

Subject 2: Git

Theme 1: Introduction to Git

What is Git?

Git is a distributed version control system (DVCS) created by Linus Torvalds in 2005. Unlike traditional version control systems, Git is designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on a project simultaneously without interfering with each other's work, by keeping track of changes to files and coordinating work among multiple developers.

Key Features of Git:

- **Distributed Architecture:** Every developer has a local copy of the entire project history, not just the current state of the files. This allows for robust and flexible workflows, offline work, and faster operations.
- **Branching and Merging:** Git's branching model is simple and flexible, allowing developers to create, use, and merge branches for feature development, bug fixes, and experimentation without affecting the main codebase.
- **Data Integrity:** Every file and commit is checksummed using SHA-1, ensuring that the repository's data integrity is maintained.
- **Efficiency:** Git is fast and efficient in handling large projects. It optimizes storage and retrieval of data, making operations like branching, merging, and committing very fast.

Basic Concepts:

- **Repository:** A directory that contains all your project files and the entire revision history. You can create a repository from scratch (`git init`) or clone an existing one (`git clone`).
- **Commit:** A snapshot of your project at a given point in time. Each commit is identified by a unique SHA-1 hash and includes a message describing the changes.
- **Branch:** A pointer to a commit. Branches allow you to diverge from the main line of development and continue to do work without messing with the main codebase.
- **Remote:** A version of your project that is hosted on the internet or a network. Examples include repositories on GitHub, GitLab, or Bitbucket. You can push and pull changes to and from remotes.
- **Staging Area (Index):** A place where you can group changes before committing them. It's like a clipboard for changes that you want to include in the next commit.
- **Working Directory:** The files and directories that you are currently working on. These are your actual project files.

Why Use Git?

- **Collaboration:** Git enables multiple developers to work on a project simultaneously, providing powerful tools for merging changes and resolving conflicts.
- **Version Control:** Git keeps a history of changes, allowing you to revert to previous states, compare changes, and understand the evolution of your project.
- **Branching:** Git's lightweight branching allows you to work on multiple features or bug fixes concurrently without affecting the main codebase.
- **Speed:** Git is designed to be fast, even for large projects, making operations like branching, merging, and committing very efficient.
- **Open Source:** Git is free and open-source software, which means it is continuously improved by a large community of contributors.

Real-World Use Cases:

- **Software Development:** Git is used by individual developers and large teams to manage source code, track changes, and collaborate on software projects.
- **Documentation:** Writers and technical documentation teams use Git to manage changes to documentation, track revisions, and collaborate with other writers.
- **Configuration Management:** System administrators use Git to manage configuration files, track changes, and ensure consistency across systems.

Useful link: <https://git-scm.com/about>

Assignment:

- Write a brief summary (200-300 words) on the history of Git and its core principles.
- Create a personal repository on GitHub, GitLab, or Bitbucket and document the steps taken.

Tests.

Test 1.

1. What is Git?

- a) A programming language
- b) A version control system
- c) A text editor
- d) An operating system

2. Who created Git?
 - a) Bill Gates
 - b) Mark Zuckerberg
 - c) Linus Torvalds
 - d) Steve Jobs
3. Which of the following is a core principle of Git?
 - a) Centralized workflow
 - b) Decentralized development
 - c) Manual version tracking
 - d) Single user access
4. What year was Git initially released?
 - a) 1999
 - b) 2005
 - c) 2010
 - d) 2015
5. Which of the following best describes a repository in Git?
 - a) A central server for code storage
 - b) A collection of documents
 - c) A database for storing code changes
 - d) A project management tool

Test 2.

1. Why is version control important in software development?
2. Explain the main difference between Git and other version control systems like SVN.
3. Describe the purpose of a commit in Git.
4. What is a branch in Git and why is it useful?

5. Explain the concept of a merge in Git.

Practical Test

1. **Create a Git Repository and Make a Commit**
 - **Task:** Initialize a new Git repository in a local directory. Create a new file, add some content, and commit the changes with an appropriate message.
2. **Write a Summary on Git's History and Core Principles**
 - **Task:** Write a 200-300 word summary on the history of Git, why it was created, and its core principles. Include key points such as who created Git, the problem it was solving, and how it revolutionized version control.

Theme 2: Installing Git

Git needs to be installed on your local machine to begin using it. The installation process varies depending on your operating system.

Installation on Different Operating Systems

Linux

1. **Debian-based distributions (Ubuntu):**

```
sudo apt-get update
```

```
sudo apt-get install git
```

2. **RPM-based distributions (Fedora, CentOS):**

```
sudo dnf install git
```

3. **Arch-based distributions:**

```
sudo pacman -S git
```

macOS

There are several ways to install Git on macOS:

1. **Using Homebrew:** Homebrew is a popular package manager for macOS. If you don't have Homebrew installed, you can install it by running:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once Homebrew is installed, you can install Git:

```
brew install git
```

2. **Using the macOS Installer:** Download the latest installer from the official Git website: <https://git-scm.com/download/mac> Follow the instructions to complete the installation.

Windows

There are a few ways to install Git on Windows:

1. **Using Git for Windows:** Download the installer from <https://git-scm.com/download/win> and follow the installation instructions.
2. **Using Chocolatey:** Chocolatey is a package manager for Windows. If you don't have Chocolatey installed, you can install it by running:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex
((New-Object
System.Net.WebClient).DownloadString('https://community.chocolate
y.org/install.ps1'))
```

Once Chocolatey is installed, you can install Git:

```
choco install git
```

Verifying the Installation

After installing Git, you should verify the installation to ensure it's working correctly. Open a terminal (or Command Prompt on Windows) and run:

```
git --version
```

You should see output similar to:

```
git version 2.x.x
```

Configuring Git

Once Git is installed, you need to configure it with your personal information. This is important because Git uses this information to label your commits with the correct author information.

1. Setting your name:

```
git config --global user.name "Your Name"
```

2. Setting your email:

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

You can also set other configurations, such as your preferred text editor:

```
git config --global core.editor "nano" # or vim, code, etc.
```

To see all the configurations, run:

```
git config --list
```

Installing Git is the first step in leveraging its powerful version control capabilities. Whether you are on Linux, macOS, or Windows, the installation process is straightforward. After installation, configuring Git with your personal information ensures that all your work is correctly attributed to you. With Git installed and configured, you're ready to start tracking changes, collaborating with others, and managing your projects efficiently.

Useful links: <https://git-scm.com/downloads>

Assignment:

- Install Git on your local machine and configure your username and email.
- Provide a step-by-step installation guide for your operating system (Windows, macOS, or Linux).

Tests.

Test 1.

1. Which command is used to check if Git is installed on your system?

- a) `git --check`
- b) `git --version`
- c) `git --install`
- d) `git --status`

2. **What is the command to install Git on a Debian-based Linux system?**

- a) `sudo apt-get install git`
- b) `sudo yum install git`
- c) `brew install git`
- d) `choco install git`

3. **Which file is typically used to configure Git with your username and email?**

- a) `.bashrc`
- b) `.gitignore`
- c) `.gitconfig`
- d) `.profile`

4. **Which command sets your username in Git?**

- a) `git config --global user.name "Your Name"`
- b) `git set user.name "Your Name"`
- c) `git init user.name "Your Name"`
- d) `git commit user.name "Your Name"`

5. **How do you verify the Git configuration settings?**

- a) `git verify`
- b) `git config --list`
- c) `git settings --list`

d) `git list-config`

Test 2.

1. Describe the steps to install Git on a Windows system.
2. Explain how to configure your Git username and email, and why this configuration is important.
3. What command would you use to change your Git editor to VS Code?
4. How would you upgrade Git to the latest version on a macOS system using Homebrew?
5. What is the purpose of the `.gitconfig` file and where is it typically located?

Practical Test

1. **Install Git on Your Local Machine**
 - **Task:** Install Git on your operating system (Windows, macOS, or Linux) and configure your global username and email.
2. **Document the Installation Process**
 - **Task:** Write a step-by-step guide for installing Git on your specific operating system, including how to configure your username and email.

Theme 3: Basic Git Commands

Git provides a range of commands to help you manage your version-controlled projects. Here are the most fundamental commands you'll need to get started with Git:

Initializing a Repository

- **git init:** Initializes a new Git repository in the current directory. This command creates a `.git` directory that tracks all changes to files in the project.

```
git init
```

Cloning a Repository

- **git clone:** Creates a copy of an existing repository from a remote URL to your local machine. This is often used to get a copy of a project hosted on platforms like GitHub, GitLab, or Bitbucket.


```
git clone [url]
```

Staging Changes

- **git add**: Adds changes in the working directory to the staging area. This command prepares changes to be committed. You can add specific files or directories, or use `.` to add all changes.

```
git add [file]  
git add .
```

Committing Changes

- **git commit**: Records changes to the repository. Each commit is a snapshot of the repository at a specific point in time. Always include a descriptive commit message with the `-m` flag.

```
git commit -m "Your descriptive message"
```

Checking the Status

- **git status**: Displays the state of the working directory and the staging area. It shows which changes have been staged, which haven't, and which files aren't being tracked by Git.

```
git status
```

Viewing Commit History

- **git log**: Shows the commit history for the repository. This command lists commits in reverse chronological order, with the most recent commits at the top.

```
git log
```

To view a simplified log:

```
git log --oneline
```

Undoing Changes

- **git checkout**: Restores files in the working directory to match the version in the index or a specific commit. This command can also be used to switch branches.

```
git checkout [file]
git checkout [commit]
```

- **git reset**: Unstages changes that have been added to the staging area, or can reset the current branch to a specific commit. This command can be destructive, so use it with caution.

```
git reset [file]
git reset --hard [commit]
```

- **git revert**: Creates a new commit that undoes changes made by a previous commit, without rewriting commit history.

```
git revert [commit]
```

Removing Files

- **git rm**: Removes files from the working directory and the staging area. Use this command to delete files from the project.

```
git rm [file]
```

Comparing Changes

- **git diff**: Shows differences between the working directory and the index or between two commits. This command helps you see what changes have been made.

```
git diff
git diff [commit1] [commit2]
```

Tagging

- **git tag**: Creates a tag, which is a reference to a specific commit. Tags are often used to mark release points, like **v1.0** or **v2.0**.

```
git tag [tag-name]
```

To push tags to a remote repository:

```
git push origin [tag-name]
```

Understanding and using these basic Git commands will enable you to start tracking changes, committing updates, and managing your projects effectively. As you become more comfortable with these commands, you can explore more advanced features and workflows in Git.

Useful links: <https://education.github.com/git-cheat-sheet-education.pdf>

Assignment:

- Initialize a new Git repository in a local directory.
- Add and commit a new file to the repository.
- Make some changes to the file and commit the changes.
- Create a markdown file documenting the commands used and their purposes.

Tests:

Test 1.

1. What is the command to initialize a new Git repository?

- a) `git start`
- b) `git init`
- c) `git new`
- d) `git create`

2. Which command adds a file to the staging area in Git?

- a) `git add`
- b) `git stage`
- c) `git commit`
- d) `git push`

3. What is the command to view the current status of your Git repository?

- a) `git log`
- b) `git show`

c) `git status`

d) `git diff`

4. **How do you commit changes with a message in Git?**

a) `git commit`

b) `git commit -m "message"`

c) `git add commit "message"`

d) `git save -m "message"`

5. **Which command shows the commit history of a repository?**

a) `git history`

b) `git log`

c) `git show`

d) `git commits`

Test 2.

1. Explain the purpose of the `git add` command.
2. Describe the difference between `git commit` and `git commit -m "message"`.
3. What information does the `git status` command provide?
4. How would you revert the last commit without deleting the changes made in it?
5. Explain the difference between `git clone` and `git pull`.

Practical Test

1. **Initialize a Git Repository and Make Initial Commit**

- **Task:** Initialize a new Git repository in a local directory. Create a new file, add some content, stage the file, and commit the changes with a message.

2. **Check Repository Status and View Commit History**

- **Task:** Make some changes to the file created in the previous task. Stage and commit the changes. View the status of the repository and the commit history.

3. **Undoing Changes**

- **Task:** Demonstrate how to revert a change. Make a change to the file, commit it, then revert that commit.

Theme 4: Branching and Merging

Branching and merging are two of the most powerful features in Git. They allow you to work on multiple versions of a project simultaneously, experiment with new features, and manage different lines of development efficiently.

Branching

A branch in Git is a lightweight movable pointer to a commit. When you create a new branch, you're creating a new pointer that you can move around independently of the main line of development (typically called `main` or `master`).

Creating a Branch:

- `git branch [branch-name]`: Creates a new branch with the specified name.

```
git branch feature-branch
```

Switching Branches:

- `git checkout [branch-name]`: Switches to the specified branch.

```
git checkout feature-branch
```

- Alternatively, you can create and switch to a new branch in one step:

```
git checkout -b feature-branch
```

Listing Branches:

- `git branch`: Lists all the branches in the repository. The current branch is highlighted with an asterisk (*).

```
git branch
```

Deleting a Branch:

- `git branch -d [branch-name]`: Deletes the specified branch. This command will only delete the branch if it has been fully merged with its upstream branch.

```
git branch -d feature-branch
```

- **git branch -D [branch-name]**: Forces deletion of the branch, even if it hasn't been merged.

```
git branch -D feature-branch
```

Merging

Merging is the process of integrating changes from one branch into another. The most common use case is to merge changes from a feature branch back into the main branch after the feature is complete.

Merging Branches:

- **git merge [branch-name]**: Merges the specified branch into the current branch.

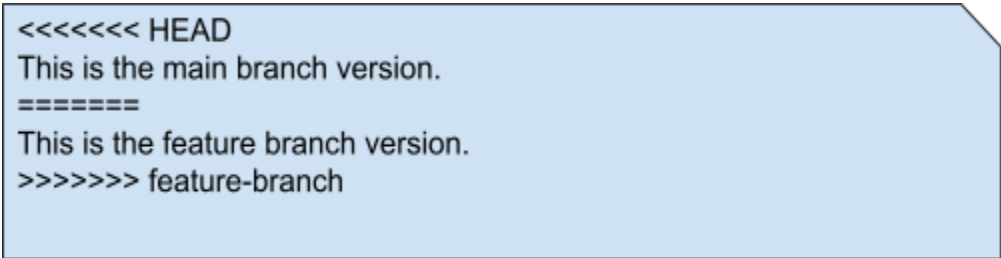
```
git checkout main
```

```
git merge feature-branch
```

When you perform a merge, Git will try to automatically combine the changes. If there are conflicts (i.e., changes that affect the same lines of code differently), Git will mark the conflicts in the files and you will need to resolve them manually.

Handling Merge Conflicts:

1. Open the conflicted file in your text editor. Git will mark the conflict with <<<<<<, =====, and >>>>>> lines.



```
<<<<<< HEAD
This is the main branch version.
=====
This is the feature branch version.
>>>>>> feature-branch
```

2. Edit the file to resolve the conflict, keeping the desired changes.

3. Once you've resolved the conflict, stage the changes and commit them.

```
git add [conflicted-file]
```

```
git commit -m "Resolve merge conflict in [conflicted-file]"
```

Rebasing

Rebasing is an alternative to merging that allows you to integrate changes from one branch into another by applying your changes on top of the other branch's commits.

Rebasing a Branch:

- **git rebase [branch-name]**: Reapplies commits from the current branch on top of the specified branch.

```
git checkout feature-branch
```

```
git rebase main
```

Rebasing can result in a cleaner project history but can also be more complex to manage, especially if conflicts arise. It is generally recommended to use rebasing for local changes that haven't been shared with others

Useful links: <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

<https://git-scm.com/book/en/v2/Git-Branching-Rebasing>

Assignment:

- Create a new branch in your repository and make some changes in it.
- Merge the branch back into the **main** branch.
- Resolve any merge conflicts that arise and document the process.

Tests:

Test 1.

1. Which command is used to create a new branch in Git?

a) **git branch new-branch**

- b) `git create new-branch`
- c) `git checkout new-branch`
- d) `git switch new-branch`

2. How do you switch to an existing branch in Git?

- a) `git change branch`
- b) `git checkout branch-name`
- c) `git branch branch-name`
- d) `git switch branch-name`

3. What is the command to list all branches in a Git repository?

- a) `git list branches`
- b) `git show branches`
- c) `git branch`
- d) `git branches`

4. Which command is used to merge a branch into the current branch?

- a) `git integrate branch-name`
- b) `git combine branch-name`
- c) `git merge branch-name`
- d) `git join branch-name`

5. What does a fast-forward merge mean in Git?

- a) The current branch is reset to match the target branch
- b) The branch is merged without creating a merge commit
- c) The merge is performed automatically without conflicts
- d) The branch history is preserved and a new commit is created

Test 2.

1. Explain the difference between a branch and a tag in Git.
2. Describe the steps to resolve a merge conflict in Git.
3. What is the purpose of using branches in a Git workflow?
4. How would you delete a branch locally and remotely in Git?
5. Explain the difference between a fast-forward merge and a three-way merge.

Practical Test

1. **Create and Switch Branches**
 - **Task:** Create a new branch, switch to it, and make some changes.
2. **Merge Branches and Resolve Conflicts**
 - **Task:** Merge a feature branch into the main branch and resolve any conflicts.
3. **Delete a Branch Locally and Remotely**
 - **Task:** Delete the feature branch both locally and remotely after merging.

Theme 5: Remote Repositories

Remote repositories are versions of your project hosted on the internet or a network. They enable collaboration with other developers by allowing them to access, contribute to, and sync their changes with a central repository. Platforms like GitHub, GitLab, and Bitbucket provide hosting for remote repositories.

Key Concepts

- **Remote:** A remote repository is a common repository that all team members use to exchange their changes. It's usually hosted on a server or a cloud service.
- **Origin:** The default name given to the main remote repository when you clone a repository.
- **Upstream:** The repository you originally forked from, which is often another remote repository.

Common Commands

Adding a Remote Repository:

- **git remote add [name] [url]**: Adds a new remote repository with the specified name and URL.

```
git remote add origin  
https://github.com/yourusername/your-repo.git
```

Listing Remote Repositories:

- **git remote -v**: Lists the remote repositories associated with the local repository, showing the URLs for fetch and push operations.

```
Git remote -v
```

Fetching Changes from a Remote:

- **git fetch [remote]**: Fetches changes from the remote repository but does not merge them into the local repository. This command updates your remote tracking branches.

```
git fetch origin
```

Pulling Changes from a Remote:

- **git pull [remote] [branch]**: Fetches changes from the remote repository and merges them into the current branch. This is a combination of **git fetch** and **git merge**.

```
git pull origin main
```

Pushing Changes to a Remote:

- **git push [remote] [branch]**: Pushes your committed changes to the specified branch on the remote repository.

```
git push origin main
```

Renaming a Remote:

- **git remote rename [old-name] [new-name]**: Renames an existing remote repository.

```
git remote rename origin upstream
```

Removing a Remote:

- **git remote remove [name]**: Removes the specified remote repository.

```
git remote remove origin
```

Cloning a Repository

Cloning is the process of making a copy of a remote repository on your local machine. This includes the entire history of the repository.

- **git clone [url]**: Clones the repository from the specified URL to your local machine.

```
git clone https://github.com/yourusername/your-repo.git
```

This command creates a directory with the same name as the repository, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

Working with Forks

Forking is a way to create a personal copy of someone else's repository. Forks are often used to propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

1. **Fork the repository on GitHub/GitLab/Bitbucket.**
2. **Clone your forked repository to your local machine:**

```
git clone https://github.com/yourusername/forked-repo.git
```

3. **Add the original repository as an upstream remote:**

```
git remote add upstream  
https://github.com/originalowner/original-repo.git
```

4. **Fetch changes from the upstream repository:**

```
git fetch upstream
```

5. **Merge changes from the upstream repository into your local branch:**

```
git merge upstream/main
```

Syncing with a Remote

Keeping your local repository in sync with the remote repository is essential for collaboration. Use `git fetch`, `git pull`, and `git push` commands to update your local repository with changes from the remote and share your changes with others.

Remote repositories are vital for collaboration in modern software development. They allow multiple developers to work on the same project, share their changes, and keep their work synchronized. By mastering commands like `git remote`, `git clone`, `git fetch`, `git pull`, and `git push`, you can effectively manage and collaborate on projects using Git.

Useful links: <https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches>

Assignment:

- Clone an existing repository from GitHub, GitLab, or Bitbucket.
- Create a new branch, make changes, and push the branch to the remote repository.
- Open a pull request for your changes and document the process.

Tests:

Test 1.

1. Which command is used to clone a remote repository?
 - a) `git download`
 - b) `git clone`
 - c) `git fetch`
 - d) `git pull`
2. How do you add a remote repository to an existing local repository?
 - a) `git add remote origin URL`
 - b) `git remote add origin URL`
 - c) `git connect origin URL`
 - d) `git link origin URL`
3. What is the command to push changes to a remote repository?

- a) `git upload`
 - b) `git send`
 - c) `git push`
 - d) `git commit`
4. Which command fetches changes from a remote repository without merging them?
- a) `git fetch`
 - b) `git pull`
 - c) `git merge`
 - d) `git update`
5. How do you set the default remote repository for push and pull operations?
- a) `git set origin`
 - b) `git set remote`
 - c) `git remote default`
 - d) `git push -u origin main`

Test 2.

1. Explain the difference between `git fetch` and `git pull`.
2. Describe the steps to rename a remote repository.
3. What is the purpose of the `git remote -v` command?
4. How would you handle a situation where you need to update your local branch with changes from a remote repository?
5. Explain the process of resolving conflicts that arise after pulling changes from a remote repository.

Practical Test

1. **Clone a Remote Repository**
 - **Task:** Clone an existing remote repository to your local machine.

2. Add a Remote Repository and Push Changes

- **Task:** Add a remote repository to an existing local repository and push changes.

3. Fetch and Merge Changes from a Remote Repository

- **Task:** Fetch changes from a remote repository and merge them into your local branch.

4. Resolve Conflicts After Pulling Changes

- **Task:** Pull changes from a remote repository, resolve any conflicts, and commit the resolved changes.

Theme 6: Git Workflows

Git workflows provide structured methods for using Git in projects, managing source code, features, releases, and bug fixes efficiently. Here are the common Git workflows:

1. Centralized Workflow

All developers commit directly to the `main` branch, similar to SVN.

Steps:

1. Clone the repository.
2. Create a feature branch.
3. Commit changes to the branch.
4. Push and merge into `main`.

2. Feature Branch Workflow

Create a new branch for every feature or bug fix.

Steps:

1. Create a new branch for each feature.
2. Commit changes to the branch.
3. Merge the feature branch into `main`.
4. Delete the feature branch.

3. Gitflow Workflow

A structured model with branches for features, releases, and hotfixes.

Main Branches:

- **main**: Official release history.
- **develop**: Integration branch for features.

Supporting Branches:

- **feature, release, hotfix**.

4. Forking Workflow

Used in open-source projects, each developer has their fork and works independently.

Steps:

1. Fork the repository.
2. Clone your fork.
3. Create a branch for each feature.
4. Push to your fork and create a pull request.

Choosing the right Git workflow depends on the project's complexity and team size. Centralized and Feature Branch Workflows are good for small teams, Gitflow for larger projects with regular releases, and Forking Workflow for open-source projects.

Useful links: <https://www.atlassian.com/git/tutorials/comparing-workflows>

Assignment:

- Implement the Feature Branch Workflow for a small project:
 - Create a new feature branch.
 - Make some changes and commit them.
 - Merge the feature branch back into the **main** branch.
- Document the workflow steps with commands used.

Tests:

Test 1.

1. **Which Git workflow involves having a single main branch where all changes are merged after thorough review and testing?**
 - a) Gitflow Workflow

- b) Centralized Workflow
 - c) Feature Branch Workflow
 - d) Forking Workflow
2. **In Gitflow Workflow, which branch is used for ongoing development and integrating new features?**
- a) `main`
 - b) `feature`
 - c) `release`
 - d) `develop`
3. **Which Git workflow is most commonly used in open-source projects where contributors work on their own copies of the repository?**
- a) Gitflow Workflow
 - b) Centralized Workflow
 - c) Feature Branch Workflow
 - d) Forking Workflow
4. **In the Feature Branch Workflow, what is the primary purpose of a feature branch?**
- a) To maintain a history of all changes
 - b) To work on new features in isolation from the main branch
 - c) To create backup copies of the repository
 - d) To store deployment scripts
5. **Which command is used to create a new branch and switch to it in one step?**
- a) `git branch new-branch`
 - b) `git checkout new-branch`
 - c) `git checkout -b new-branch`
 - d) `git switch new-branch`

Test 2.

1. Explain the main differences between the Gitflow Workflow and the Feature Branch Workflow.
2. Describe the steps involved in the Forking Workflow from forking a repository to submitting a pull request.
3. What are the benefits of using a Feature Branch Workflow in a collaborative development environment?
4. How does the Centralized Workflow differ from other Git workflows, and in what scenarios is it most effective?
5. What is the purpose of the **release** branch in the Gitflow Workflow, and how is it used?

Practical Test

1. **Implement the Feature Branch Workflow**
 - **Task:** Create a new feature branch, make changes, and merge it back into the main branch.
2. **Implement the Gitflow Workflow**
 - **Task:** Follow the Gitflow Workflow to create a feature branch, a release branch, and merge changes into the develop and main branches.
3. **Implement the Forking Workflow**
 - **Task:** Fork a repository, clone it locally, make changes, push them, and create a pull request.

Theme 7: Advanced Git Features

Advanced Git features provide more powerful tools for managing your repositories, history, and collaboration workflows. Here are some key advanced features:

1. Git Stash

Stashing allows you to temporarily save changes that are not ready to be committed.

- **Stash changes:**

git stash

- **Apply stashed changes:**

```
git stash apply
```

- **List stashes:**

```
git stash list
```

- **Apply and remove a stash:**

```
git stash pop
```

2. Interactive Rebase

Interactive rebasing allows you to rewrite commit history by editing, combining, or reordering commits.

- **Start an interactive rebase:**

```
git rebase -i HEAD~[number of commits]
```

- **Commands in the rebase editor:**

1. **pick**: Use the commit.
2. **reword**: Edit the commit message.
3. **edit**: Amend the commit.
4. **squash**: Combine commits.
5. **drop**: Remove the commit.

3. Cherry-Pick

Cherry-picking allows you to apply a commit from one branch to another.

- **Cherry-pick a commit:**

```
git cherry-pick [commit-hash]
```

4. Git Hooks

Git hooks are scripts that run automatically at certain points in the Git workflow, such as before a commit or after a merge.

- **Common hooks:**

- `pre-commit`: Runs before a commit is made.
- `pre-push`: Runs before pushing to a remote repository.
- `post-merge`: Runs after a successful merge.
- **Creating a hook:** Hooks are shell scripts stored in the `.git/hooks` directory. For example, to create a `pre-commit` hook:

```
touch .git/hooks/pre-commit  
  
chmod +x .git/hooks/pre-commit  
  
# Add your script to the file
```

5. Submodules

Submodules allow you to keep a Git repository as a subdirectory of another Git repository.

- **Add a submodule:**

```
git submodule add [repository-url] [path]
```

- **Update submodules:**

```
git submodule update --init --recursive
```

6. Git Bisect

Git bisect helps you find the commit that introduced a bug by performing a binary search.

- **Start bisect:**

```
git bisect start
```

- **Mark current commit as bad:**

```
git bisect bad
```

- **Mark a known good commit:**

```
git bisect good [commit-hash]
```

- Git will checkout commits for you to test and mark as good or bad until the bad commit is found.

7. Reflog

Reflog allows you to view the history of changes to the tips of branches and other references.

- Show reflog:

```
git reflog
```

- Recover a commit:

```
git checkout [reflog-hash]
```

Advanced Git features like stashing, rebasing, cherry-picking, hooks, submodules, bisect, and reflog provide powerful tools for managing complex workflows, debugging, and customizing your Git experience. Mastering these features can significantly enhance your productivity and collaboration in software development.

Useful links: <https://www.atlassian.com/git/tutorials/advanced-overview>

Assignment:

- Use Git stash to save your changes temporarily and apply them later.
- Perform an interactive rebase to combine multiple commits into one.
- Create and apply a Git hook for pre-commit or pre-push actions.
- Document the commands and steps used for each advanced feature.

Tests:

Test 1.

1. What command is used to interactively stage changes for the next commit?

a) `git commit`

b) `git add`

c) `git stash`

d) `git add -p`

2. Which command allows you to rewrite history in a Git repository by modifying previous commits?
- a) `git merge`
 - b) `git rebase`
 - c) `git reset`
 - d) `git checkout`
3. How can you temporarily save changes in your working directory without committing them?
- a) `git store`
 - b) `git hold`
 - c) `git stash`
 - d) `git shelve`
4. What is the purpose of the `git cherry-pick` command?
- a) To revert a specific commit
 - b) To apply changes from a specific commit to another branch
 - c) To delete a commit from history
 - d) To create a new branch from a specific commit
5. Which command is used to create a lightweight, movable reference to a commit?
- a) `git branch`
 - b) `git tag`
 - c) `git alias`
 - d) `git reference`

Short Answer Questions

1. Explain the difference between `git rebase` and `git merge`.
2. Describe a scenario where `git stash` would be particularly useful.

3. What are the benefits of using `git bisect` to find a bug?
4. How does `git cherry-pick` differ from `git merge`?
5. What are annotated tags in Git, and how do they differ from lightweight tags?

Practical Test

1. Rebase a Branch

- **Task:** Rebase a feature branch onto the main branch.

2. Use Git Stash

- **Task:** Stash changes in your working directory, switch branches, and then apply the stashed changes.

3. Cherry-Pick a Commit

- **Task:** Apply a specific commit from one branch to another using `git cherry-pick`.

4. Create and Annotate a Tag

- **Task:** Create an annotated tag for a specific commit.

5. Find a Bug with Git Bisect

- **Task:** Use `git bisect` to find a commit that introduced a bug.