Version Control with git

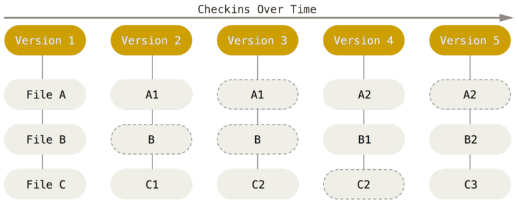
Workshop 2

DSA3101 AY 25/26 Sem II

Introduction to Git

What is Git?

Git is a distributed version control system that tracks changes in files and coordinates work among multiple developers.



Key Features:

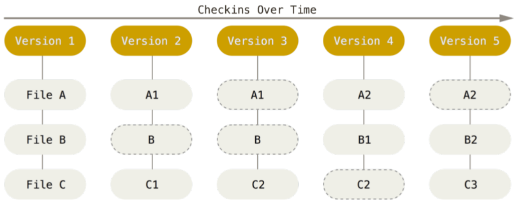
- Tracks every change in your project — nothing is ever lost.
- Allows rollback to previous or alternate versions with ease.
- Git is fully distributed: supports solo and collaborative development workflows.
- Enables experimentation without fear of breaking the project.
- Maintains complete project history.

Introduction to Git

cont'd

What is Git?

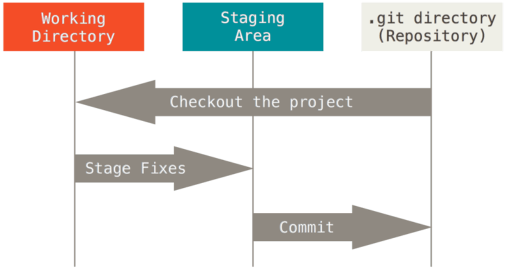
Git is a distributed version control system that tracks changes in files and coordinates work among multiple developers.



Why Git? Unlike traditional systems, Git gives every developer a complete copy of the project history, making it faster and more reliable for team collaboration.

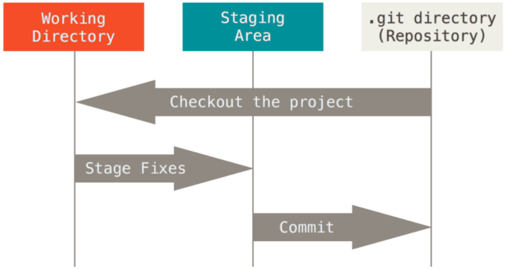
Git is not GitHub — Git works independently of GitHub.

Mental Model



- **Working directory**: your local project folder where you create, modify, and delete files directly on your computer.
- *Analogy*: It's like your desk where you're actively working on documents — these files are editable and not yet tracked as a formal version.

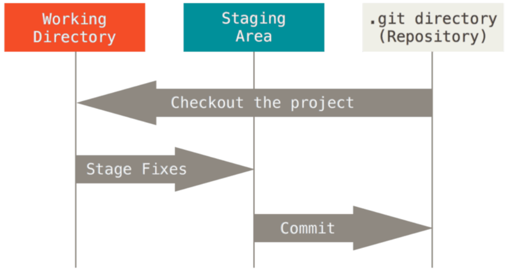
Mental Model



- **Staging area**: a temporary, intermediate zone where you gather the changes you intend to include in your next commit. Only the changes you add here will be part of the next recorded version. The staging area is also commonly referred to as the *index*.
- *Analogy*: Think of it as your shopping cart — you pick and choose which updates to include before "checking out" with a commit.

Mental Model

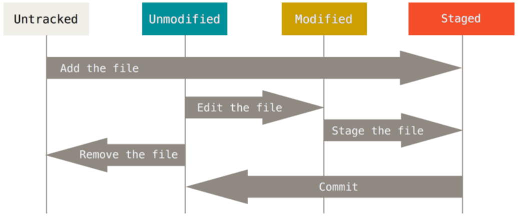
cont'd



- **Repository**: The repository is Git's internal database. It stores the full history of your project, including all committed snapshots, metadata, branches, and configuration. Changes are not permanently recorded until they are committed to the repository.
- *Analogy*: Think of the repository as a digital album — every time you commit, you take a snapshot of your selected work and file it away permanently. Each snapshot is timestamped, organised, and retrievable, so you can always go back and review or restore previous versions.

Mental Model

cont'd



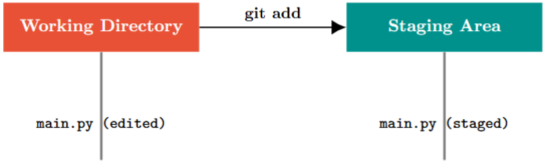
- In the working directory, a file can be either *tracked* or *untracked*. A tracked file is one that Git is monitoring — it can be staged and committed. An untracked file is not yet under version control.
- Once a file is tracked, it can be in one of the following states:
  - **Unmodified/Clean**: Identical to the last committed version.
  - **Modified**: Has changes that have not yet been staged.
  - **Staged**: Changes have been marked to be included in the next commit.

Most Essential Git Commands

We start with three core commands to help you understand the mental model better:

- `git add` — taking a snapshot
- `git commit` — saving to your local album
- `git push` — syncing with the cloud album

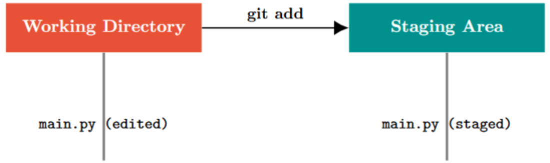
Git Add: Taking a Snapshot



Git Add stages changes from your working directory to the staging area, preparing them for the next commit.

- *Analogy*: Think of git add like taking a snapshot with your camera. You're selecting which files you want to "photograph" and preparing them for the final picture.
- The staging area acts as a "preview" where you can review what will be included in your next commit before making it permanent.

Git Add: Taking a Snapshot
cont'd



Git Add stages changes from your working directory to the staging area, preparing them for the next commit.

- To stage specific file

```
git add filename.txt
```

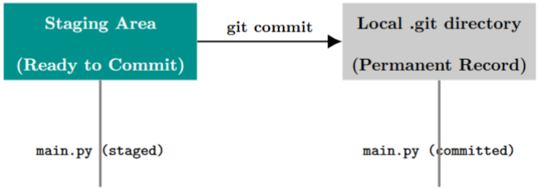
- To stage all JavaScript files

```
git add *.js
```

- To stage all changes within the current directory/folder

```
git add .
```

Git Commit: Saving to Your Local Album

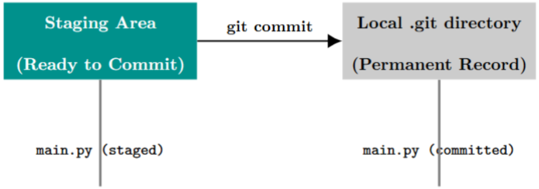


Git Commit creates a permanent snapshot of the staged changes in your local repository with a descriptive message.

- *Analogy*: Think of git commit like adding your snapshots to your personal photo album at your own house. You're permanently saving the staged changes with a description of what happened.
- Write clear, descriptive commit messages that explain what you changed and why. Future you (and your teammates) will thank you!

Git Commit: Saving to Your Local Album

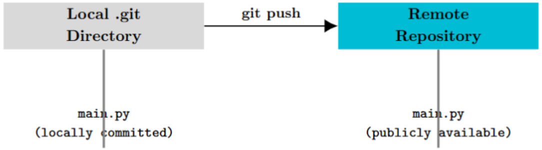
cont'd



Git Commit creates a permanent snapshot of the staged changes in your local repository with a descriptive message.

```
$ git commit -m "Add user login
feature"

$ git commit -m "Fix bug in
payment system"

$ git commit -am "Update and
commit all changes"
```

Git Push: Syncing with the Cloud (Album)

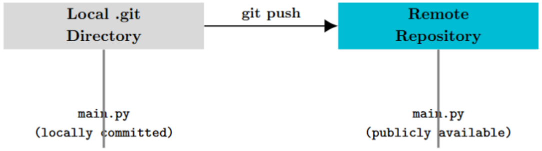


Git Push synchronises your local repository with the remote cloud repository, making your latest commits available to collaborators.

- *Analogy*: Think of git push like syncing your local photo album with your cloud storage. You're uploading your latest commits to cloud and sharing with others.
- Always commit your changes locally before pushing. You can't push unstaged or uncommitted changes!

Git Push: Syncing with the Cloud (Album)

cont'd



Git Push synchronises your local repository with the remote cloud repository, making your latest commits available to collaborators.

- To push to main branch

```
git push origin main
```

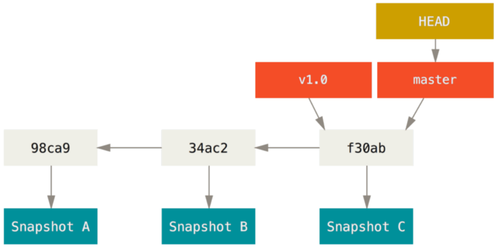
- To push to specific branch

```
git push origin feature-x
```

- To set upstream and push

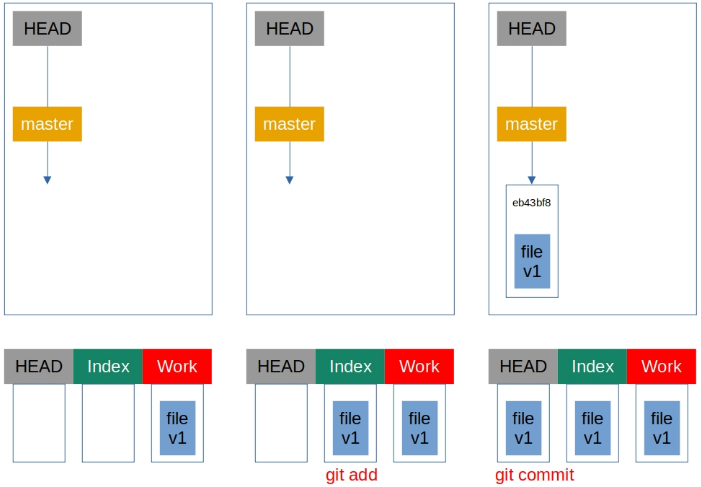
```
git push -u origin main
```

Terminology

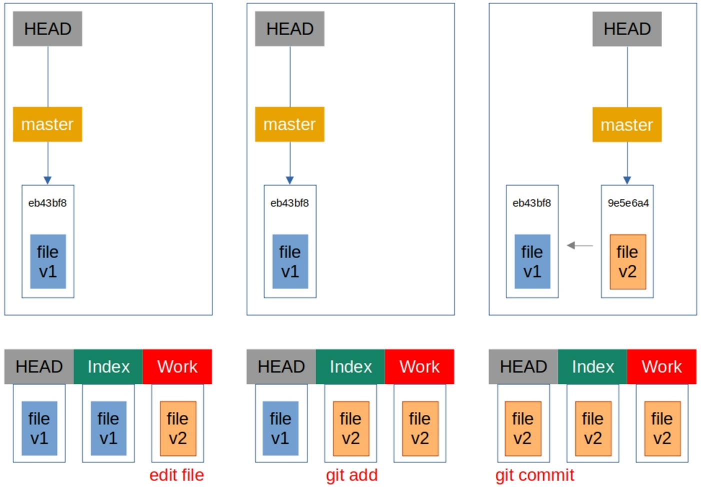


- Each commit or snapshot is assigned a hash, using SHA-1. This is a 160-bit checksum.
- In fact, in the object database, every file is named according to its SHA-1 hash.
- 98ca9 commit is the parent of 34ac2 commit in this diagram.
- There is only a single branch in this repository (master).
- HEAD is a special pointer, that keeps track of which branch you are currently on.
  - `HEAD^` refers to the parent of `HEAD`.
  - `HEAD`∼2 refers to the grand-parent of `HEAD`.
- v1.0 is a tag; a friendly name for a commit.

Example Workflow (Forward)



Example Workflow (Forward)



Common Commands

- `git pull`, `git fetch`
- `git add`, `git commit`, `git push`
- `git status`, `git diff`, `git log`
- `git branch`, `git checkout`, `git switch`
- `git merge`, `git rebase`
- `git reset`, `git restore`, `git blame`

Help!

To get help on any of these commands, run it the command with the `help` option. For instance:

```
$ git log --help
```

Common Commands

cont'd

- Checking on the current situation:

```
$ git status



On branch master
Changes to be committed:
Your branch is up to date with 'origin/master'.
  (use "git restore --staged <file>..." to unstage)

        modified:   file1
        modified:   file2
        modified:   file3
```

Common Commands

cont'd

- Displaying the difference between files

```
$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your
 PR; if we have to read the whole diff to figure out why you're
 contributing in the first place, you're less likely to get feedback
 and have your change
- included.
+ merged in. Also, split your changes into comprehensive chunks if your
+ patch is longer than a dozen lines.
```

Common Commands

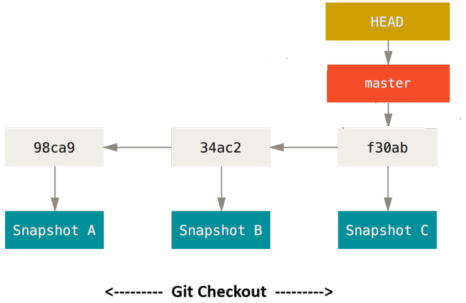
cont'd

- Exploring history

```
$ git log --stat


commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git Checkout & Restore: Time Travel

Git Checkout and **Git Restore**: move to a specific commit in history.



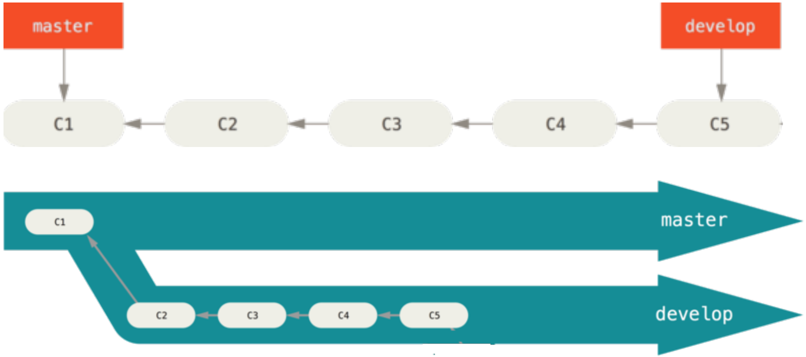
- *Analogy*: Think of these commands like time travel. You can visit any point in your project's history or undo changes to get back to a previous state.
- To go to specific commit

```
git checkout a1b2c3d file.txt
```

- To discard the recent changes

```
git restore --staged file.txt
git restore file.txt
```

Git Branching

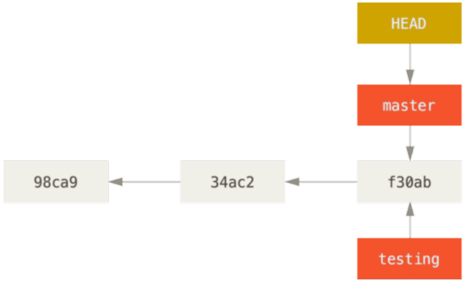


Git Branching

cont'd

- To test out new features/debug things, we can create new branches.
- In git, this is fast and lightweight.
- There are two branches here: `develop` is ahead of `master`.
- We can work on experimental new features in `develop`, while the main codebase in `master` remains stable in production.

Git Branch: Parallel Universes



Git Branch is to create a new branch that is an independent line of development that allows you to work on features, experiments, or fixes without affecting the main codebase.

- *Analogy*: Think of branches like parallel universes. You can work on different features in separate "realities" without affecting the main timeline.
- To list all branches

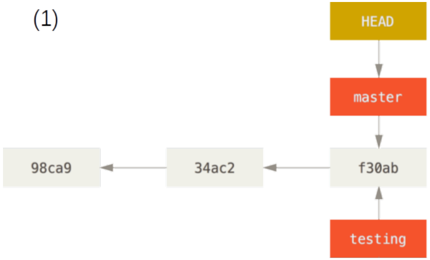
```
git branch
```

- To create and switch to a new branch

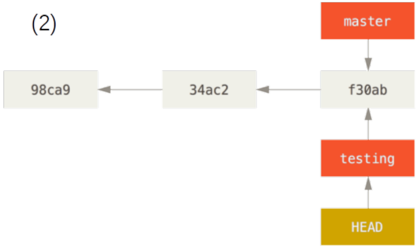
```
git branch testing
git checkout testing
```

Local Branches

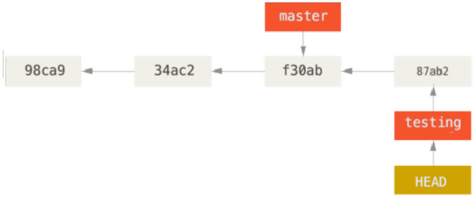
(1)



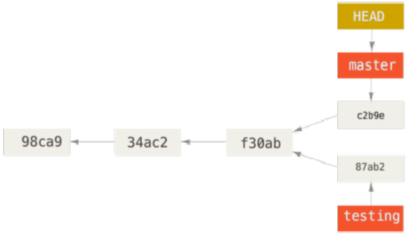
(2)



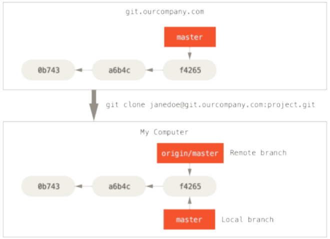
(3)



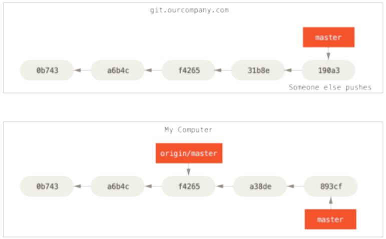
(4)

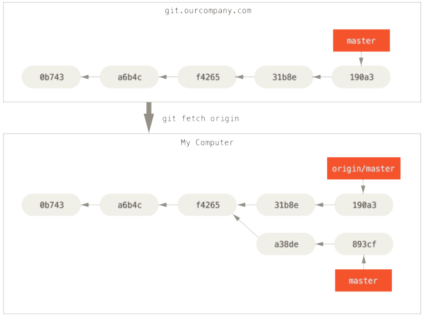


Remote Branches



```
git clone janedoe@git.ourcompany.com:project.git
```

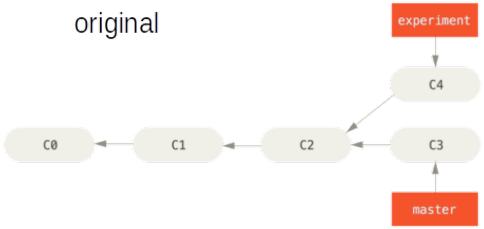


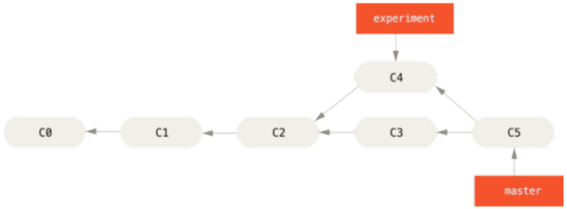


```
git fetch origin
```

Git Merge

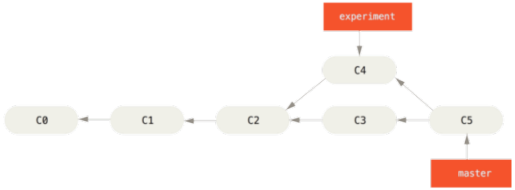
original





With merge

Git Merge: Combining Work



Git Merge combines changes from one branch into another, creating a new commit that includes work from both branches.

- *Analogy*: Think of merging like two rivers flowing together. The separate streams of work combine into one unified flow, bringing together all the changes.
- To merge the experiement branch with the main branch

```
git checkout main
git merge experiment
```

Merge Conflicts: Resolving Differences

⚠ **Conflict in File**

```
function greet() {
<<<<<<< HEAD
return "Hello World!";
=======
return "Hi there!";
>>>>>>> feature-branch
}
```

↓ **Resolve** ↓

✓ **After Resolution**

```
function greet() {
return "Hello there!";
}
```

What are Merge Conflicts? They occur when Git can't automatically merge changes because the same file was modified differently in both branches.

Resolution Steps:

- Identify conflicted files (Git will tell you);
- Open files and look for conflict markers;
- Choose which changes to keep;
- Remove conflict markers;
- Git add and commit the resolved files.

Don't Panic! Conflicts are normal and easy to resolve once you understand the markers.

Github

What is GitHub? GitHub is a web-based platform that provides hosting for Git repositories, plus additional collaboration and project management features.

Key Features:

- Remote repository hosting;
- Issue tracking and project management;
- Pull requests for code review;
- Team collaboration tools.

Relationship: Git is the tool while GitHub is the platform. You use Git locally, then push to GitHub to share and collaborate.

Ignoring Files

- The `.gitignore` file contains patterns that let git know which files should intentionally not be tracked.
- A separator / at the end of a line indicates that it is a folder that should be ignored.

Examples

1. `hello.*` matches any file or directory that begins with `hello`.
2. `/hello.*` matches `hello.R` but not `dir1/hello.R` (pattern restricted to only this directory).
3. `foo/` will match a directory and everything under it, but not any file named `foo`.
4. `foo/*` will match any files under the directory `foo`, but not `foo/bar/hello.R`. The asterisk does not match the / separator character.

git Command Line

- The git command line works from the git-bash shell.
  - Full documentation for every command is accessible through `git <command_name> --help`.
- Several commands can take revisions, and file paths as input.
  - A revision can be a commit, branch or a tag
- Revisions and file paths can be separated using `--`

Examples

- `git show abde09`
  - Provides details about commit that begins with abde09.
- `git show abde09 -- test.sql`
  - Provides details about file test.sql in commit that begins with abde09.
- `git diff 6c05cfc..b1ab8f7  -- src/01_git_workshop.Rmd`
  - Shows differences between file `src/01_git_workshop.Rmd` between older commit `6c05cfc` and newer commit `b1ab8f7`.

Other Useful git Commands

- `git bisect`, `git blame`
- `git rebase`
- `git log -S`
- `git grep`

Using github in DSA3101 Projects

- Create pull requests and tag your team-mate to review the code.
- As a reviewer, read, run and suggest improvements.
- Create topic branches, not individually named branches.
- Use the project management features to track issues and tasks.
- Use the webpage to document your model or API.

Help!

1. The git website.
2. A very neat interactive visualisation
3. On the magic of git.
4. Github documentation pages.

1. Use it for your own single-person projects.
2. Use it in school projects.

Do not be afraid of “breaking” things. With git, it is almost *always* possible to recover.