Git Good

How to understand Git

Sumner Evans

14 February 2023

Mines ACM

A bit about me

My name is Sumner, I'm a **software engineer at Beeper**.

- I graduated from Colorado School of Mines in 2018 with my bachelor's in CS and 2019 with a master's in CS.
- I am an adjunct professor. Currently I'm teaching CSCI 341. I've taught 400, 406, and 564 in the past as well.
- I enjoy skiing, volleyball, and soccer.
- I'm a 4th degree black belt in ATA taekwondo.

Overview

1. Why use Git?

2. Understanding the Git data model

This talk is interactive!

If you have questions at any point, feel free to interrupt me.

Why use Git?

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 1

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.

Example Scenario 2

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.

Example Scenario 2

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.

Example Scenario 2

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.

Example Scenario 2

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.

Example Scenario 2

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.

Example Scenario 2

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.

Example Scenario 2

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.

Version Control Systems

Version Control Systems (VCSs) such as Git solve these problems.

- 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.

Version Control Systems

Version Control Systems (VCSs) such as Git solve these problems.

- 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.

Git is popular

Git is a very popular version control system.

Services such as GitHub and GitLab provide free hosting for Git repositories.

It has become the de-facto industry standard for source control.

Understanding the Git data model

Git is a distributed version control system

Distributed because you can use it without being connected to a central server. You have a full copy of the code on your own computer.

Version control because it keeps track of changes to files.

But how does it keep track of all of the changes?

Git is a distributed version control system

Distributed because you can use it without being connected to a central server. You have a full copy of the code on your own computer.

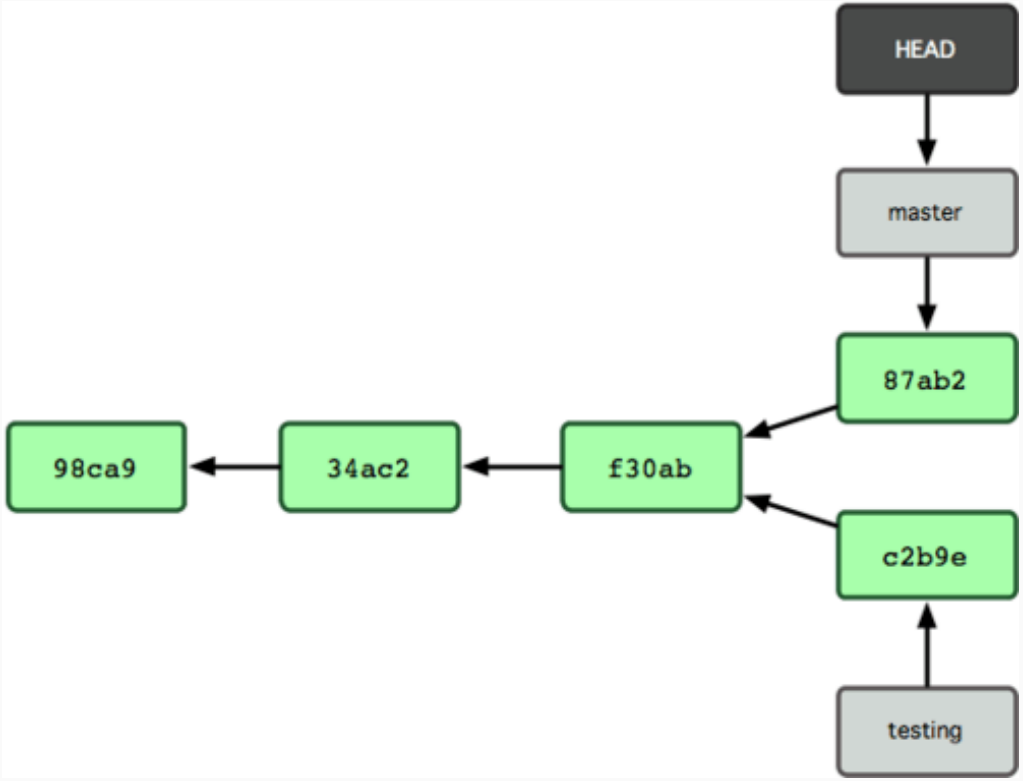
Version control because it keeps track of changes to files.

But how does it keep track of all of the changes?

Commits: what are they?

Commits are composed of two pieces of information: the previous commit(s) and a set of changes to be applied to those commits.[1]

Commits reference their parent(s).



[1] This is a bit of a lie, more on that later.

Commits: creating

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have "staged".

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: creating

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have "staged".

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: creating

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have "staged".

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: creating

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have "staged".

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run `git status` to show the list of files staged for commit.

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   git.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   git.pdf
        modified:   git.tex
```

The `git.tex` file has only some lines staged for commit.

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run `git status` to show the list of files staged for commit.

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   git.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   git.pdf
        modified:   git.tex
```

The `git.tex` file has only some lines staged for commit.

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run `git status` to show the list of files staged for commit.

```
> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   git.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   git.pdf
        modified:   git.tex
```

The `git.tex` file has only some lines staged for commit.

Commits: what will be committed, but with more detail?

To see the details of the changes that are staged (that is, will be committed), you can run `git diff --cached`.

If you want to see the details of the changes that are *not* staged, you can run `git diff`.

`git diff` optionally accepts a list of files to diff.

```
diff --git a/git.tex b/git.tex
index 2c01a7b..91148d1 100644
--- a/git.tex
+++ b/git.tex
@@ -33,9 +33,7 @@

 \section{Why use Git?}

-\begin{frame}{Why use Git? I}
-
-    Example Scenario:
+\begin{frame}{Example Scenario 1}

     \begin{enumerate}[<+->]
         \item You start a project called ``my-proj'' and write a ton of code.
```

Commits: what will be committed, but with more detail?

To see the details of the changes that are staged (that is, will be committed), you can run `git diff --cached`.

If you want to see the details of the changes that are *not* staged, you can run `git diff`.

`git diff` optionally accepts a list of files to diff.

```
diff --git a/git.tex b/git.tex
index 2c01a7b..91148d1 100644
--- a/git.tex
+++ b/git.tex
@@ -33,9 +33,7 @@

 \section{Why use Git?}

-\begin{frame}{Why use Git? I}
-
-    Example Scenario:
+\begin{frame}{Example Scenario 1}

     \begin{enumerate}[<+->]
         \item You start a project called ``my-proj'' and write a ton of code.
```

Commits: summary

Use `git add` and `git reset` (or variants) to stage/unstage changes for your.

Use `git commit` to create a commit from the currently staged changes (which you can see by running `git diff --cached`).

Commits: summary

Use `git add` and `git reset` (or variants) to stage/unstage changes for your.

Use `git commit` to create a commit from the currently staged changes (which you can see by running `git diff --cached`).

Branches

ohea

Advanced Topics

Gitignore

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.

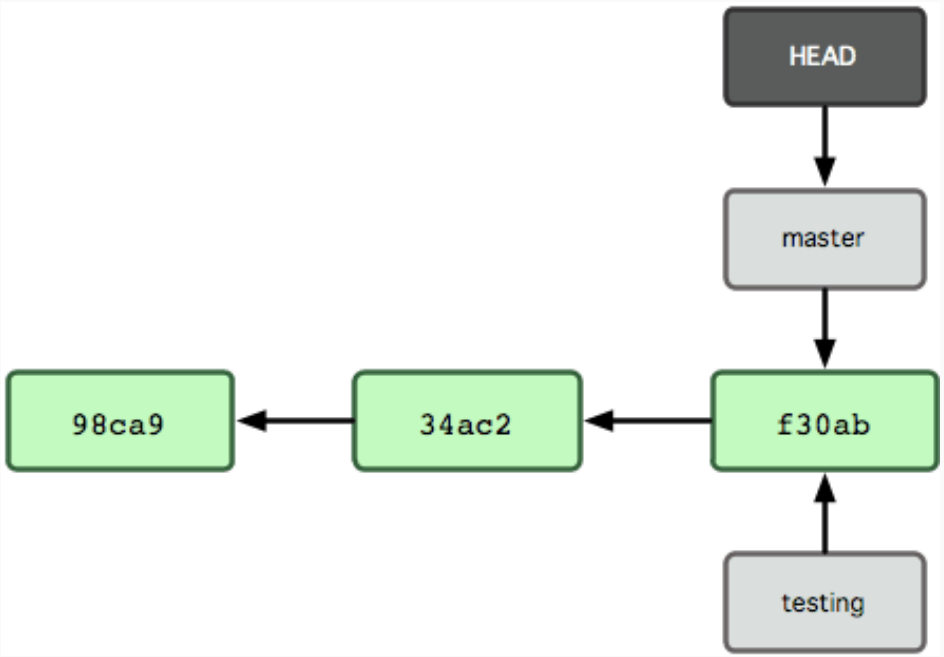
Branches I

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[2]II

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



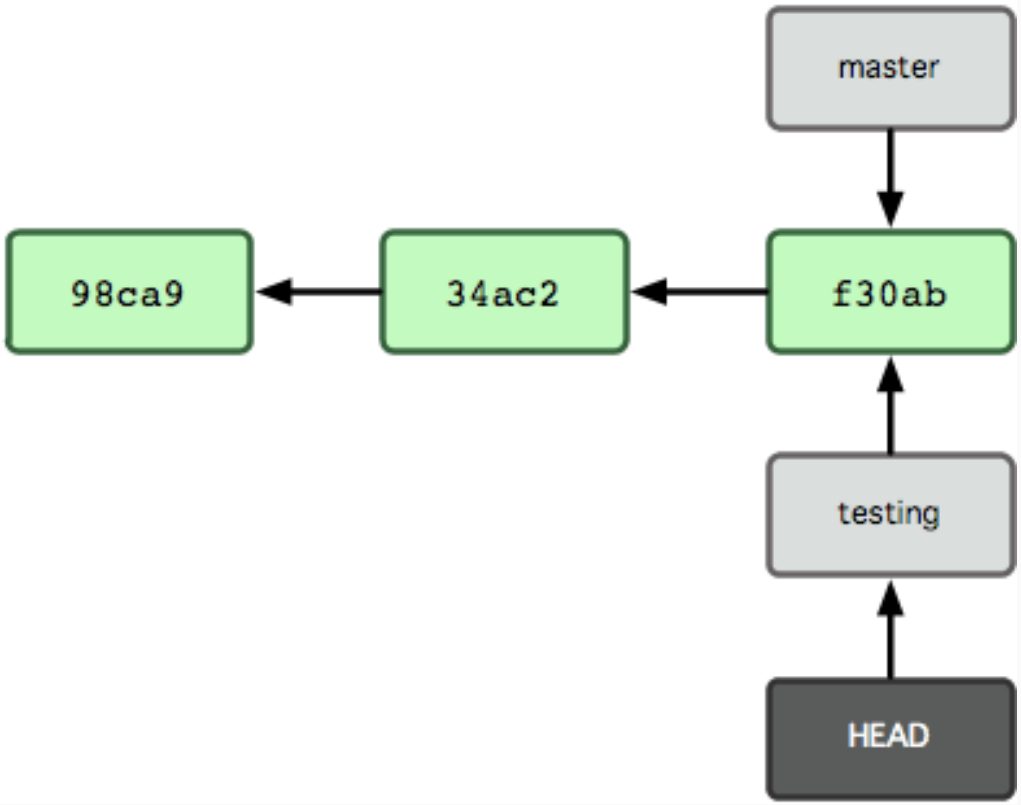
Note, `HEAD` is pointer to the current branch.

[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`

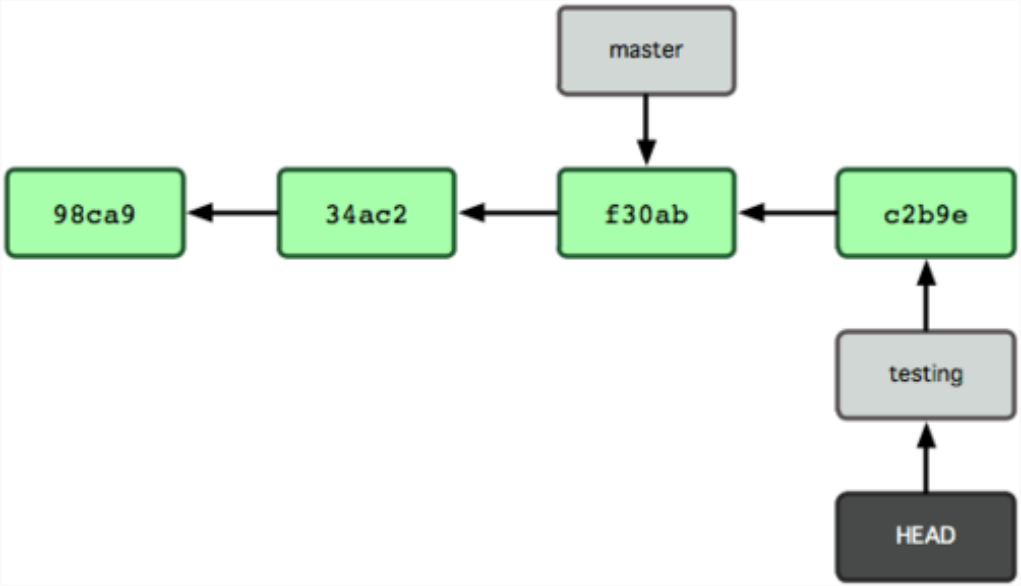
Branches III

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



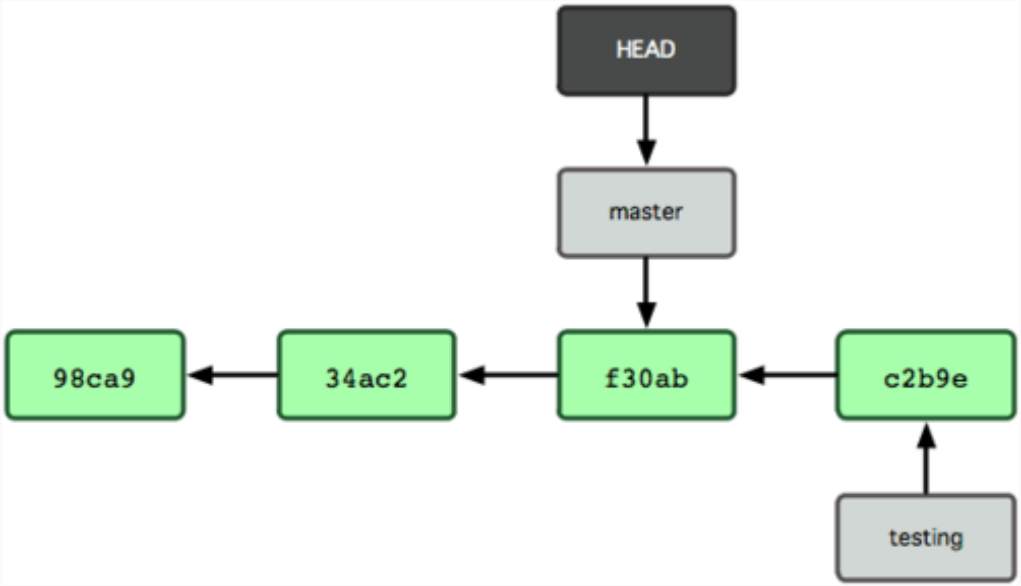
Branches IV

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



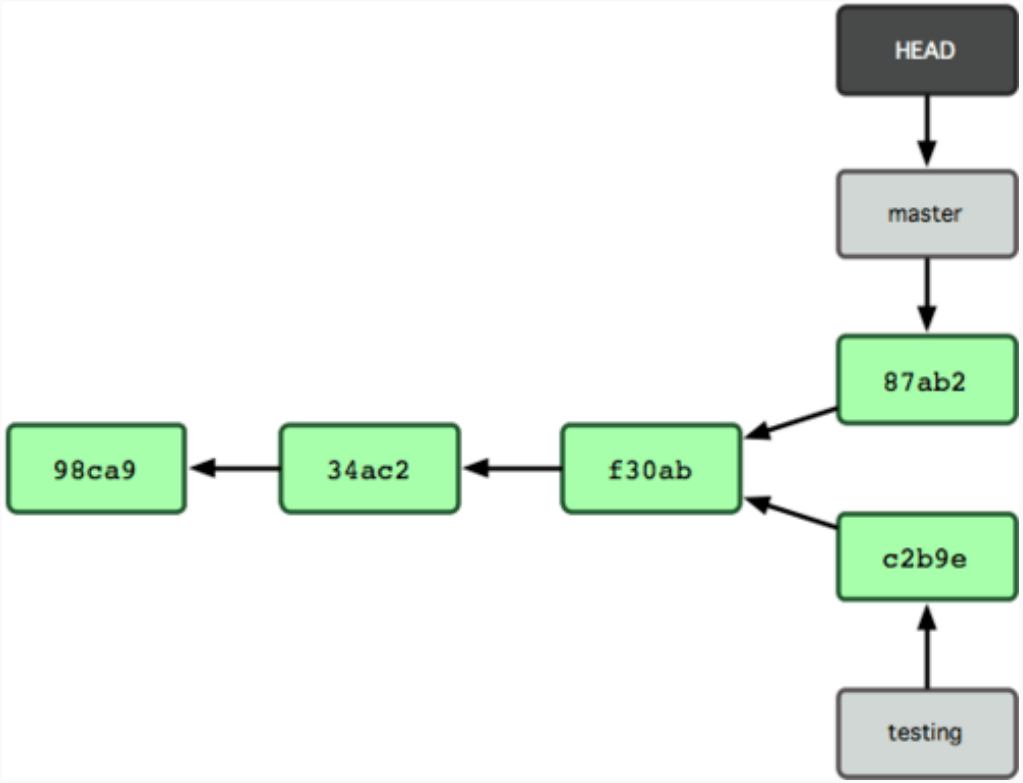
Branches V

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[3]:

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.