vision**

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

## **Mission**

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

## **DEPARTMENT VISION AND MISSION**

### **Vision**

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

### **Mission**

1. To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.
2. To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

### **Program Specific Outcomes:**

**PSO1:** To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.

**PSO2:** Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

|                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                                         |            |     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|-----|
| 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                                                                                                                                                                                               |            |     |
| <b>Course objectives:</b> <ul style="list-style-type: none"><li>• .To familiar with basic command of Git</li><li>• To create and manage branches</li><li>• To understand how to collaborate and work with Remote Repositories</li><li>• To familiar with virion controlling commands</li></ul> |                                                                                                                                                                                                         |            |     |
| Sl.NO                                                                                                                                                                                                                                                                                          | Experiments                                                                                                                                                                                             |            |     |
| 1                                                                                                                                                                                                                                                                                              | <b>Setting Up and Basic Commands</b><br><br>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                                                                                                                                                                                                                                                                                              | <b>Creating and Managing Branches</b><br>Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."                                           |            |     |
| 3                                                                                                                                                                                                                                                                                              | <b>Creating and Managing Branches</b><br>Write the commands to stash your changes, switch branches, and then apply the stashed changes.                                                                 |            |     |
| 4                                                                                                                                                                                                                                                                                              | <b>Collaboration and Remote Repositories</b><br><br>Clone a remote Git repository to your local machine.                                                                                                |            |     |
| 5                                                                                                                                                                                                                                                                                              | <b>Collaboration and Remote Repositories</b><br><br>Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.                                      |            |     |
| 6                                                                                                                                                                                                                                                                                              | <b>Collaboration and Remote Repositories</b><br><br>Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.                                    |            |     |
| 7                                                                                                                                                                                                                                                                                              | <b>Git Tags and Releases</b><br><br>Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.                                                               |            |     |
| 8                                                                                                                                                                                                                                                                                              | <b>Advanced Git Operations</b>                                                                                                                                                                          |            |     |

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | Write the command to cherry-pick a range of commits from "source-branch" to the current branch.                                                                                        |
| 9                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | <b>Analysing and Changing Git History</b><br><br>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                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | <b>Analysing and Changing Git History</b><br><br>Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."                             |
| 11                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | <b>Analysing and Changing Git History</b><br>Write the command to display the last five commits in the repository's history.                                                           |
| 12                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | <b>Analysing and Changing Git History</b><br><br>Write the command to undo the changes introduced by the commit with the ID "abc123".                                                  |
| <b>Course outcomes (Course Skill Set):</b><br>At the end of the course the student will be able to: <ul style="list-style-type: none"> <li>• Use the basics commands related to git repository</li> <li>• Create and manage the branches</li> <li>• Apply commands related to Collaboration and Remote Repositories</li> <li>• Use the commands related to Git Tags, Releases and advanced git operations</li> <li>• Analyse and change the git history</li> </ul> |                                                                                                                                                                                        |

### 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.

## CONTENTS

| Sl.No. | EXPERIMENT NAME                                                                                                                                                                                     | CO's | Page No |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|---------|
|        | Introduction                                                                                                                                                                                        |      |         |
| 1      | <b>Setting Up and Basic Commands</b><br>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      | <b>Creating and Managing Branches</b><br>Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."                                       |      |         |
| 3      | <b>Creating and Managing Branches</b><br>Write the commands to stash your changes, switch branches, and then apply the stashed changes.                                                             |      |         |
| 4      | <b>Collaboration and Remote Repositories</b><br>Clone a remote Git repository to your local machine.                                                                                                |      |         |
| 5      | <b>Collaboration and Remote Repositories</b><br>Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.                                      |      |         |
| 6      | <b>Collaboration and Remote Repositories</b><br>Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.                                    |      |         |
| 7      | <b>Git Tags and Releases</b><br>Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.                                                               |      |         |
| 8      | <b>Advanced Git Operations</b><br>Write the command to cherry-pick a range of commits from "source-branch" to the current branch.                                                                   |      |         |
| 9      | <b>Analysing and Changing Git History</b><br>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     | <b>Analysing and Changing Git History</b><br>Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."                                              |      |         |
| 11     | <b>Analysing and Changing Git History</b><br>Write the command to display the last five commits in the repository's history.                                                                        |      |         |
| 12     | <b>Analysing and Changing Git History</b><br>Write the command to undo the changes introduced by the commit with the ID "abc123".                                                                   |      |         |

## INTRODUCTION TO GIT

Git is a distributed version control system[9] that tracks changes in any set of computer files, usually used for coordinating work among programmers who are collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different computers

### Characteristics

Git's design is a synthesis of Torvalds's experience with Linux in maintaining a large distributed development project, along with his intimate knowledge of file-system performance gained from the same project and the urgent need to produce a working system in short order. These influences led to the following implementation choices:

- **Strong support for non-linear development**

Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. In Git, a core assumption is that a change will be merged more often than it is written, as it is passed around to various reviewers. In Git, branches are very lightweight: a branch is only a reference to one commit.

- **Distributed development**

Like Darcs, BitKeeper, Mercurial, Bazaar, and Monotone, Git gives each developer a local copy of the full development history, and changes are copied from one such repository to another. These changes are imported as added development branches and can be merged in the same way as a locally developed branch.

- **Compatibility with existing systems and protocols**

Repositories can be published via Hypertext Transfer Protocol Secure (HTTPS), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), or a Git protocol over either a plain socket or Secure Shell (ssh). Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Subversion repositories can be used directly with git-svn.

- **Efficient handling of large projects**

Torvalds has described Git as being very fast and scalable,<sup>[37]</sup> and performance tests done by Mozilla<sup>[38]</sup> showed that it was an order of magnitude faster diffing large repositories than Mercurial and GNU Bazaar; fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server.

- **Cryptographic authentication of history**

The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. The structure is similar to a Merkle tree, but with added data at the nodes and leaves.<sup>[40]</sup> (Mercurial and Monotone also have this property.)

- **Toolkit-based design**

Git was designed as a set of programs written in C and several shell scripts that provide wrappers around those programs.<sup>[41]</sup> Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.

- **Pluggable merge strategies**

As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is needed.

- **Garbage accumulates until collected**

Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository. Garbage collection can be called explicitly using `git gc`.

## Commonly used Linux commands

- **ls**-Display information about files in the current directory
- **pwd**-Displays the current working directory
- **mkdir**- Creates a directory
- **cd**- To navigate between different folders
- **cat**-displays the contents on terminal
- **clear**-Clear the terminal
- **rmdir**-Removes the empty directories from the directory list
- **cp**-Moves files from one directory to other another
- **mv**-Rename and Replace the files
- **rm**-Delete files
- **uname**-Command to get basic information about OS
- **locate**-Find a file in Database
- **touch**-create empty files
- **ln**-Create shortcuts to other files
- **ps**-Display the Processes in terminal
- **man**-Access manual for all Linux commands
- **grep**-Search for a specific string in output
- **echo**-Display active process on the terminal
- **wget**-Download files from net
- **whoami**-Create or update passwords for existing users
- **sort**-sort the file contents
- **cal**-View the calendar in terminal
- **df**-check the details of the file system
- **wc**-check the lines, word count, and characters in a file using different option

## Program 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

- **git --version**
- **git init**
- **git status**
- **git add file names**
- **git commit -m "commit message"**
- **git log**

## PROGRAM 2: CREATING AND MANAGING BRANCHES

### Aim

To create a new Git branch named feature-branch, switch between branches, and merge the feature branch into the master branch.

### Theory

A branch in Git is an independent line of development. Branches allow developers to work on new features or modifications without affecting the main project. A developer can create a separate branch, make changes, test them, and later merge those changes into the main branch. The main branch may be named master or main. In this program, master is used.

### Commands Used

| Command                                  | Purpose                                            |
|------------------------------------------|----------------------------------------------------|
| <code>git branch</code>                  | Lists the available branches.                      |
| <code>git branch feature-branch</code>   | Creates a new branch.                              |
| <code>git checkout feature-branch</code> | Switches to the feature branch.                    |
| <code>git checkout master</code>         | Switches to the master branch.                     |
| <code>git merge feature-branch</code>    | Merges the feature branch into the current branch. |
| <code>git log --oneline</code>           | Displays a compact commit history.                 |

### Procedure

#### Step 1: Check the existing branches

```
git branch
```

#### Step 2: Create a new branch

```
git branch feature-branch
```

#### Step 3: Switch to the feature branch

```
git checkout feature-branch
```

#### Step 4: Create a feature file

```
echo "New feature implementation" > feature.txt
```

#### Step 5: Stage the file

```
git add feature.txt
```

#### Step 6: Commit the changes

```
git commit -m "Added new feature"
```

#### Step 7: Switch to master

```
git checkout master
```

### Step 8: Merge the feature branch

```
git merge feature-branch
```

### Example / Sample Output

```
$ git branch
* master

$ git branch feature-branch
$ git checkout feature-branch
Switched to branch 'feature-branch'

$ git add feature.txt
$ git commit -m "Added new feature"
[feature-branch abc1234] Added new feature
1 file changed, 1 insertion(+)

$ git checkout master
Switched to branch 'master'

$ git merge feature-branch
Updating abc1234..def5678
Fast-forward
 feature.txt | 1 +
```

### Explanation

The feature is first developed independently on feature-branch. After completing the development, the user switches to master and executes git merge feature-branch. This incorporates the changes from the feature branch into the master branch.

### Result

Thus, the feature-branch was successfully created, changes were committed to it, the master branch was accessed, and the feature branch was successfully merged into master.

## **PROGRAM 3: STASHING CHANGES AND SWITCHING BRANCHES**

### **Aim**

To temporarily store uncommitted changes using Git stash, switch between branches, and restore the stored changes.

### **Theory**

During software development, a developer may be working on a file when an urgent task requires switching to another branch. Committing incomplete work may not be desirable. Git provides the stash mechanism to temporarily save uncommitted changes. The git stash command stores modified changes and returns the working directory to a clean state. After completing work on another branch, the developer can return to the original branch and use git stash pop to restore the saved changes.

### **Commands Used**

| Command                  | Purpose                                                           |
|--------------------------|-------------------------------------------------------------------|
| git branch               | Displays available branches.                                      |
| git checkout branch-name | Switches to a branch.                                             |
| git stash                | Temporarily stores uncommitted changes.                           |
| git stash pop            | Applies the most recent stash and removes it from the stash list. |
| git stash list           | Displays all stored stashes.                                      |
| git status               | Displays the current working status.                              |

### **Procedure**

#### **Step 1: Display branches**

```
git branch
```

#### **Step 2: Switch to feature branch**

```
git checkout feature-branch
```

#### **Step 3: Make changes**

```
echo "Temporary development work" >> sample.txt
```

#### **Step 4: Check status**

```
git status
```

#### **Step 5: Stash changes**

```
git stash
```

#### **Step 6: Switch to master**

```
git checkout master
```

#### **Step 7: Return to feature branch**

```
git checkout feature-branch
```

#### **Step 8: Restore changes**

```
git stash pop
```

### **Example / Sample Output**

```
$ git status
modified: sample.txt

$ git stash
Saved working directory and index state WIP on feature-branch

$ git checkout master
Switched to branch 'master'

$ git checkout feature-branch
Switched to branch 'feature-branch'

$ git stash pop
On branch feature-branch
Changes not staged for commit:
  modified: sample.txt
```

### **Explanation**

git stash is useful when work is incomplete and the developer needs to temporarily change branches. Unlike git commit, it does not create a normal project-history commit. The changes can later be recovered using git stash pop. If

multiple stashes are present, git stash list can be used to view them.

### **Result**

Thus, the uncommitted changes were successfully stored using git stash, the branch was changed, and the changes were restored using git stash pop.

## PROGRAM 4: COLLABORATION AND REMOTE REPOSITORIES

### Aim

To clone a remote Git repository to a local machine and synchronize changes between the local and remote repositories.

### Theory

Git supports collaboration through remote repositories. A remote repository is generally hosted on platforms such as GitHub, GitLab, or Bitbucket. Developers can download a complete copy of a remote repository using `git clone`. After cloning, changes from other developers can be obtained using `git pull`, while local commits can be uploaded using `git push`.

### Commands Used

| Command                    | Purpose                                                      |
|----------------------------|--------------------------------------------------------------|
| <code>git clone URL</code> | Copies a remote repository to the local system.              |
| <code>git pull</code>      | Downloads and integrates changes from the remote repository. |
| <code>git push</code>      | Uploads local commits to the remote repository.              |
| <code>git status</code>    | Displays the current repository status.                      |
| <code>git log</code>       | Displays commit history.                                     |

### Procedure

#### Step 1: Clone the remote repository

```
git clone https://github.com/student/demo.git
```

#### Step 2: Enter the repository

```
cd demo
```

#### Step 3: Check status

```
git status
```

#### Step 4: Obtain latest remote changes

```
git pull
```

#### Step 5: Make a modification

```
echo "Updated project information" >> README.md
```

#### Step 6: Stage the modification

```
git add README.md
```

#### Step 7: Commit the change

```
git commit -m "Updated project information"
```

#### Step 8: Upload the commit

```
git push
```

### Example / Sample Output

```
$ git clone https://github.com/student/demo.git
Cloning into 'demo'...
Receiving objects: 100%
```

```
$ cd demo
$ git pull
Already up to date.
```

```
$ git add README.md
$ git commit -m "Updated project information"
[master abc1234] Updated project information
1 file changed, 1 insertion(+)
```

```
$ git push
Enumerating objects...
Writing objects...
```

### **Explanation**

git clone is normally used when obtaining a project for the first time. git pull synchronizes the local repository with changes available on the remote repository. After making and committing local changes, git push transfers those commits to the remote repository.

### **Result**

Thus, a remote Git repository was successfully cloned to the local machine, remote changes were obtained using git pull, and local changes were uploaded using git push.

## PROGRAM 5: FETCHING CHANGES AND REBASING

### Aim

To fetch the latest changes from a remote repository and rebase the local feature branch onto the updated remote master branch.

### Theory

Rebase is a Git operation used to move or replay local commits on top of another branch. It helps maintain a cleaner and more linear project history. The git fetch command downloads information about the remote repository without automatically merging changes into the current branch. After fetching, git rebase can place the local feature-branch commits on top of the latest remote master branch.

### Commands Used

| Command                                  | Purpose                                         |
|------------------------------------------|-------------------------------------------------|
| <code>git clone URL</code>               | Creates a local copy of a remote repository.    |
| <code>git fetch origin</code>            | Downloads the latest remote information.        |
| <code>git checkout feature-branch</code> | Switches to the feature branch.                 |
| <code>git rebase origin/master</code>    | Rebases the feature branch onto updated master. |
| <code>git rebase --continue</code>       | Continues a rebase after resolving conflicts.   |
| <code>git push</code>                    | Uploads changes to the remote repository.       |

### Procedure

#### Step 1: Clone repository

```
git clone https://github.com/student/demo.git
```

#### Step 2: Enter repository

```
cd demo
```

#### Step 3: Switch to feature branch

```
git checkout feature-branch
```

#### Step 4: Fetch remote changes

```
git fetch origin
```

#### Step 5: Rebase feature branch

```
git rebase origin/master
```

### Example / Sample Output

Before rebase:

```
A---B---C  master
          D---E  feature-branch
```

After git rebase origin/master:

```
A---B---C---D'---E'  feature-branch
```

### Explanation

Git takes the commits made on the feature branch and reapplies them on top of the latest origin/master. If a conflict occurs, use git status, resolve the conflict, git add filename, and then git rebase --continue. The rebase can be cancelled using git rebase --abort.

### Result

Thus, the latest changes were fetched from the remote repository and the local feature branch was successfully rebased onto the updated remote master branch.

## PROGRAM 6: MERGING A FEATURE BRANCH WITH A CUSTOM COMMIT MESSAGE

### Aim

To merge feature-branch into master using a custom commit message.

### Theory

Git merge combines the development history of two branches. A feature branch is commonly used to develop a new function without disturbing the main branch. After completing and testing the feature, it can be merged into the master branch. A custom merge message can be provided using the -m option. The --no-ff option can be used to ensure that Git creates an explicit merge commit.

### Commands Used

| Command                          | Purpose                           |
|----------------------------------|-----------------------------------|
| git checkout master              | Switches to master.               |
| git pull                         | Downloads the latest changes.     |
| git merge --no-ff feature-branch | Merges the feature branch.        |
| -m "message"                     | Provides a custom commit message. |
| git push                         | Uploads the merge commit.         |

### Procedure

#### Step 1: Switch to master

```
git checkout master
```

#### Step 2: Update master

```
git pull
```

#### Step 3: Merge feature branch

```
git merge --no-ff feature-branch -m "Merged feature branch into master"
```

#### Step 4: Push changes

```
git push
```

### Example / Sample Output

```
A---B---C master
      |
      D---E feature-branch
```

After merging:

```
A---B---C-----F master
      \           /
      D---E---
```

### Explanation

The --no-ff option preserves the existence of the feature branch in the project history. This makes it easier to identify when a feature was developed and merged. The custom message clearly describes the purpose of the merge.

### Result

Thus, the feature-branch was successfully merged into the master branch using a custom merge commit message.

## PROGRAM 7: CREATING GIT TAGS AND RELEASES

### Aim

To create a lightweight Git tag named v1.0 for a commit in the local repository.

### Theory

A Git tag is a reference used to identify a specific commit. Tags are commonly used to mark important versions or releases of a software project, such as v1.0, v2.0, or v3.1. A lightweight tag is simply a pointer to a particular commit. Unlike an annotated tag, it does not contain additional information such as a tag message or tagger details.

### Commands Used

| Command              | Purpose                                       |
|----------------------|-----------------------------------------------|
| git tag v1.0         | Creates a lightweight tag named v1.0.         |
| git tag              | Lists all available tags.                     |
| git show v1.0        | Displays information about the tagged commit. |
| git push origin v1.0 | Pushes the tag to a remote repository.        |

### Procedure

#### Step 1: Check commit history

```
git log --oneline
```

#### Step 2: Create the tag

```
git tag v1.0
```

#### Step 3: Verify the tag

```
git tag
```

#### Step 4: Display tagged commit details

```
git show v1.0
```

#### Step 5: Push tag to remote

```
git push origin v1.0
```

### Example / Sample Output

```
$ git tag v1.0
```

```
$ git tag
v1.0
```

```
$ git show v1.0
```

```
commit a1b2c3d
Author: Student
Date:   Mon Aug 24 2026
```

```
    Added login functionality
```

### Explanation

Tags are useful for identifying stable versions of software. For example, after completing and testing the first version of a project, a developer can create the tag v1.0. Later, the project may be tagged as v1.1 or v2.0. A lightweight tag can be deleted using `git tag -d v1.0`.

### Result

Thus, a lightweight Git tag named v1.0 was successfully created and verified for the current commit.

## PROGRAM 8: CHERRY-PICKING A RANGE OF COMMITS

### Aim

To cherry-pick a range of commits from source-branch and apply them to the current branch.

### Theory

Cherry-picking is a Git operation used to apply the changes introduced by one or more specific commits to another branch. It is useful when only selected changes are required instead of merging an entire branch. A range of commits can be selected using `git cherry-pick <oldest-commit>^..<newest-commit>`.

### Commands Used

| Command                                 | Purpose                                       |
|-----------------------------------------|-----------------------------------------------|
| <code>git log source-branch</code>      | Displays commits in the source branch.        |
| <code>git checkout target-branch</code> | Switches to the branch receiving the changes. |
| <code>git cherry-pick</code>            | Applies selected commits.                     |
| <code>git cherry-pick --continue</code> | Continues after resolving a conflict.         |
| <code>git cherry-pick --abort</code>    | Cancels the cherry-pick operation.            |

### Procedure

#### Step 1: View source branch history

```
git log source-branch --oneline
```

#### Step 2: Switch to target branch

```
git checkout master
```

#### Step 3: Cherry-pick required range

```
git cherry-pick A1B2C3^..G7H8I9
```

### Example / Sample Output

Suppose source-branch contains:  
A1B2C3 Added search function  
D4E5F6 Fixed login issue  
G7H8I9 Updated interface  
J1K2L3 Added documentation

Command:  
`git cherry-pick A1B2C3^..G7H8I9`

This includes A1B2C3, D4E5F6 and G7H8I9.

### Explanation

The ^ after the first commit means Git should include the first specified commit in the range. If a conflict occurs, use `git status`, resolve the conflict, `git add filename`, and `git cherry-pick --continue`. To cancel the operation, use `git cherry-pick --abort`.

### Result

Thus, the selected range of commits from source-branch was successfully applied to the current branch using Git cherry-pick.

## PROGRAM 9: ANALYSING A SPECIFIC GIT COMMIT

### Aim

To use Git commands to view detailed information about a specific commit, including the commit ID, author, date, commit message, and changes.

### Theory

Git maintains a complete history of changes made to a project. Each commit is identified by a unique commit ID, usually represented as a SHA hash. The `git log` command is used to view commit history, while `git show` provides detailed information about a particular commit.

### Commands Used

| Command                                | Purpose                                                |
|----------------------------------------|--------------------------------------------------------|
| <code>git log</code>                   | Displays commit history.                               |
| <code>git log --oneline</code>         | Displays compact commit history.                       |
| <code>git show commit-id</code>        | Displays detailed information about a specific commit. |
| <code>git show --stat commit-id</code> | Displays a summary of files and changes.               |

### Procedure

#### Step 1: Display commit history

```
git log --oneline
```

#### Step 2: Select required commit

```
abc1234
```

#### Step 3: Display detailed information

```
git show abc1234
```

### Example / Sample Output

```
commit abc1234
Author: John Doe <john@example.com>
Date:   Mon Aug 24 10:20:00 2026

    Updated login module

diff --git a/login.py b/login.py
...
```

### Explanation

The output provides the commit ID, author, date, commit message and diff. The diff shows the actual modifications made to files. The command is useful for investigating the development history and determining who made a change, when it was made, and what was modified.

### Result

Thus, the specified Git commit was successfully analyzed using `git show`, and its author, date, message, and changes were displayed.

## PROGRAM 10: DISPLAYING COMMITS BY AUTHOR AND DATE

### Aim

To display all commits made by the author John Doe between January 1, 2023 and December 31, 2023.

### Theory

Git provides filtering options with git log. Commits can be filtered based on author and date. The --author option filters commits according to the author name. The --since and --until options specify the beginning and ending dates.

### Commands Used

| Command                                                               | Purpose                                       |
|-----------------------------------------------------------------------|-----------------------------------------------|
| git log --author="John Doe" --since="2023-01-01" --until="2023-12-31" | Displays matching commits by author and date. |
| git log --oneline ...                                                 | Displays matching commits in compact format.  |

### Procedure

#### Step 1: Open the Git repository

```
cd project
```

#### Step 2: Execute filtered log command

```
git log --author="John Doe" --since="2023-01-01" --until="2023-12-31"
```

#### Step 3: Use compact output if required

```
git log --oneline --author="John Doe" --since="2023-01-01" --until="2023-12-31"
```

### Example / Sample Output

```
commit a1b2c3d
Author: John Doe <john@example.com>
Date:   Fri Mar 10 10:20:00 2023
```

```
    Updated homepage
```

```
commit d4e5f6g
Author: John Doe <john@example.com>
Date:   Tue Sep 15 14:30:00 2023
```

```
    Fixed login issue
```

### Explanation

--author selects commits created by John Doe. --since specifies the beginning of the period and --until specifies the ending date. --oneline displays each commit in a compact format. This is useful for analysing individual contributions to a software project.

### Result

Thus, all commits made by John Doe between January 1, 2023 and December 31, 2023 were successfully displayed.

## PROGRAM 11: DISPLAYING THE LAST FIVE COMMITS

### Aim

To display the five most recent commits from the Git repository history.

### Theory

Git stores every committed change in its repository history. The git log command is used to examine this history. When a repository contains a large number of commits, the -n option can restrict the number of commits displayed. For five commits, git log -n 5 or git log -5 can be used.

### Commands Used

| Command              | Purpose                                             |
|----------------------|-----------------------------------------------------|
| git log -n 5         | Displays the five latest commits.                   |
| git log -5           | Short form of the above command.                    |
| git log -5 --oneline | Displays the five latest commits in compact format. |

### Procedure

#### Step 1: Open the repository

```
cd GitDemo
```

#### Step 2: Display five latest commits

```
git log -n 5
```

#### Step 3: Display compact output

```
git log -5 --oneline
```

### Example / Sample Output

```
commit a1b2c3d
Author: Student
Date: Mon Aug 24 2026
    Updated README

commit d4e5f6g
Author: Student
Date: Sun Aug 23 2026
    Fixed login issue

commit h7i8j9k
Author: Student
Date: Sat Aug 22 2026
    Added database module

commit 1lm2n3o
Author: Student
Date: Fri Aug 21 2026
    Added user interface

commit p4q5r6s
Author: Student
Date: Thu Aug 20 2026
    Initial project setup
```

### Explanation

The number 5 tells Git to display only the five latest commits. This does not modify or delete the remaining commits; it only limits the amount of history shown. The same approach can be used with another number, such as git log -10.

### Result

Thus, the five most recent commits in the Git repository were successfully displayed using git log -n 5.

## PROGRAM 12: UNDOING CHANGES USING GIT REVERT

### Aim

To undo the changes introduced by a specific commit with the ID abc123 while preserving the existing Git history.

### Theory

Git provides several methods for undoing changes. The git revert command is used to reverse the changes introduced by a particular commit while maintaining that commit in the project history. Unlike deleting or rewriting history, git revert creates a new commit that contains the reverse changes. This is useful when working with shared or remote repositories.

### Commands Used

| Command               | Purpose                                                  |
|-----------------------|----------------------------------------------------------|
| git revert abc123     | Creates a new commit that reverses the specified commit. |
| git status            | Displays the status during a conflict.                   |
| git revert --continue | Continues a revert after resolving conflicts.            |
| git revert --abort    | Cancels an ongoing revert.                               |

### Procedure

#### Step 1: View commit history

```
git log --oneline
```

#### Step 2: Identify target commit

```
abc1234
```

#### Step 3: Revert the commit

```
git revert abc1234
```

#### Step 4: Resolve conflicts if required

```
git status
```

#### Step 5: Continue after resolving

```
git add filename; git revert --continue
```

### Example / Sample Output

Before reverting:

```
A → B → C → D
          ↑
        abc123
```

After git revert abc123:

```
A → B → C → D → E
                ↑
            Revert commit
```

Example output:

```
[master e4f5g6h] Revert "Added incorrect login validation"
1 file changed, 5 deletions(-)
```

### Explanation

The original commit still remains in the history. The new revert commit reverses its changes. If later modifications cause conflicts, git status can be used to identify them, followed by resolving the files, git add filename, and git revert --continue. The operation can be cancelled using git revert --abort. Git revert is safer for shared repositories because it does not rewrite existing history.

### Result

Thus, the changes introduced by commit abc123 were successfully undone using git revert, while the original commit and the overall Git history were preserved.