Chapter 1: Git Fundamentals

Git

Git is a distributed version control system used to track changes in source code(files) during software development. This allows multiple people to make changes to the same file simultaneously, while keeping a complete history of who exactly made which change. This allows rollback or restore in case of unwanted changes

How to set up git?

```
git config --global user.name "git-username"

git config --global user.email "git-email"

# generate ssh key
ssh-keygen

# setup SSH key in your GitHub account

# test authentication
ssh -T git@github.com
```

How to initialize a git repository?

```
git init

# Rename default branch if needed to be changed
git branch -m <branch-name>

# make sure the remote repository already exists in your GitHub account
git remote add origin <remote-repository>
```

Basic git commands

```
git clone <remote-repository>

git add .
# or
git add <file-name>

git commit -m "<commit message>"
# or
git commit -m "<type>(<scope>): <short description>" -m "<commit details>"
-m "<footer note>"

<<comment
common types are:
feat: A new feature
fix: A bug fix
docs: Documentation changes
style: Non-functional code changes (e.g., formatting)
refactor: Code restructuring without changing functionality
test: Adding or modifying tests
chore: Miscellaneous tasks (e.g., build scripts)
comment

git push origin <branch-name>

git pull origin <branch-name>
```

Why specify origin <branch-name> in git pull or git push?

If the repository has multiple remotes (e.g., origin, upstream, fork), specifying the remote (origin) and branch name allows it to target a specific repository and branch to pull or push changes.

Specifying the branch name is useful when you are working on multiple branches locally or need to interact with a branch different from the one you are currently on.

What is <upstream>?

In Git, the term upstream refers to the remote branch that the local branch is currently tracking.

What does it mean to have multiple remotes?

Having multiple remotes in a Git repository means that your local repository is configured to interact with more than one remote repository.

List all remote

```
git remote

# To see the URLs associated with each remote
git remote -v
```

View details about the specific remote branch

```
git remote show <remote-name>
```

Rewrite the most recent commit message

```
git commit --amend -m "New commit message"
```

Rewrite older commit messages

```
git rebase -i HEAD~n # interactive rebase for last 'n' commits

<<comment
Interactive rebase has options like
pick: keep the commit as it is
edit: change the commit message and commit content
reword: only change the commit message, but keep the commit content unchanged
comment

# if file content needs to be changed
git add <file-name>

git commit --amend
git rebase --continue
```

Remove local commit

```
git reset --soft <commit-hash>
# removes commit, but committed content stays in the working directory

git reset --hard <commit-hash>
# removes the commit as well as the committed contents
```

Remove remote commit

```
git reset --hard <commit-hash>

git push origin <branch-name> --force
```

How to undo a commit?

```
git revert <commit-hash>

<<comment
creates a new commit that undoes the contents of the mentioned commit hash
comment
```

The difference between revert, rebase, and reset

git revert: Creates a new commit that reverses the changes made by a specific commit, preserving the original commit history.

git rebase: Moves or combines a series of commits to a new base commit, allowing you to "replay" commits from one branch onto another or to reorder, squash, or edit individual commits.

git reset: Moves the current branch's HEAD backward to a specific commit, potentially modifying or discarding recent commits.

Which command is used to delete a specific commit?

git rebase: used for specific commit removal

git reset: used for removing the latest commit

How to take commits of one branch and put them in another branch?

```
# using git cherry-pick

git checkout <target-branch>

git log <source-branch>

git cherry-pick <commit-hash> # specific commit
# or
git cherry-pick <commit-hash1> <commit-hash2> # multiple commit
# or
git cherry-pick <start-commit-hash>^..<end-commit-hash>

git add <resolved-files>
git cherry-pick --continue

# using git rebase

git checkout <target-branch>

git merge <source-branch>

git add <resolved-files>
git commit
```

What is a diverged branch?

A diverged branch in Git is a branch that has developed independently from another branch since they last shared a common commit history.

Let's say there was a branch named dev from which two branches were created, named featA and featB

```
A --- B --- C --- D (dev)


         F --- H (featA)
        /
A --- B --- C --- D --- E (dev)
            \
             I --- J (featB)
```

As we can see, dev, featA, and featB shared a common commit history until commit c. After that, all three branches were developed separately. So, we can say all three branches diverged from each other after commit c.

Why use git stash?

When developer needs to work on a different files, feature, branch, and don’t want to commit currently unfinished work, in these kind of scenarios git stash is used.

What git stash do?

When git stash command is performed, usually all uncommitted file changes from current project directory are removed. Instead these changes are stored inside .git as temporary commit. This resets the local git repository to match the last commit.

How to restore stashed works?

```
git status
# All uncommitted files will be listed here

git stash
# or
git stash push -m "Custom commit message passed as args"

<<comment
git stash stores only the uncommitted tracked files with
default commit.
git stash push -m stores the uncommitted tracked files
with custom commit
comment

git stash -u
# or
git stash push --include-untracked

<<comment
This create a new temporary commit that include all
uncommitted files, both tracked and untracked
comment

git stash push <file-name>
<<comment
This only remove the file name provided from project
directory, and save it as a temporary commit
comment
```

```
git stash list
# All stashed commit will be listed here

git stash apply
<<comment
Apply the latest stash without removing it from the
stash list
comment

git stash pop
<<comment
Apply the latest stash and remove it from the
stash list
comment

git stash apply stash@{n}
# or
git stash pop stash@{n}
<<comment
Apply a particular stash
comment

git stash drop
# or
git stash drop stash@{n}
<<comment
Remove specific stash to stash list
comment

git stash clear
# Remove all stash from stash list
```

What is git merge?

git merge incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch.

So, let’s say we want to merge featB to dev

```
git checkout main
git merge featB


           F --- H (featA)
          /
A --- B --- C --- D --- E --- K (dev)
              \      /
               I --- J (featB)
```

It will replay the changes made on the featB branch since it diverged from dev (i.e., D) until its current commit (J) on top of dev, and record the result in a new commit along with the names of the parent commits and a log message from the user describing the changes.

Common branching strategy

main: Latest stable production codebase

dev: Main branch for active development. Other development branches merge here.

stage: Codebase for testing in an identical production-like environment.

prod: Codebase ready to be deployed in production.

setup: For initial project setups or environment configurations (e.g., CI/CD pipelines, testing frameworks).

chore: For non-functional updates like refactoring code, updating dependencies, or improving documentation.

bugfix: For fixing non-critical bugs found during development.

hotfix: For urgent fixes that need immediate production deployment.

release: Prepares code for a new release.