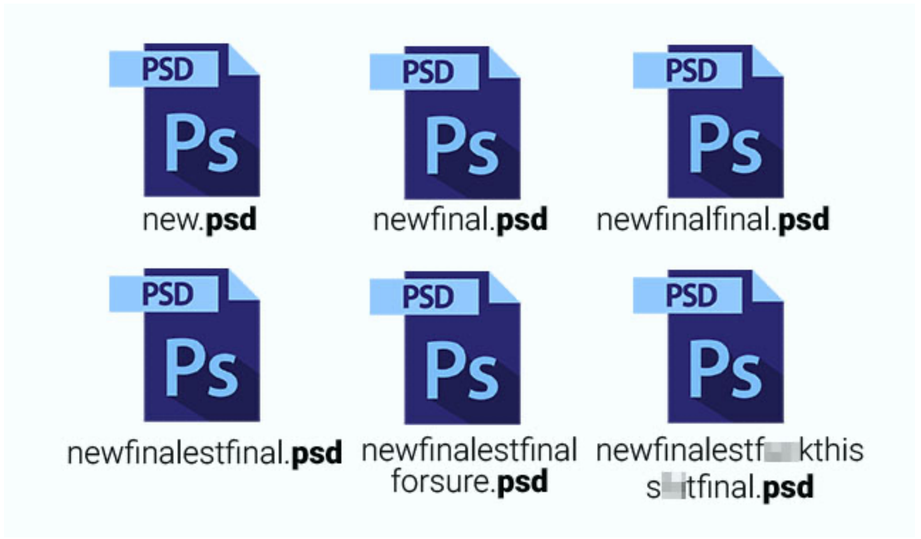
Introduction to Git

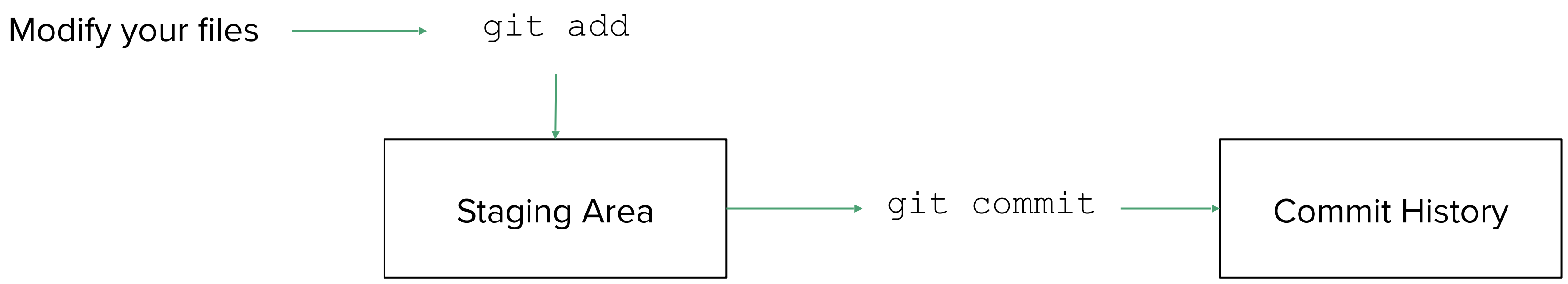
Jeriel Ng (@jerielng)
Mobile Software Engineer

What is Git?

- Version control system
- Developed by Linus Torvalds (Linux)
- Why is version control important?
  - Logged history of changes
  - Collaborative codebase


new.psd
newfinal.psd
newfinalfinal.psd
newfinalestfinal.psd
newfinalestfinal forsure.psd
newfinalestf[redacted]kthis s[redacted]tfinal.psd

Commit Process



What is a commit?

A single point of time in your Git history that contains a log of changes

Commit Messages: Doing them Effectively

What’s in a good commit message?

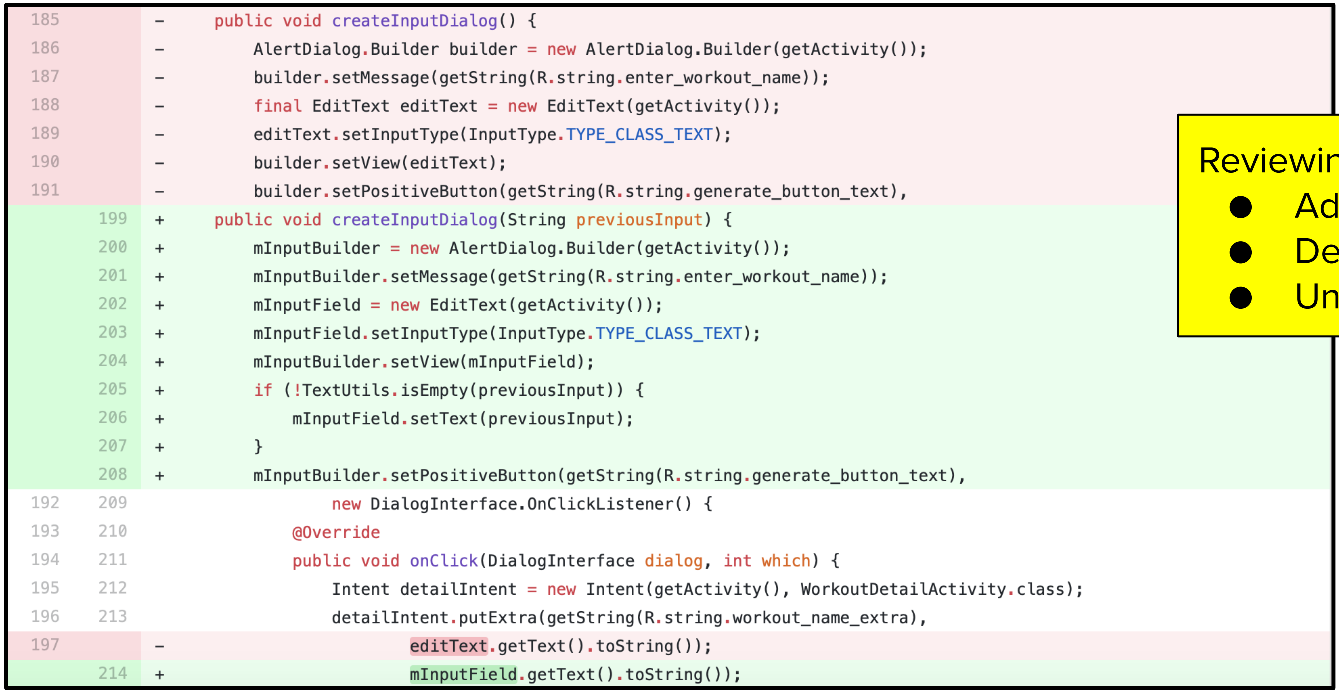
A one-line title that effectively captures the work (if applicable, prefix it with a ticket number)
A description for any details

```
git commit -m “<ticket-number>: <title>” -m “<description>”
```

Why it matters
Track bugs more easily
Simplify pull request reviews

For more details: An effective guide

Commit History: Reviewing Your Changes



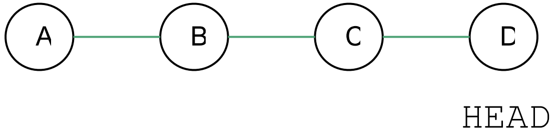
```
-    public void createInputDialog() {
-        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
-        builder.setMessage(getString(R.string.enter_workout_name));
-        final EditText editText = new EditText(getActivity());
-        editText.setInputType(InputType.TYPE_CLASS_TEXT);
-        builder.setView(editText);
-        builder.setPositiveButton(getString(R.string.generate_button_text),
+    public void createInputDialog(String previousInput) {
+        mInputBuilder = new AlertDialog.Builder(getActivity());
+        mInputBuilder.setMessage(getString(R.string.enter_workout_name));
+        mInputField = new EditText(getActivity());
+        mInputField.setInputType(InputType.TYPE_CLASS_TEXT);
+        mInputBuilder.setView(mInputField);
+        if (!TextUtils.isEmpty(previousInput)) {
+            mInputField.setText(previousInput);
+        }
+        mInputBuilder.setPositiveButton(getString(R.string.generate_button_text),
                 new DialogInterface.OnClickListener() {
             @Override
             public void onClick(DialogInterface dialog, int which) {
                 Intent detailIntent = new Intent(getActivity(), WorkoutDetailActivity.class);
                 detailIntent.putExtra(getString(R.string.workout_name_extra),
-                        editText.getText().toString());
+                        mInputField.getText().toString());
```

Reviewing your diff:

- Additions - green lines
- Deletions - red lines
- Untouched - white lines

Commit History: Undoing Your Changes

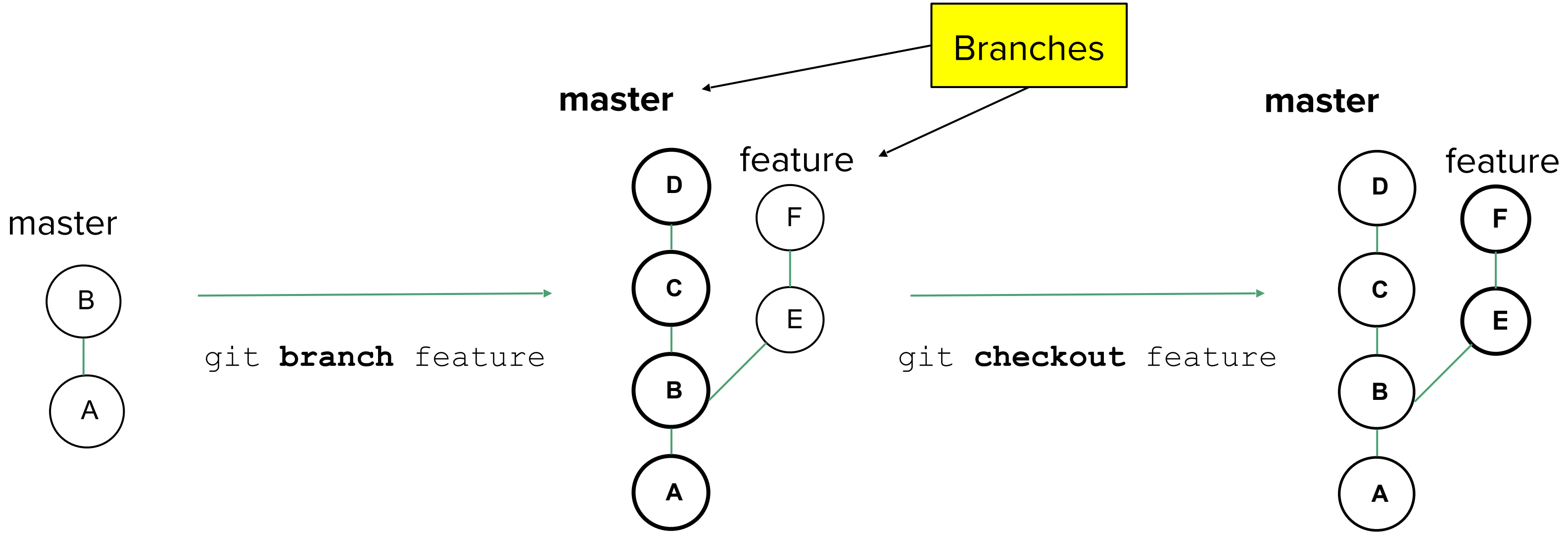
First off,
what is `HEAD`?



`HEAD` is the most recent commit of the branch you're currently on

`git reset --hard`	**Default** `git reset --mixed`	`git reset --soft`
Completely discard changes, whether staged or not	Removes files from staging area, but keeps changes	Moves `HEAD` pointer, but keeps changes **in** the staging area

Alternate Timelines: Branches & Forks



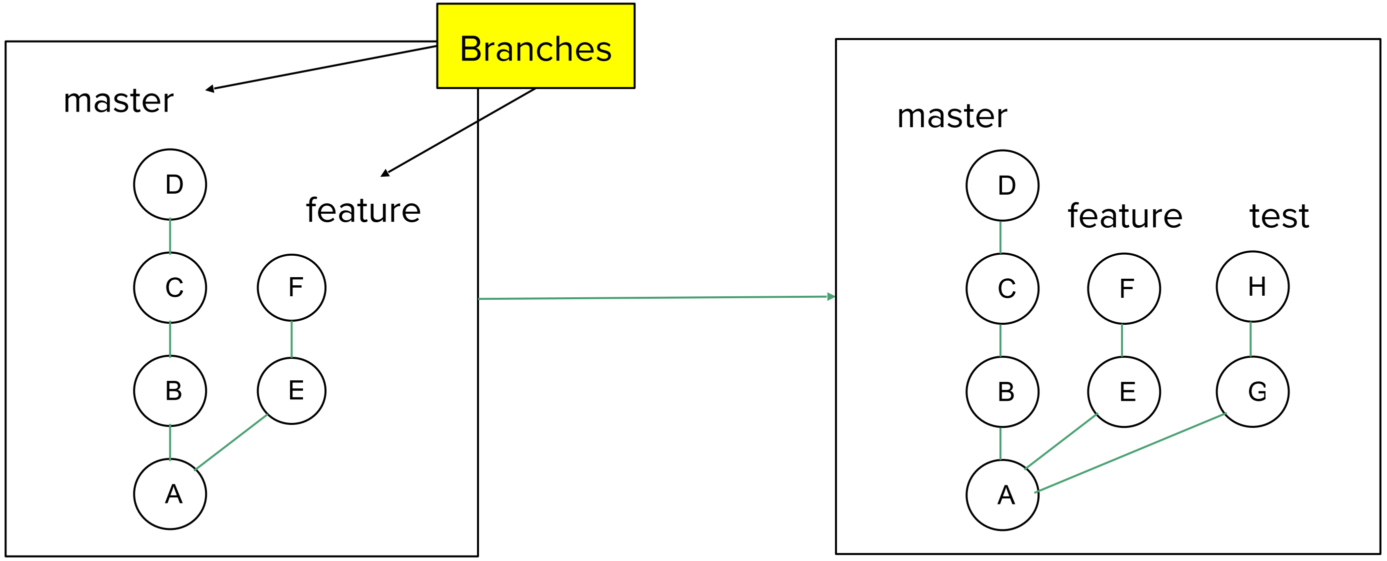
```
git branch feature
```

```
git checkout feature
```

You can also accomplish this with a single line:

```
git checkout -b feature
```

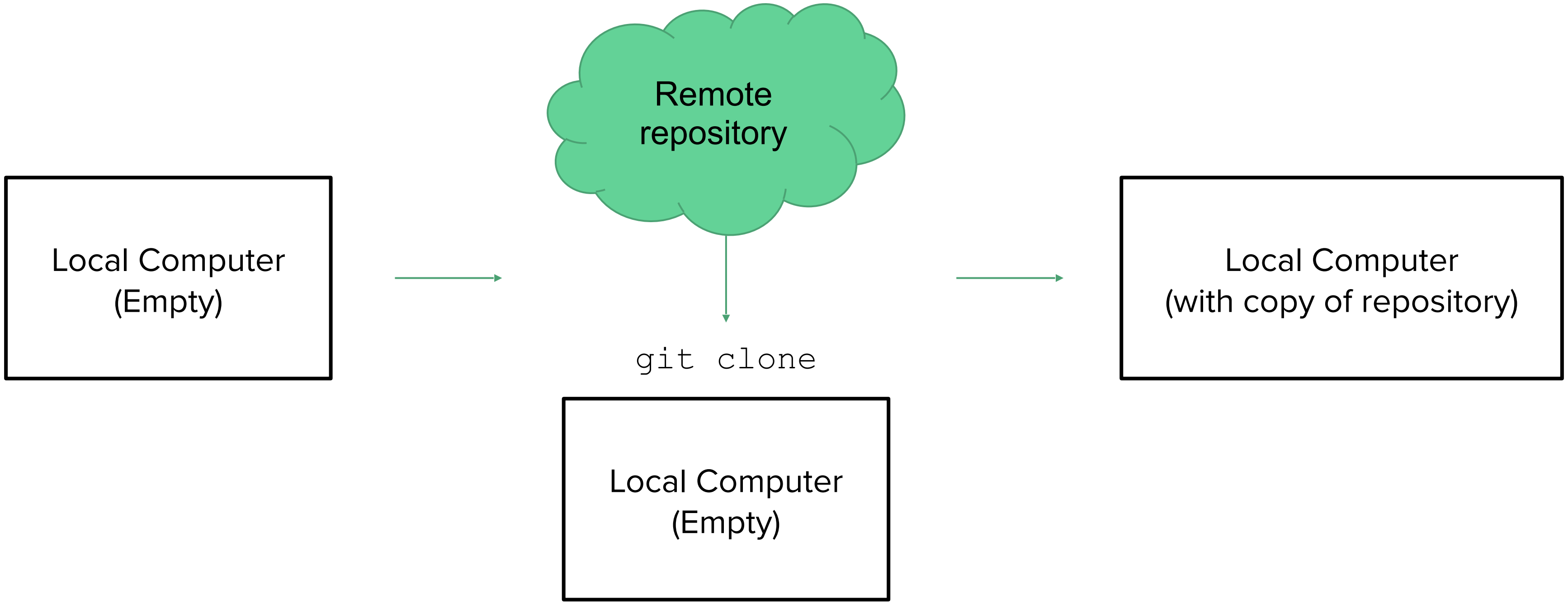
Alternate Timelines: Branches & Forks



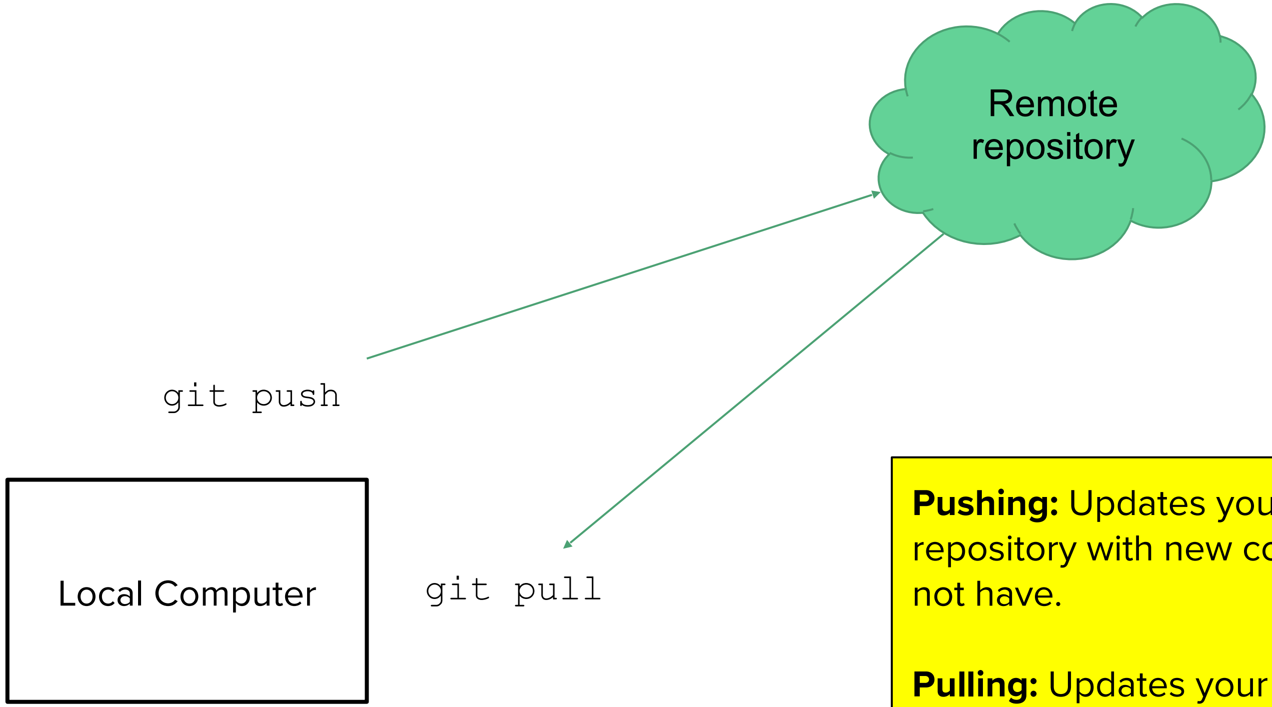
`awesome_project`

Fork of `awesome_project`

Working with Remotes: Setup



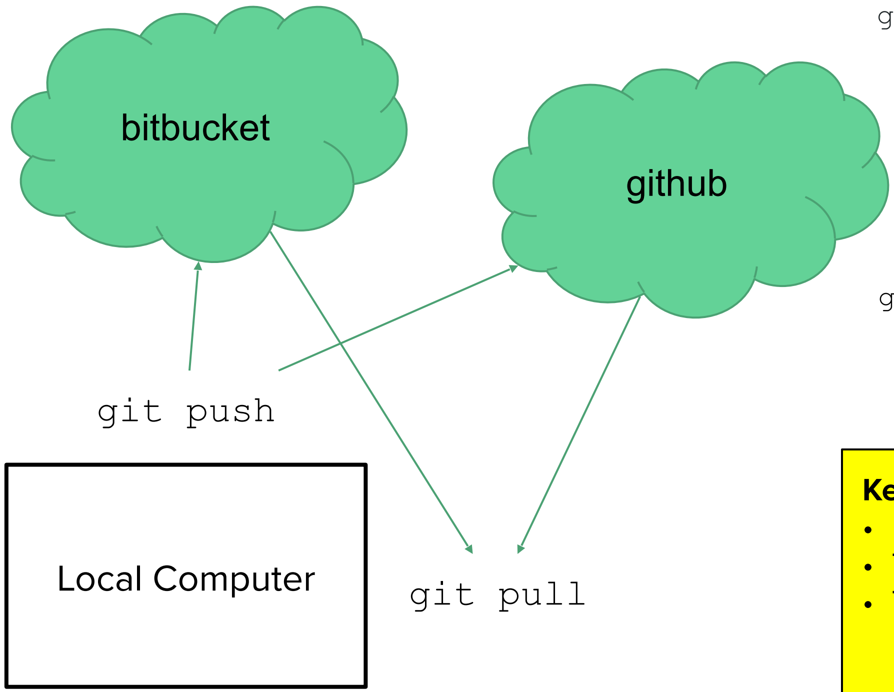
Working with Remotes: Updating



Pushing: Updates your remote repository with new commits it does not have.

Pulling: Updates your local repository.

Managing Multiple Remotes



Cheat Sheet Commands

```
git remote -v
git remote add <remote-name> <remote-url>
```

How to Push/Pull

```
git pull/push <remote-name> <branch-name>

     e.g. git push bitbucket develop
```

Key Takeaways

- It's the same repo hosted in different places
- They're essentially forks
- The URL is key to connecting to a remote
    - URLs can either be SSH or HTTPS

Where are these “remotes”?

GitHub

GitLab

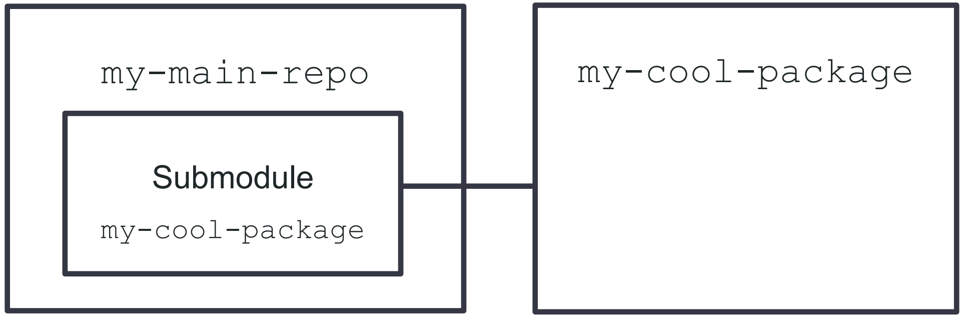
Bitbucket

Built on top of Git

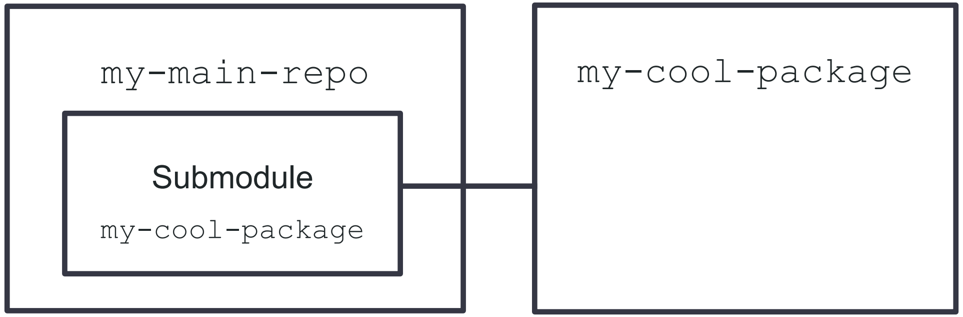
- GUI-based platforms
- Features:
    - Repository Hosting
    - Pull Requests
    - CI/CD Integration

Submodules: What are they?

- Containing another Git repository within your Git repository
- Why use a submodule?
  - Share common code across multiple repositories
  - Isolate your code as an encapsulated piece
  - Maintain a separate commit history



Managing your Submodules



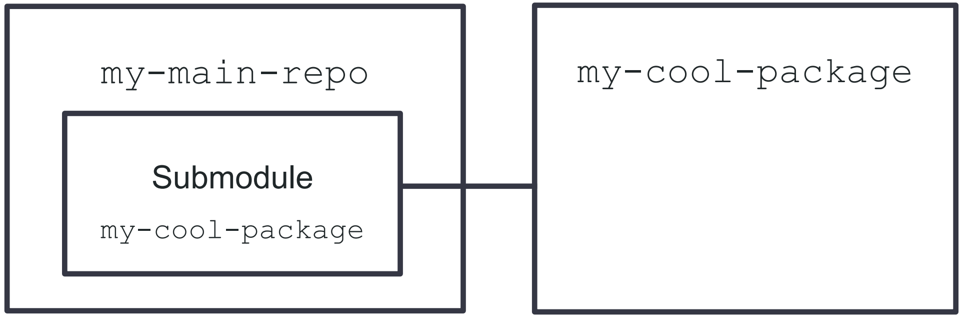
How does it work?

You’re not committing the submodule’s code. You’re committing a reference to the submodule + its commit hash.

How do I update it?

Use Git to update the submodule to the latest commit

Common Submodule Issues



Commands

```
git submodule <follow-up command>

git submodule add
git submodule init
git submodule update
```

My submodule folder is empty

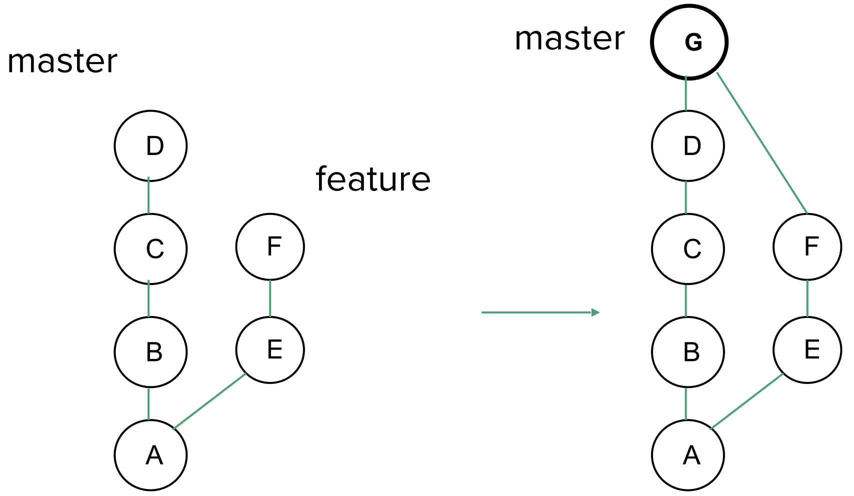
You need to initialize and update it:

```
git submodule init
git submodule update
```

My submodule doesn't have the changes I just made

Check the commit hash it's pointing to. It may not have your latest commit in the original project.

Merging vs. Rebasing

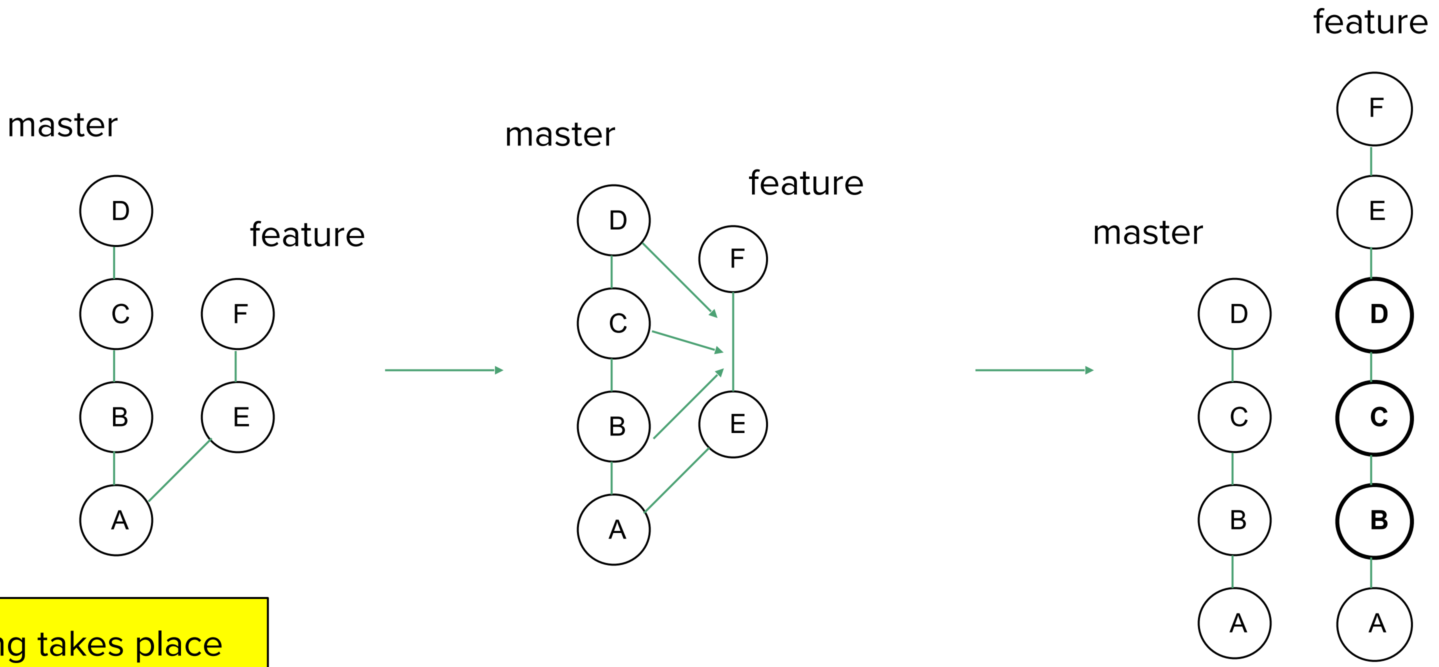


G is a newly created “merge commit” to combine the histories of both master and feature

```
(master): git merge feature
```

Notice merging takes place from the **master** branch, not the **feature** branch

Merging vs. **Rebasing**

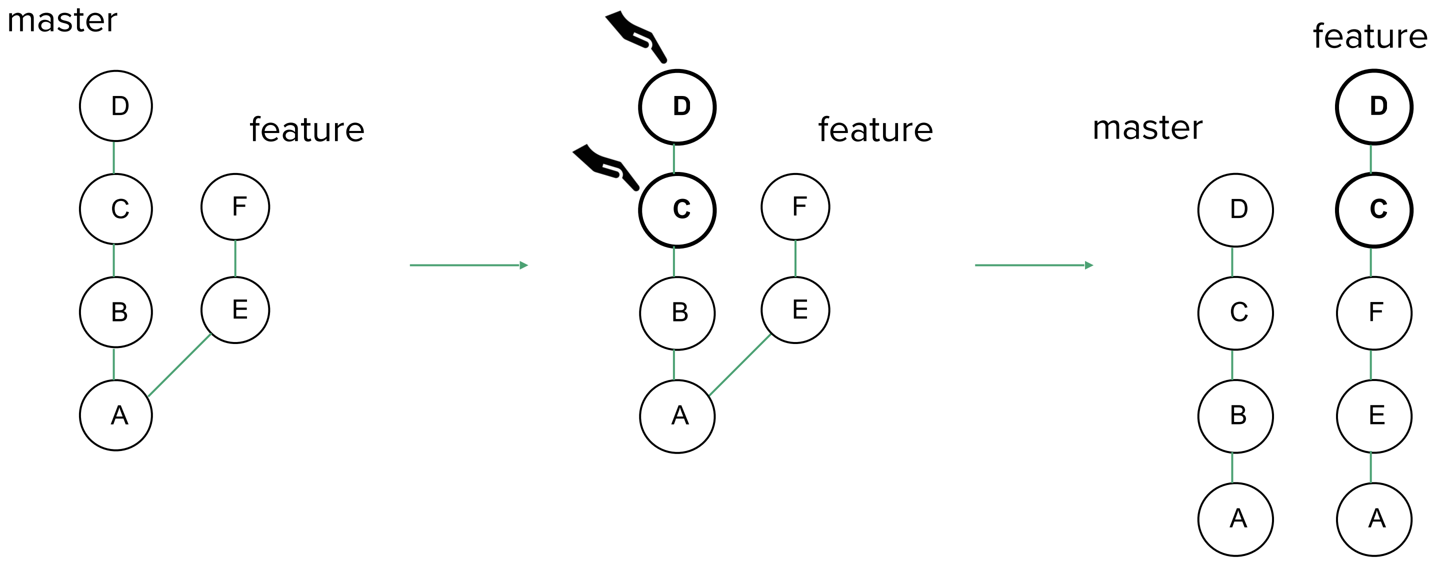


`(feature): git rebase master`

Notice rebasing takes place from the **feature** branch, not the **master** branch

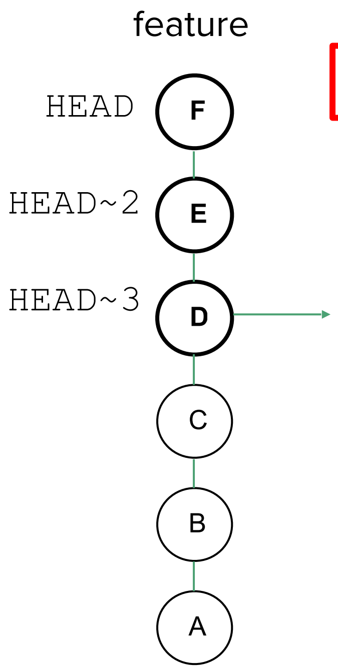
Fun with Commits: Cherry-Picking

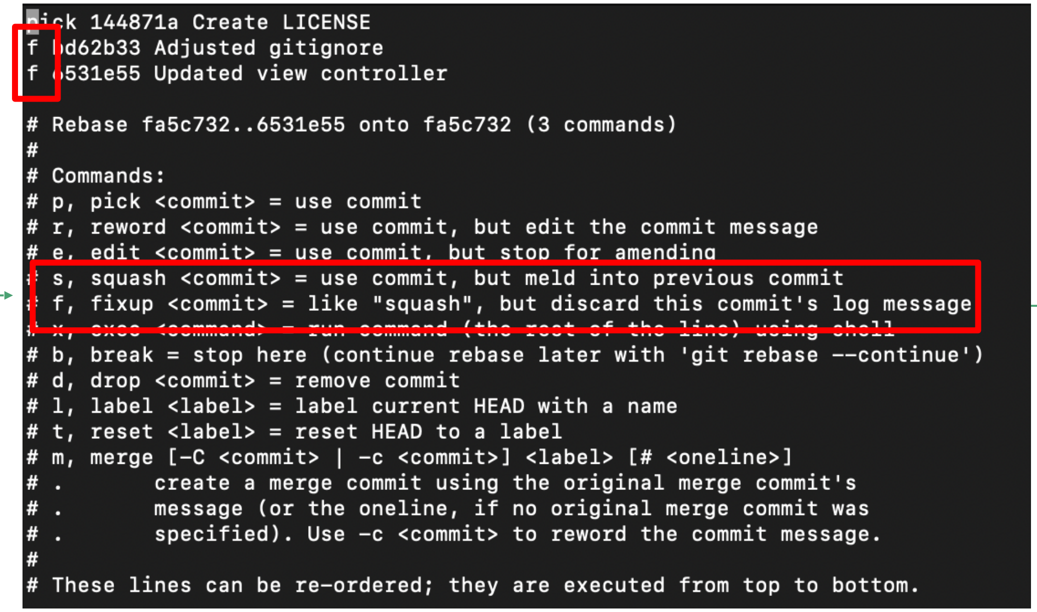
Note:
When rebasing and cherry-picking, the commit hash will be **different** from the original since these are technically **separate** commits.



```
(feature): git cherry-pick C D
```

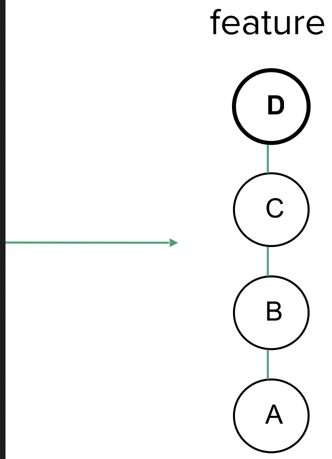
Squashing Your Commits





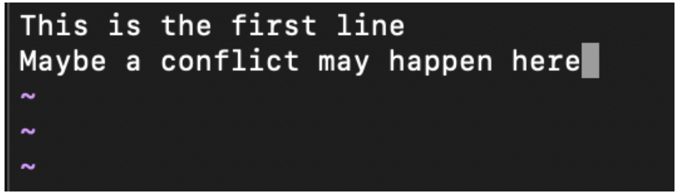
```
pick 144871a Create LICENSE
f bd62b33 Adjusted gitignore
f 6531e55 Updated view controller

# Rebase fa5c732..6531e55 onto fa5c732 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
```



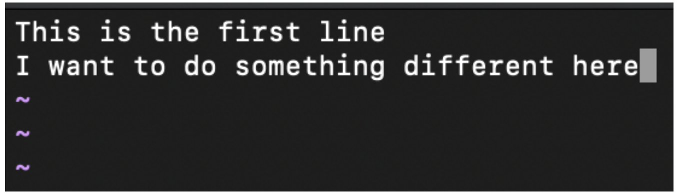
(feature): `git` **`rebase`** `-i HEAD~3`

Merge Conflicts & How to Resolve Them



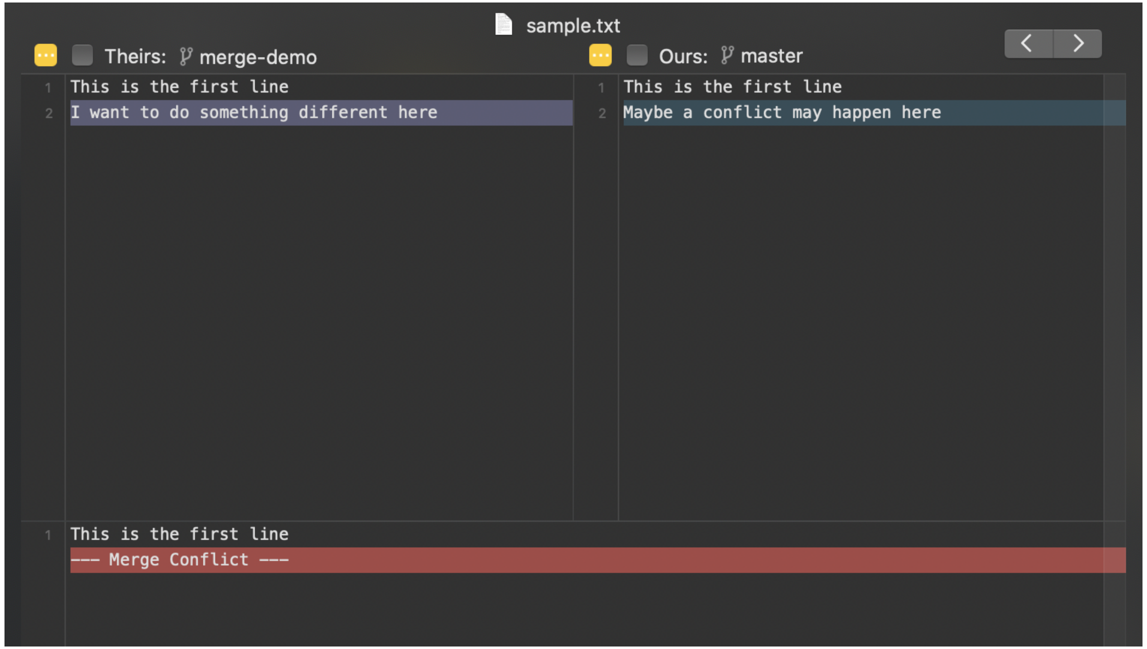
```
This is the first line
Maybe a conflict may happen here
~
~
~
```

Coder A's changes



```
This is the first line
I want to do something different here
~
~
~
```

Coder B's changes



sample.txt

Theirs: merge-demo

```
This is the first line
I want to do something different here
```

Ours: master

```
This is the first line
Maybe a conflict may happen here
```

```
This is the first line
--- Merge Conflict ---
```

Merge Conflicts & How to Resolve Them

- Best approach: **Use a text editor or GUI tool** (Especially for big files)
  - Visual Studio Code
  - GitKraken
  - Fork
  - SourceTree
- Have an understanding of **who** changed **what** and **why**
- Run your code afterwards to make sure it compiles

Freeform Implementations

- Not one single way to use Git
  - Update Strategies
  - Branching Strategies
    - Gitflow
    - Triangular Workflow
  - Repo structure
    - Monorepo vs Polyrepo
- There is no single best approach
- As a team, agree on a consistent way of working

Further Resources

- Git Reference Manual
- Interactive Tutorials on Git
- GitHub Learning Lab
- Desktop GUI tools for Git
    - Fork
    - GitKraken
    - Sourcetree

Questions