

# Git Gud Workshop

Advanced Topics Session  
UWB ACM, Spring 2020  
Presented by Lizzy

# Agenda

- Quick Review
- All About Branches
- What is a Merge?
- Rebasing
- Tips and Tricks
- Git as a Portfolio

# Quick Review

What everyone should already know about git, and a few things to do to get ready for this workshop



# Commands You Should Know...

- `git clone`
- `git status`
- `git add`
- `git commit`
- `git push`

These are all commands you should already know, and we will not spend time in this session explaining how to use them.

# Get ready for this workshop...

- You should already have git installed on your machine.
- You should already have a GitHub account.

Lastly...

- Open this webpage in your browser:
  - <https://github.com/UWB-ACM/Git-Gud-Spring2020>
- Clone this repository to your computer.
- You will not be pushing anything to this repository; all your work will be local.

# All About Branches



# What is a branch?

Most people will be familiar with the master branch; this is the canonical name for the ~~production-ready fully tested~~ *most important* history of changes in a repository.

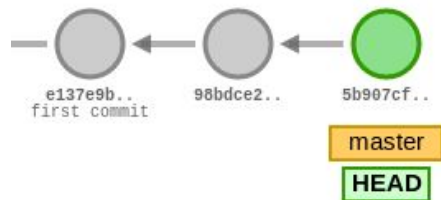
Fundamentally, a “branch” is a label for a commit history chain, and the branch’s label points to the most recent commit in that history chain.

**Branches are one of the most important features of git -- it lets us create separate chains of commits that aren’t related to each other.**

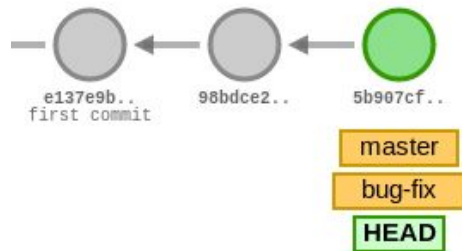
Branches are usually used to develop new features, make quick fixes to existing code, or perform code maintenance tasks *without affecting the master branch*.

# Visualizing branches

Say we are on the master branch. We have a history of 3 commits. Our repository is represented like this:



When we create a new branch (called bug-fix in this example), the new branch's label points to the same commit as the current branch's label:

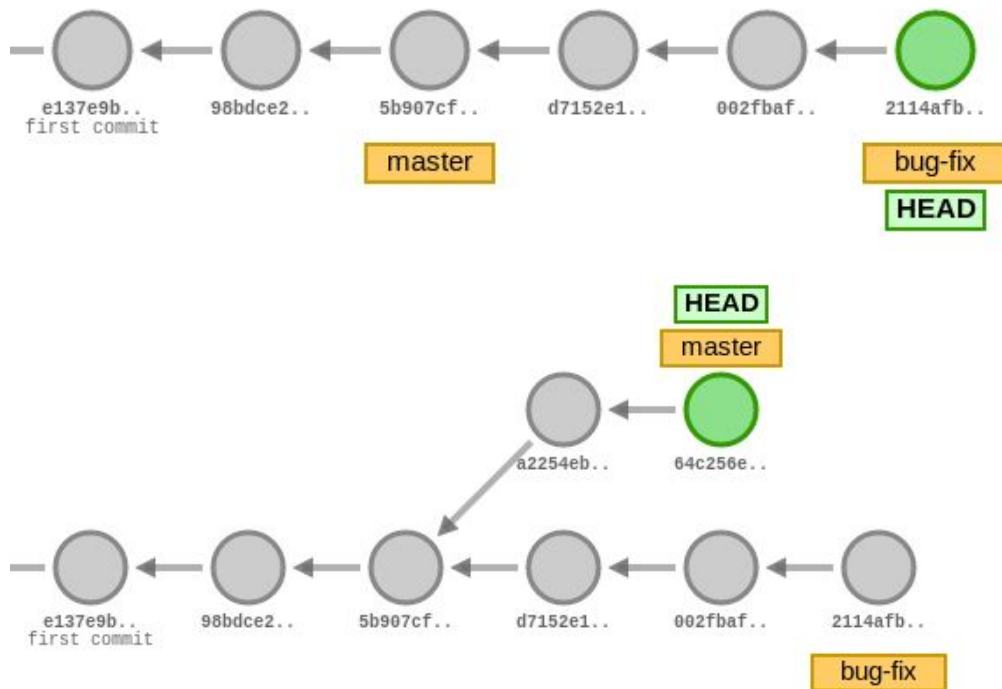




# Visualizing branches - continued

Now that we are on the bug-fix branch, we can create commits that are specifically related to the task at hand: fixing the bug.

At the same time, we (or other developers) can continue adding commits to the master branch.



# Branches in git - Usage

Command	What it Does
<code>git branch &lt;name&gt;</code>	<b>Creates a new branch called &lt;name&gt;</b> , which points to the same commit as the current value of HEAD. Usually this is the most recent commit in your current branch.
<code>git checkout &lt;name&gt;</code>	<b>Switches the HEAD to the existing branch called &lt;name&gt;.</b>
<code>git checkout -b &lt;name&gt;</code>	<b>Creates a new branch called &lt;name&gt; <u>and</u> switches to it.</b> This is a shortcut of the previous two steps.
<code>git branch -v</code>	List all branches in the local repository.
<code>git branch -a</code>	List all branches, both local and remote.

# Let's Practice! (5 min)

Practice navigating branches in the repository you cloned earlier.

What branches already exist in this repository? Use `git branch -a` to find out.

Create a new branch off of master with your GitHub username. Try one of these techniques:

- Single step:
  - `git checkout -b <username>`
- Two steps:
  - `git branch <username>`
  - `git checkout <username>`

## ...what now?

We know how to create branches and switch between them. We also know how to commit changes to the branch we are currently on.

But, if we're developing new changes on a feature branch, how do we get those back into master?

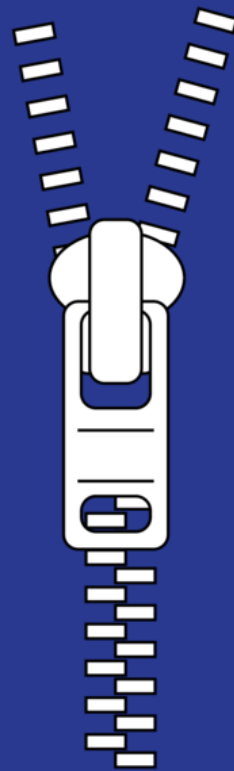
We need to know how to blend two branches back together.

There are two major ways of doing this, and we will talk about them in-depth next:

- Merging
- Rebasing

# What is a Merge?

Zipper Your Commits



# What is a merge?

A merge is a special type of commit.

Normally, commits only have one parent; this gives us the linear commit history that we normally see.

When you perform a merge, you create a commit that has *two* parents.

A merge is most often used to merge two branches together. This facilitates the incorporation of commits on a second branch into your primary branch.

# What is a merge? - continued

Let's say we want to merge two branches, master and feature, and create the merge commit on the master branch.

Our end result will be that all the commits on feature will be incorporated into master.

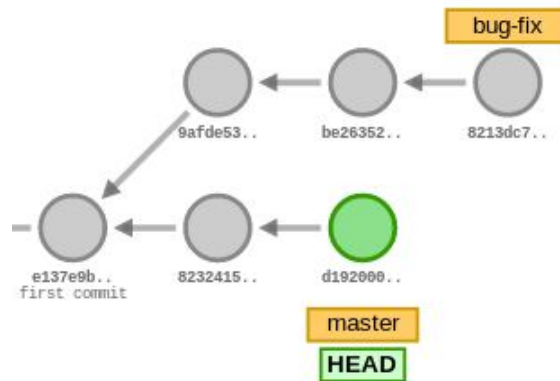
Merging does this by combining all changes from master and feature and blending them. Git then creates a commit which accounts for this specific action that traces its parentage to both master and feature.

# Visualizing merges

Remember our bug-fix branch?

Let's say we want to merge bug-fix into master.

When we do this, we get a final commit graph which looks like this.





# git merge - Usage

Command	What it Does
<code>git checkout feature</code> <code>git merge master</code>	<b>Merges master <i>into</i> feature</b> and creates a merge commit.
<code>git checkout master</code> <code>git merge feature</code>	<b>Merges feature <i>into</i> master</b> and creates a merge commit.

When these commands are successful, a text editor will open and ask you to write a commit message. It will be prepopulated with something like:

Merge branch 'feature' into master

# Let's Practice! (5 min)

In the repository you cloned earlier, you should see a branch called merge1.

Check out that branch with `git checkout merge1`.

Now, merge master into this branch. You may leave the default commit message as is.

# Not all merges are seamless...

We said earlier:

When these commands are successful, a text editor will open and ask you to write a commit message. It will be prepopulated with something like:

```
Merge branch 'feature' into master
```

This only happens when the merge doesn't detect any conflicts.

# What is a merge conflict?

Often, when working on code or documentation, multiple people change the same lines in repository files.

`git` isn't smart enough to know which change is "correct"; only the user should be able to decide.

A merge conflict is `git`'s way of telling us that we need to decide which specific sections of a file should be retained.

# Merge conflict example

In the master branch, we started off with an empty README. We added one line to this document and committed the change:

```
# Hello, world!
```

Meanwhile, on a different branch called `bug-fix`, our extraterrestrial colleague also added one line to the document and committed the change, but it is a different line:

```
# Greetings, Earthlings!
```

# Merge conflict example - continued

When we try to merge bug-fix into master, git detects that each branch has changes to the same line!

So, our terminal output looks like this:

```
$ git checkout master
$ git merge bug-fix
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the
result.
```

# Anatomy of a merge conflict

Git told us what we need to do -- fix the conflicts and commit the result.

Let's open up the file in question to see what this conflict looks like.

```
1 <<<<<< HEAD
2 # Hello, world!
3 =====
4 # Greetings, Earthlings!
5 >>>>>> bug-fix
```

Begin merge conflict

Changes from our HEAD branch, which is master

Section divider

Changes from the other branch

End merge conflict; also shows name of source branch

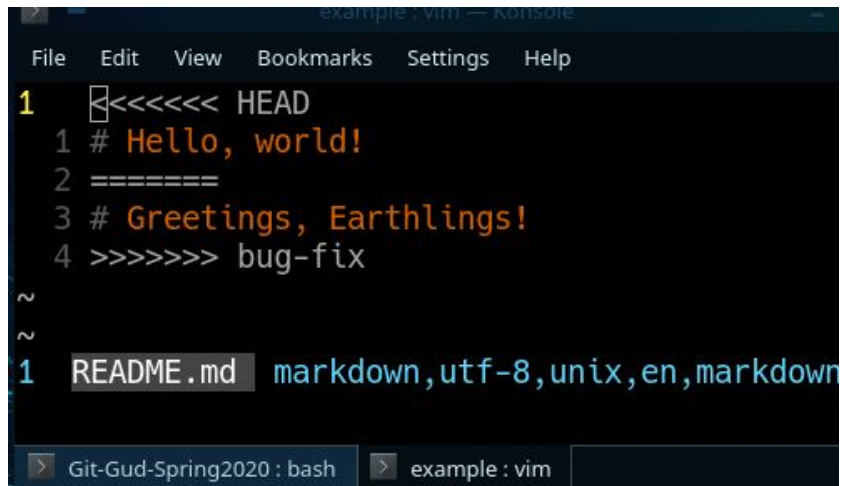
# Resolving merge conflicts

Now it's time to decide which lines to keep and which lines to discard.

Do this in your preferred text editor, IDE, etc and save your changes to the file.

Now run `git commit`; this tells git to create the merge commit we've been trying to create.

Write your commit message as usual, save, and close. All done!



```
example : vim — Konsole
File Edit View Bookmarks Settings Help
1 <<<<<<< HEAD
1 # Hello, world!
2 =====
3 # Greetings, Earthlings!
4 >>>>>>> bug-fix
~
~
1 README.md markdown,utf-8,unix,en,markdown
> Git-Gud-Spring2020 : bash > example : vim
```



# Let's Practice! (5 min)

In the repository you cloned earlier, you should see a branch called `merge2`.

Check out that branch with `git checkout merge2`.

Now, merge `master` into this branch. Are there any conflicts?

Resolve the conflict however you see fit. Now, finish the merge by running `git add` and then `git commit`.

# Rebasing

Or, How I Learned to Stop  
Worrying and Love the  
--continue



# What is a rebase?

Generically, rebasing in git lets the user reapply commits to a different parent commit.

This action lets the user rewrite history! (Commit history, that is.)

When the rebase action is performed, it takes all the specified commits to be reapplied, and replays them one-by-one on top of the new parent commit.

Rebasing can do other things as well, but all rebase actions share this quality.

# But, why use rebase instead of merge?

There are two philosophies in git repo management: have a clean, linear history, or have a history which shows the development process clearly.

Rebasing doesn't create a new commit, whereas merging does. This gives us a clean, linear history, and especially in industry, this is desirable.

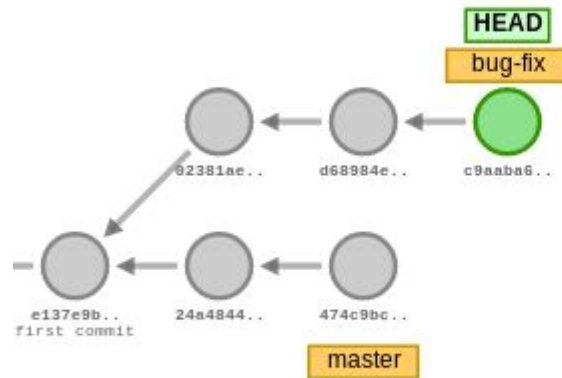
Merging has the advantage of being easier to understand conceptually. It also shows us the full history of development in the repository -- hopefully good, sometimes bad, and probably ugly.

# Visualizing rebasing

In this scenario, we have 3 commits on our bug-fix branch, but master has more recent updates that we also want in our branch.

We do this by **rebasing bug-fix onto master**.

Now, the bug fix commits are parented to master's most recent commit instead of an old, out-of-date commit, and the bug-fix branch contains those new updates.



# git rebase - Usage

Command	What it Does
<code>git checkout feature</code> <code>git rebase master</code>	<p>This command replays all new commits on feature on top of master's most recent commit.</p> <p>“New commits” refer to commits that have been made on the current branch (feature) after the current branch was branched from the target branch (master).</p>

Full documentation is available at:

<https://git-scm.com/docs/git-rebase>

# Let's Practice! (5 min)

In the repository you cloned earlier, you should see a branch called `rebase1`. (Run `git branch -a`).

Check out that branch with `git checkout rebase1`.

Now, you're going to rebase this branch's changes onto master. Do this by running `git rebase master`.

# Conflicts happen when rebasing, too

Just like merging can introduce conflicts, rebase can introduce conflicts for the same reasons. If the two branches modify the same lines or sets of lines, you will have to resolve the conflict.

Because rebasing replays commits in order, you may encounter conflicts at multiple steps during the rebase action.



# Anatomy of a rebase conflict

Git's output may look scary sometimes, but it is actually very helpful.

When we deal with a conflict in a rebase operation, we see something which looks like this:

```
$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: <commit message from first commit>
```

```
Using index info to reconstruct a base tree...
```

```
M      README.md
```

```
Falling back to patching base and 3-way merge...
```

```
Auto-merging README.md
```

So far,  
so good...

# Anatomy of a rebase conflict - continued

Then we get:

```
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
Patch failed at 0001 <commit message from first commit>
hint: Use 'git am --show-current-patch' to see the failed patch
```

Zoinks  
Scoob!

# Anatomy of a rebase conflict - continued

Lastly, we get instructions on how to deal with the conflict:

Resolve all conflicts manually, mark them as resolved with `"git add/rm <conflicted_files>"`, then run `"git rebase --continue"`.

You can instead skip this commit: run `"git rebase --skip"`. To abort and get back to the state before `"git rebase"`, run `"git rebase --abort"`.

We can see that we deal with this in almost the same way as a merge conflict!

# Resolve the rebase conflict

Remember this? Resolve the issue the same way, including removing the section delimiters (ie the <<<<<< ===== >>>>>> lines). Conflicts can appear in multiple places in the same file.

```
1 <<<<<< HEAD
  # Hello, world!
2 =====
3 # Greetings, Earthlings!
4 >>>>>> bug-fix
```

Begin merge conflict

Changes from our HEAD  
branch, which is master

Section divider

Changes from the other branch

End merge conflict; also shows  
name of source branch

# Resolve the rebase conflict - continued

Now we run:

```
$ git add <resolved_files>  
$ git rebase --continue
```

Git will continue rebasing the next commit, if there is one. Repeat this process if needed until all commits are applied.

If git can autoapply subsequent commits, it will and you do not need to take action.

# Let's Practice! (5 min)

In the repository you cloned earlier, you should see a branch called `rebase2`.

Check out that branch with `git checkout rebase2`.

Now, you're going to rebase this branch's changes onto master. Resolve the conflicts, and run `git add README.md` and `git rebase --continue` to finish.

What happens when you run `git status` now?

## One last note...

Because we are changing the parent commit for our branch and essentially creating new commit records for our branch, git creates a new commit SHA hash for each commit on the branch.

If the old commits were already on a branch in the remote, you will see a message like this:

Your branch and 'origin/rebase2' have diverged,  
and have 2 and 1 different commits each, respectively.  
(use "git pull" to merge the remote branch into yours)

# How do we fix this?

You are rebasing for a specific reason, and that reason shouldn't be deprioritized just because of this mechanical issue.

From here, you have a few choices:

- Learn how to undo a rebase using [git reflog](#)
- Perform the rebase action on a new feature branch, which you can then push to your remote without any issues. (Create the branch before you rebase.)
- Force-push the branch, but only if you know what you're doing.

DO NOT **ever** force-push the master branch, or other shared branches.



# Tips And Tricks

Useful Commands



# git stash

The repository's stash is a special “holding area” for file diffs which are not associated with a commit. Think of it like a “temporary file” in Windows - short-term storage for current tasks in progress.

The stash can store multiple entries (or “refs”), and stores them exactly like a stack; the most recent stash entries are at the top of the stash.

Why would I use it?

- Pull changes into your branch from the remote without needing to commit your work-in-progress.
- Switch branches without needing to commit your work-in-progress.
- Get a clean working tree for whatever reason without losing your work.

# git stash - Usage

Command	What It Does
<code>git stash</code>	<b>Saves all changes in your currently tracked files to the stash, and cleans the working tree.</b> This only saves changes in files which have been committed previously.
<code>git stash -u</code>	<b>Saves all changes to the stash</b> (including both tracked and untracked files), and cleans the working tree.
<code>git stash pop</code>	Retrieves the most recent stash entry, applies the corresponding changes to the files in your working tree, and removes the stash entry if the application was successful.

Full documentation available at <https://git-scm.com/docs/git-stash>

# git add -p

This is arguably my favorite git command ever.

Software development is not a linear process -- we fix issues as we find them, add comments for unrelated tasks, and add new pieces of logic, and more, all in the same sitting.

Instead of adding and committing *all* the changes in a specific file, you can selectively pick and choose which changes you want to add.

Patch adds with the -p flag allow us to pick specific diffs to add to our staging area, so we can construct meaningful commits related to a specific problem we tackled, and write meaningful commit messages.

# git rebase -i <ref>

This command begins an “interactive” rebase for a range of commits you specify. It opens an editor which displays information about each commit in that range, and lets you choose to modify the commits, including:

- Reorder commits
- Blending two commits together (squash)
- Deleting select commits (drop)
- Edit commit, including unstaging files or adding other files (edit)
- Edit the commit message for this commit (reword)

Full documentation is available at:

[https://git-scm.com/docs/git-rebase#\\_interactive\\_mode](https://git-scm.com/docs/git-rebase#_interactive_mode)

# Git as a Portfolio

How to Maximize your Git-Fu



# Why should you care?

Me



The guy she tells me not to worry about



# Cool things to use GitHub for...

## Continuous integration!

- Travis CI integrations
- GitHub Actions

## Code reviews!

- Work on open source projects, or collaborate with friends!
- Receiving (and giving) code reviews makes you a better engineer

## Documentation!

- Good READMEs matter more than you think!



# More cool things to use GitHub for...

## GitHub Pages!

- Make a portfolio website and host it for free using GitHub pages!
- Show off projects (aka repositories) you're especially proud of with their own web pages!
- Write your posts in raw HTML/CSS/JS, or write them in Markdown and use a Jekyll theme.
- If you're feeling ambitious, make your own site theme.



# All done!

Stick around if you have questions.