Git

Sumner Evans

May 10, 2017

Introduction

What is Git?

- Git is a system which tracks changes made to a code base.
- Git was created by Linus Torvalds to help facilitate Linux kernel development.
- Linus's motto when he built Git was to take Concurrent Version System (CVS) as an example of what *not* to do and if in doubt, make the opposite decision. [1]

[1] https://en.wikipedia.org/wiki/Git

What is Git?

- Git is a system which tracks changes made to a code base.
- Git was created by Linus Torvalds to help facilitate Linux kernel development.
- Linus's motto when he built Git was to take Concurrent Version System (CVS) as an example of what *not* to do and if in doubt, make the opposite decision. [1]

[1] https://en.wikipedia.org/wiki/Git

What is Git?

- Git is a system which tracks changes made to a code base.
- Git was created by Linus Torvalds to help facilitate Linux kernel development.
- Linus's motto when he built Git was to take Concurrent Version System (CVS) as an example of what *not* to do and if in doubt, make the opposite decision. [1]

[1] https://en.wikipedia.org/wiki/Git

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called “my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of “my-proj” for backup purposes.
4. You continue development on “my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified `main()`.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? III

How Version Control Systems (VCS) solve this:

- VCS keeps track of *revisions*, changes in the code in entities called *changesets* or *commits*.
- Most VCS allow version merging. That means multiple people can be working on the same file and resolve discrepancies later. Git is very elegant in handling merge conflicts such as this.

Why use Version Control? III

How Version Control Systems (VCS) solve this:

- VCS keeps track of *revisions*, changes in the code in entities called *changesets* or *commits*.
- Most VCS allow version merging. That means multiple people can be working on the same file and resolve discrepancies later. Git is very elegant in handling merge conflicts such as this.

Why use Git?

Git is a VCS so it solves all of the issues I've described. So why use Git over some other VCS?[2]

- It's a *distributed* VCS. That means that you have a full copy of the code and every change ever made by anyone to that code on your local machine. A beneficial side effect of this is that you can work offline.
- It's faster.
- Git rarely fully deletes anything, this is good because you can undo most actions.
- Everyone else is using it.

[2] List inspired by
`https://www.git-tower.com/blog/8-reasons-for-switching-to-git`

Why use Git?

Git is a VCS so it solves all of the issues I've described. So why use Git over some other VCS?[2]

- It's a *distributed* VCS. That means that you have a full copy of the code and every change ever made by anyone to that code on your local machine. A beneficial side effect of this is that you can work offline.
- It's faster.
- Git rarely fully deletes anything, this is good because you can undo most actions.
- Everyone else is using it.

[2] List inspired by
`https://www.git-tower.com/blog/8-reasons-for-switching-to-git`

Why use Git?

Git is a VCS so it solves all of the issues I've described. So why use Git over some other VCS?[2]

- It's a *distributed* VCS. That means that you have a full copy of the code and every change ever made by anyone to that code on your local machine. A beneficial side effect of this is that you can work offline.
- It's faster.
- Git rarely fully deletes anything, this is good because you can undo most actions.
- Everyone else is using it.

[2] List inspired by
`https://www.git-tower.com/blog/8-reasons-for-switching-to-git`

Why use Git?

Git is a VCS so it solves all of the issues I've described. So why use Git over some other VCS?[2]

- It's a *distributed* VCS. That means that you have a full copy of the code and every change ever made by anyone to that code on your local machine. A beneficial side effect of this is that you can work offline.
- It's faster.
- Git rarely fully deletes anything, this is good because you can undo most actions.
- Everyone else is using it.

[2] List inspired by
`https://www.git-tower.com/blog/8-reasons-for-switching-to-git`

How to Get Git

Git is awesome! How do I get it? **Good news: Git is cross platform.**

- **Linux:** Install the `git` package using your distribution's package manager
- **OS X:** I recommend using Homebrew: `brew install git` (`http://brew.sh/`)
- **Windows:** Download the installer from `https://git-scm.com/`

If you need a GUI, check out SourceTree or GitKraken.

You should also setup SSH which is extremely easy, but I will not cover that here as this is a talk about Git not SSH.

Hot to use Git (locally)

- Initialize (`git init`): Initializes a Git *repository* on your local machine in the current *working directory*.
- Add (`git add`): Marks files to include in the next *commit*, an entity which stores the state of the repository at a given time.
- Reset (`git reset`): Opposite of `add`; marks the file as not included in the next commit.
- Commit (`git commit`): Creates a commit.
- Log (`git log`): Shows the history of your repository.
- Diff (`git diff [file]`): Determines the difference between the file's current state and its state at the last commit.
- Git ignore (modify the `.gitignore` file): Any file that matches one of the patterns in `.gitignore` will not be tracked by Git.

How to use Git with a remote

- Add Remote (`git remote add [name] [url]`): Adds a *remote*, a version of the repository hosted externally from your local machine. Most likely on something like Github or an company's internal network.
- Push (`git push -u origin master`): Pushes all changes on the given branch to the remote.
- Clone (`git clone`): Copies the entire repository to the location.
- Fetch (`git fetch`): Retrieves changes from the remote.
- Merge (`git merge`): Merges a branch into another branch. (More on branches later, but in this case, we are merging the `origin/master` into our local `master` branch.)
- Pull (`git pull`): Retrieves any new changes from the remote and merges them with your local changes.

Undoing Things

- Undo the last n commits (given that you haven't pushed them yet) `git reset --hard HEAD~n`
- Undo the n^{th} to last commit by creating a new commit that reverts all of the changes `git revert HEAD~n`
- Somebody's done it before. Just Google it.

Merging Changes I

What happens if multiple developers make changes to the same file? This will cause *merge conflicts*.

There are plenty of tools which you can use to *resolve* such conflicts. None of them are that good because merge conflicts are just terrible in general.

Play around with a bunch of them and see which one you like best. Here are a few to get you started: `Meld`, `KDiff3`, and `vimdiff`.

Merging Changes II

Invoking the *mergetool*: use `git mergetool`.

For each *conflict*, you can choose to take their version, your version, a combination of the two or neither.

Most UIs will give you three panes: one for the *remote* version of the file, one for the *local* version of the file and one for the merged version of the file.

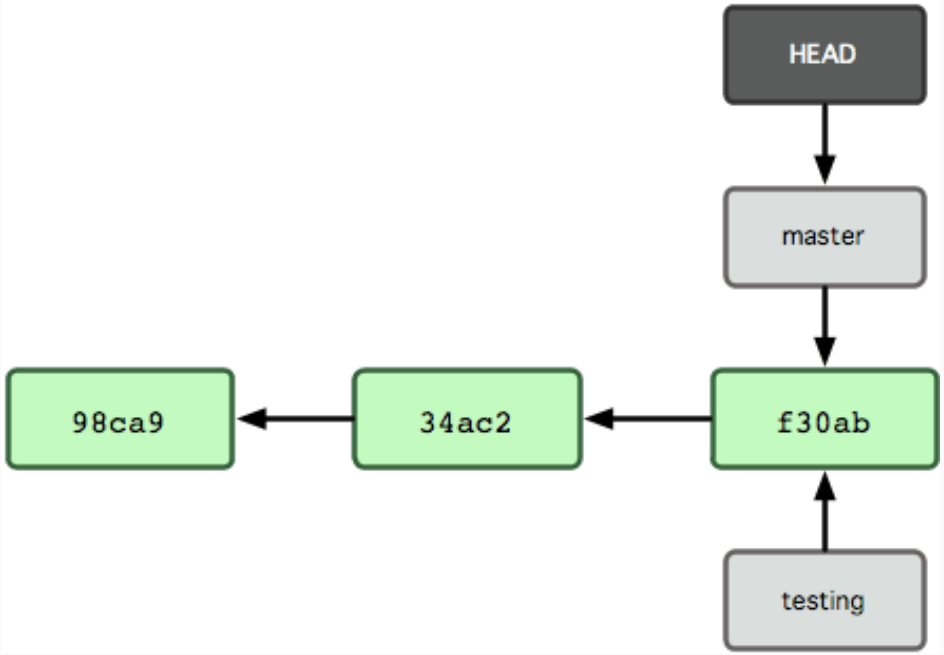
Branches I

Branches allow you to separate develop a given functionality without affecting the original code base.

For example, if you have an established product and you want to add a feature but you are uncertain about its viability, you can create a branch and build a prototype on that branch. If it fails, you can delete the branch and never see it again or if it works, you can *merge* the branch back into `master`.

Branches[3] II

Branches can be thought of as *bookmarks* pointing to a specific changeset.

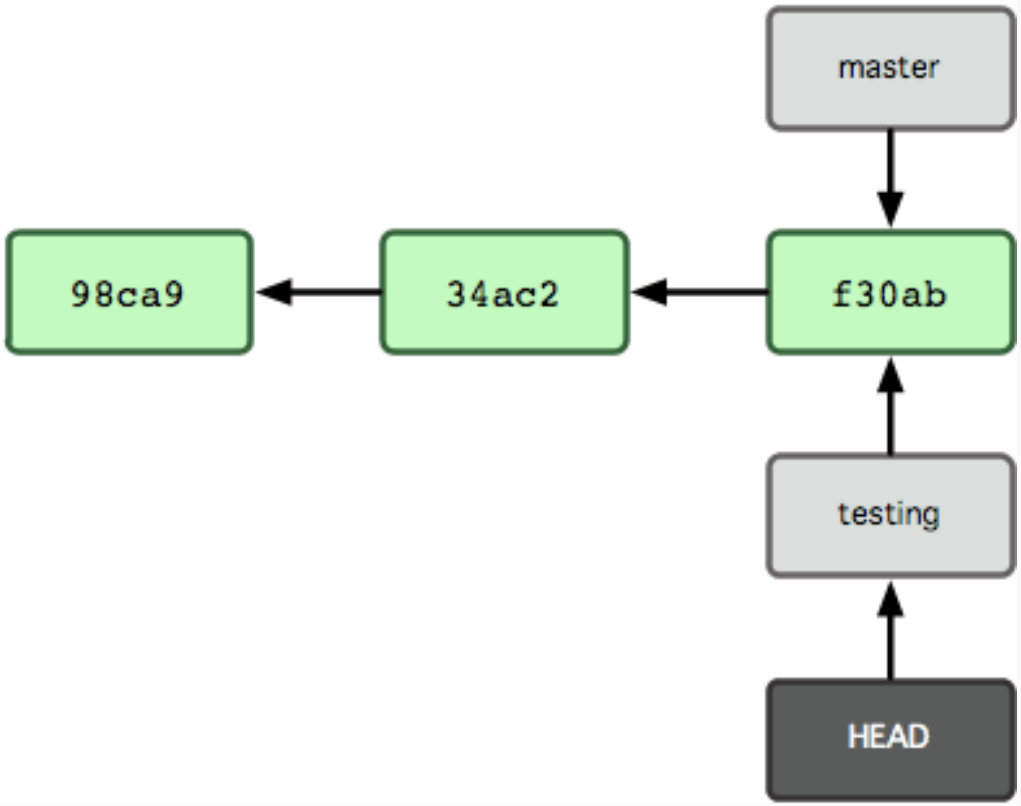


Note, `HEAD` is pointer to the current branch.

[3] Info in the rest of the *Branches* section is mainly from `https://git-scm.com/book/en/v1/Git-Branching-What-a-Branch-Is`

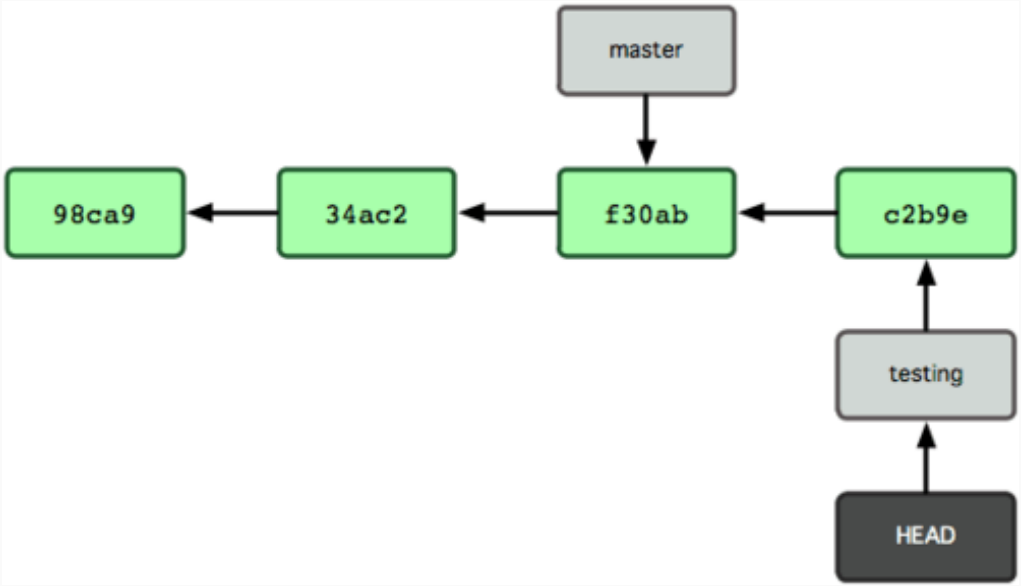
Branches III

To switch branches, use `git checkout [branch]`. This moves `HEAD` to point to your new branch.



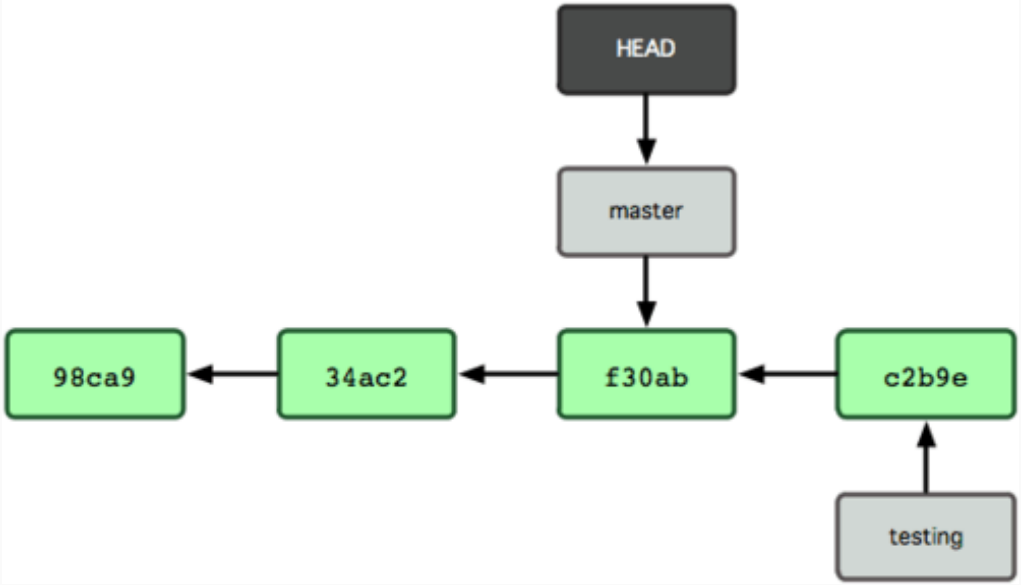
Branches IV

If you commit something to your new branch, the branch pointer moves to the new commit. The pointer to `master` will not move.



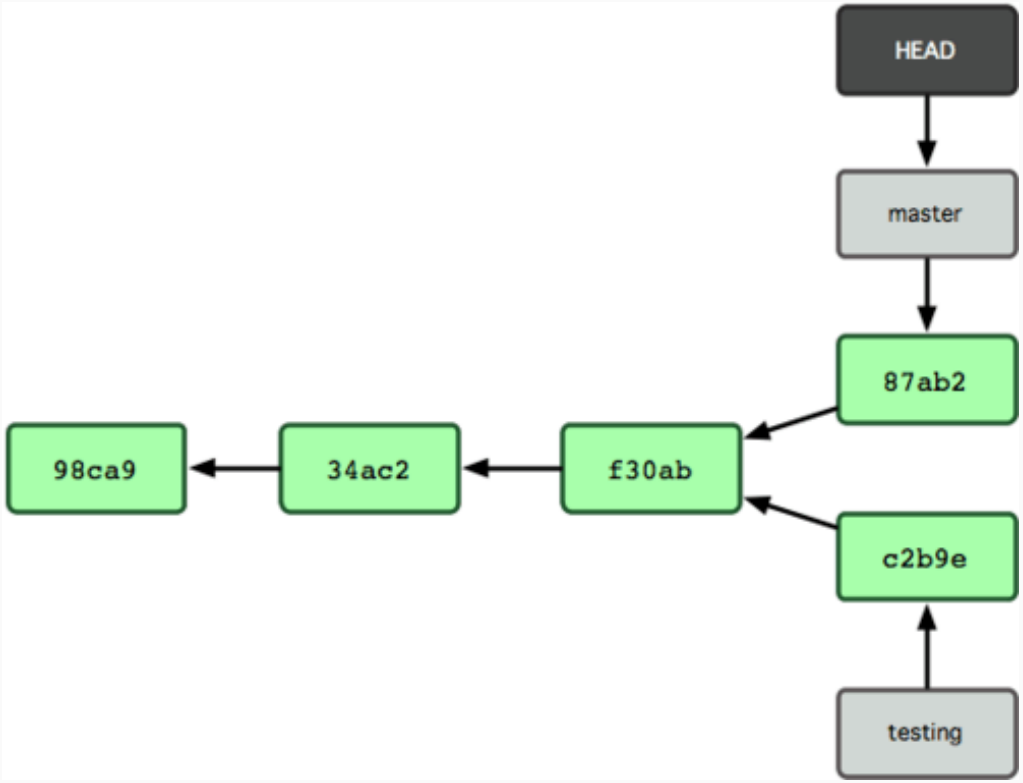
Branches V

Of course, you can always switch back to `master` using `git checkout master`.



Branches VI

If you make a commit on the master branch, the `master` pointer moves to that new commit. At this point, the branch histories have diverged.



Branches VII: How to actually do it

- **Create a Branch:** `git branch [branch_name]`
- **Switch to a branch:** `git checkout [branch_name]`
- **Create a new branch and switch to it:**
  `git checkout -b [branch_name]`
- **List branches:** `git branch`
- **Push branch to remote:** `git push -u [remote_name]`
  `[branch_name]`

Merging Changes: Branch Edition

If you want changes from a different branch in your current branch, you can use `git merge [other branch]`.

When you merge changes from another branch, one of two things will happen[4]:

1. Your branch will be fast-forwarded to the other branch.
   This means that there are no changes in your current branch that are not in the other branch.
2. A merge commit will be created and your branch pointer will be updated to point to this commit.
   This happens when there are changes in the current branch that are not in the other branch.

[4] These are the only ones I can thing of off the top of my head. You can force either of these functionalities with their respective command line options.

Branch and Development Workflow

Branches are extremely scalable so you can ignore them, or use them for everything, it's your choice.

One methodology used by companies in the industry is `Git Flow`. This is a system whereby new branches are created for every bugfix, new feature, release, and hotfix. If you want to learn more about it, look at this website: `http://nvie.com/posts/a-successful-git-branching-model/`

The Random Stuff: Stashing

When you have changes in your working directory, merges and switching branches (sometimes) doesn't work. You have two main options here:

1. Commit your changes. If you can do this, you should.
2. Occasionally you just don't want to commit. In this case, you will want to *stash* your changes using `git stash [save [stash_name]]`.
3. To un-stash, use `git stash pop`. You may have to resolve merge conflicts.

The Random Stuff: Submodules

What is a submodule? It is literally a repository inside of another repository.

Why is this useful? If you have a custom library shared between many projects, you can place that library in a standalone Git repository. Then you can add it as a submodule to your products via `git submodule add [clone_url]`.

The submodule is its own repository so it can be contributed to independently, but it can also be modified and contributed to as a submodule.

The Random Stuff: Rebasing

What is it good for? If you want to keep your graph clean, you can use rebasing to avoid merge commits.

Why would you use this? I don't know. I never have and I normally want to see all of the changes in a graph, but it's a thing that you can do in Git and if you want to learn more, read the docs.

The Random Stuff: Aliases

As one would expect from something designed and built by Linus Torvalds, Git supports the concept of *aliasing* one git command to another name.

For example, you might want to alias `checkout` to `co`. This particular alias can be achieved using
`git config --global alias.co checkout`.

Now you can invoke `git checkout` using `git co`.

Another option is using your shell's alias functionality. This is often more powerful, but that isn't part of this talk.

The Random Stuff: Resources/Tips

I obviously was unable to tell you about everything you can do with Git. I've really only scratched the surface.

- `man git *`: The man pages on Git are good. Use them as your first line of defense.
- `git-scm.com/book/en/v2`: A huge resource about how to do everything Git.
- `gitignore.io`: Generates a `.gitignore` file for a given project type, OS, and IDE.
- `git reset --hard HEAD`: Undoes all changes since the last commit.
- `git diff HEAD:file1 file2`: Shows the difference between `file1` and `file2`.

Where to Go from Here?

- If you haven't ever used Git, start by using it locally and with Github.
- If you know the basics, start exploring branches. Learn about them and find a flow which works best for you.
- If you know most things about Git, just keep using it. Try and start remembering how to do certain things that you find yourself often Googleing for.

Become the person everyone asks for Git advice. It's used in the industry, so many companies will want to see knowledge of this tool.

Where to Go from Here?

- If you haven't ever used Git, start by using it locally and with Github.
- If you know the basics, start exploring branches. Learn about them and find a flow which works best for you.
- If you know most things about Git, just keep using it. Try and start remembering how to do certain things that you find yourself often Googleing for.

Become the person everyone asks for Git advice. It's used in the industry, so many companies will want to see knowledge of this tool.

Where to Go from Here?

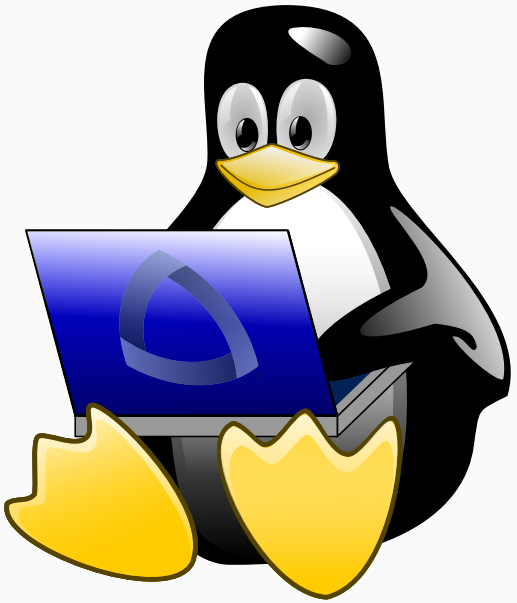
- If you haven't ever used Git, start by using it locally and with Github.
- If you know the basics, start exploring branches. Learn about them and find a flow which works best for you.
- If you know most things about Git, just keep using it. Try and start remembering how to do certain things that you find yourself often Googleing for.

Become the person everyone asks for Git advice. It's used in the industry, so many companies will want to see knowledge of this tool.

Copyright Notice

This presentation was from the **Mines Linux Users Group**. A mostly-complete archive of our presentations can be found online at `https://lug.mines.edu`.

Individual authors may have certain copyright or licensing restrictions on their presentations. Please be certain to contact the original author to obtain permission to reuse or distribute these slides.



Colorado School of Mines
Linux Users Group