Branching

Aside from making commits, VCS enables branching.
Remember, the basic structure is a tree

Got a new feature? Just make a branch to see if it will work.
git checkout -b newbranch

Of course, branching is pointless unless you can merge your changes back to your master branch. If only....

OMG Merging

Two basic kinds of merge, "fast-forward" and "three-way merge".
Branches are kept in git as changes from the master branch.

Example

Example, Cont'd

What does a git workflow look like?

- Create a folder for your project
  - Optionally put some already written code in it
- Use git to initialize the .git directory.
- Work on some files.
- Add files to a commit
- Create a commit of those files
  - With a message explaining the changes made since the last commit.
- Optionally, pull changes from another git repository such as Github, Bitbucket, Gitorious, or your own and then push your changes to that repo.
- Go back to "Work on some files."

Actually working with git

Playing with others

Git works without a server, and you can simply copy your directory (which contains a .git directory) and give it to someone else.

But you can also have a git server (such as the ones I already mentioned) on a remote host. Most of those git hosts work over SSH, so that your files are transmitted securely.

Working with hosts is as simple as adding "git push" once you're done with committing.

For the love of all that you hold dear, do this. All the version control in the world does nothing if your hard drive crashes with the only copy of your repo on it.

Intro to Git

Sane Version Control

More cool commands

- git diff ▪ shows the difference between various files and even trees
- git archive --output HEAD.tar.gz HEAD ▪ git archive --output foo.zip HEAD
- git rm filename ▪ git mv filename ▪ use this to actually delete from the repo. Cleaner than just deleting through the filesystem.
- git checkout foo.cpp ▪ Checkout an individual file from the HEAD. This is great if you made a bunch of changes to different files, but want one file in particular back (or if you deleted that file on accident via the filesystem!)
- git log ▪ to show the history of commits.
- .gitignore ▪ not a command but a file. Put filenames for regular expressions that map to filenames in here) and git will never add them to the repo or commits. This is useful for temp files, binary files, or files with sensitive data in them.
- git reset HEAD filename ▪ removes that file from the next commit
- git branch ▪ lists all local branches

How does it work?

Git is a kind of database, stored in the .git directory.
Trees, Files aka "blobs", Commits are stored as Objects in this database.

Git tracks only the changes in trees of files.
A set of changes is referred to as a "commit".

When you make a commit, git generates a SHA1 of each file and folder starting with the root directory of your project.
Finally the SHA1's are all SHA1'd and the commit itself is given the result. This SHA1 uniquely identifies this commit over the lifetime of the project.

In fact, SHA1 Hashes are unique enough that you typically can uniquely identify a commit by only the first 7-8 characters: "Commit dec0f710"

Wait, what's a commit?

About me:

- The name's Gager
- Homeostatis Mallia, Davus del Sur
- Father of two kids
- Gets copy / pasta from google
- Three tag team

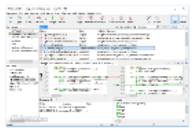
Git Client

- GitKraken
- Git-cola
- Giggle
- Gitg

Git Client

- SmartGit

is a also a cross-platform, powerful, popular GUI Git client for Linux, Mac OS X and Windows.



Intro to Git

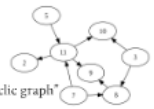
Sane Version Control

What is Git?

Git is a distributed VCS developed by Linus Torvalds (creator of the Linux kernel)

Distributed means there is no central server required.

- Work can be done "offline" simultaneously with someone else working offline.
- This is different! Traditional VCS were highly centralized.
  - In some cases, only one person could check out (edit) a file at a time. In others, two people could checkout at the same time, but whoever saved last "wins"



Where other VCS are trees, Git is technically a "directed acyclic graph"

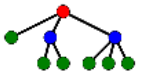
What is Version Control?

Version Control (Source Control, Revision Control, etc) is a means for managing your source code.

Version Control lets you:

- Access previous versions of your code
- Have multiple working copies of your code
- Make backups trivial
- Share your work with other people easily

Version Control Systems (VCS) are typically represented as a tree data structure.



About me:

- The name's Goper
- Hometown Malita, Davao del Sur
- Father of two kids
- Can copy / paste from google
- I love vaginant

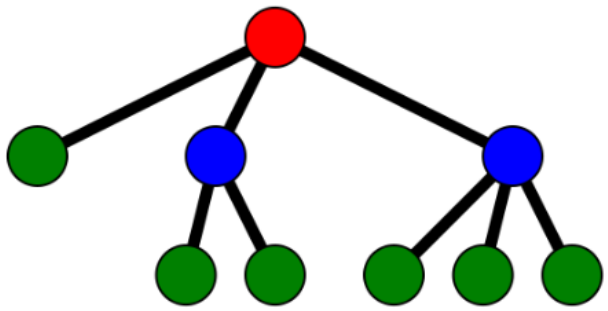
What is Version Control?

Version Control (Source Control, Revision Control, etc) is a means for managing your source code.

Version Control lets you:

- Access previous versions of your code
- Have multiple working copies of your code
- Make backups trivial
- Share your work with other people easily

Version Control Systems (VCS) are typically represented as a tree data structure.



Intro to Git

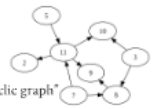
Sane Version Control

What is Git?

Git is a distributed VCS developed by Linus Torvalds (creator of the Linux kernel)

Distributed means there is no central server required.

- Work can be done "offline" simultaneously with someone else working offline.
- This is different! Traditional VCS were highly centralized.
  - In some cases, only one person could check out (edit) a file at a time. In others, two people could checkout at the same time, but whoever saved last "wins"



Where other VCS are trees, Git is technically a "directed acyclic graph"

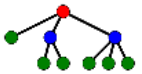
What is Version Control?

Version Control (Source Control, Revision Control, etc) is a means for managing your source code.

Version Control lets you:

- Access previous versions of your code
- Have multiple working copies of your code
- Make backups trivial
- Share your work with other people easily

Version Control Systems (VCS) are typically represented as a tree data structure.

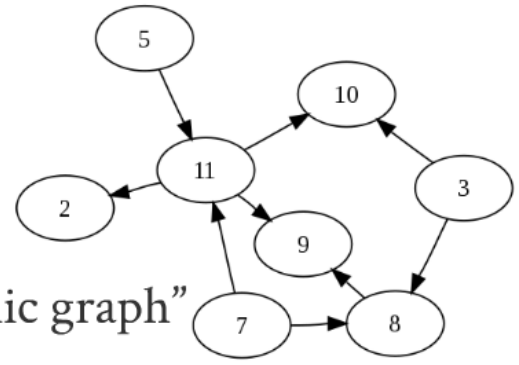


What is Git?

Git is a distributed VCS developed by Linus Torvalds (creator of the Linux kernel)

Distributed means there is no central server required.

- Work can be done "offline" simultaneously with someone else working offline.
- This is different! Traditional VCS were highly centralized.
    - In some cases, only one person could check out (edit) a file at a time. In others, two people could checkout at the same time, but whoever saved last "wins"



Where other VCS are trees, Git is technically a “directed acyclic graph”

How does it work?

Git is a kind of database, stored in the .git directory.

Trees, Files aka "blobs", Commits are stored as Objects in this database.

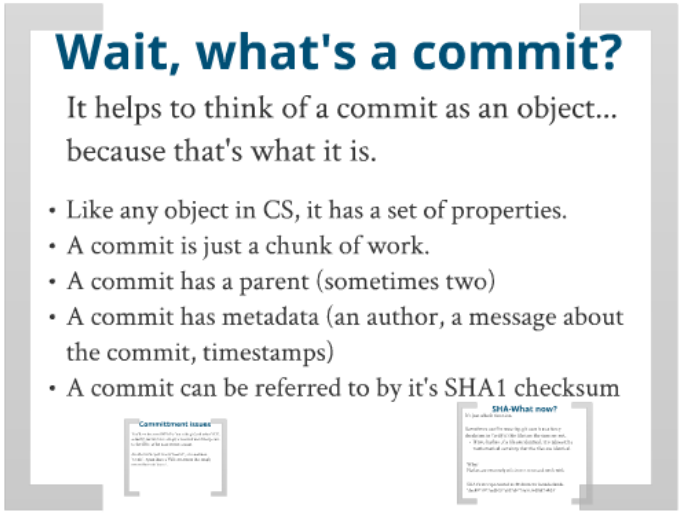
Git tracks only the changes in trees of files.

A set of changes is referred to as a "commit".

When you make a commit, git generates a SHA1 of each file and folder starting with the root directory of your project.

Finally the SHA1's are all SHA1'd and the commit itself is given the result. This SHA1 uniquely identifies this commit over the lifetime of the project.

In fact, SHA1 Hashes are unique enough that you typically can uniquely identify a commit by only the first 7-8 characters: "Commit dec89719"



How does it work?

Git is a kind of database, stored in the .git directory.

Trees, Files aka "blobs", Commits are stored as Objects in this database.

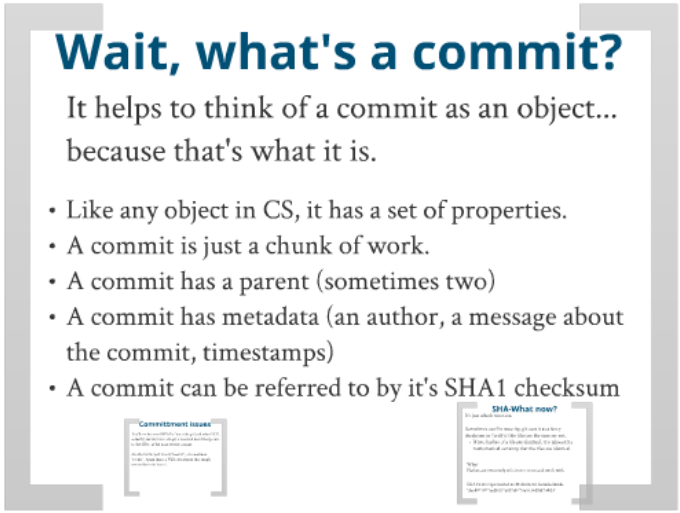
Git tracks only the changes in trees of files.

A set of changes is referred to as a "commit".

When you make a commit, git generates a SHA1 of each file and folder starting with the root directory of your project.

Finally the SHA1's are all SHA1'd and the commit itself is given the result. This SHA1 uniquely identifies this commit over the lifetime of the project.

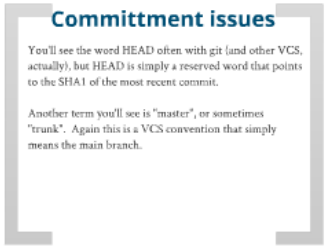
In fact, SHA1 Hashes are unique enough that you typically can uniquely identify a commit by only the first 7-8 characters: "Commit dec89719"

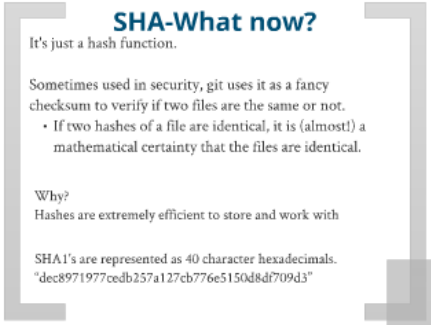


Wait, what's a commit?

It helps to think of a commit as an object... because that's what it is.

- Like any object in CS, it has a set of properties.
- A commit is just a chunk of work.
- A commit has a parent (sometimes two)
- A commit has metadata (an author, a message about the commit, timestamps)
- A commit can be referred to by it's SHA1 checksum





SHA-What now?

It's just a hash function.

Sometimes used in security, git uses it as a fancy checksum to verify if two files are the same or not.

- If two hashes of a file are identical, it is (almost!) a mathematical certainty that the files are identical.

Why?
Hashes are extremely efficient to store and work with

SHA1's are represented as 40 character hexadecimals.
“dec8971977cedb257a127cb776e5150d8df709d3”

Committment issues

You'll see the word HEAD often with git (and other VCS, actually), but HEAD is simply a reserved word that points to the SHA1 of the most recent commit.

Another term you'll see is "master", or sometimes "trunk". Again this is a VCS convention that simply means the main branch.

Git isn't written with binary files (images, audio, executables) in mind.

Don't store these in git if they change frequently, or else your repository will get huge and people will say mean things about you.

How does it work?

Git is a kind of database, stored in the .git directory.

Trees, Files aka "blobs", Commits are stored as Objects in this database.

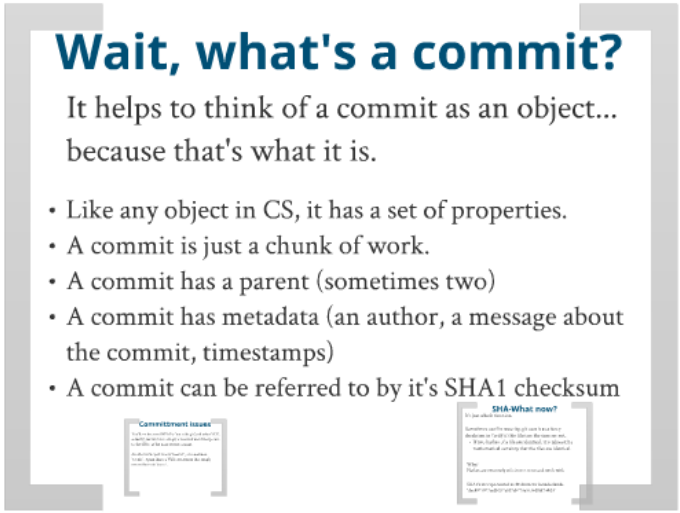
Git tracks only the changes in trees of files.

A set of changes is referred to as a "commit".

When you make a commit, git generates a SHA1 of each file and folder starting with the root directory of your project.

Finally the SHA1's are all SHA1'd and the commit itself is given the result. This SHA1 uniquely identifies this commit over the lifetime of the project.

In fact, SHA1 Hashes are unique enough that you typically can uniquely identify a commit by only the first 7-8 characters: "Commit dec89719"



What does a git workflow look like?

- Create a folder for your project
  - Optionally put some already written code in it
- Use git to initialize the .git directory.
- Work on some files.
- Add files to a commit
- Create a commit of those files
  - With a message explaining the changes made since the last commit.
- Optionally, pull changes from another git repository such as Github, Bitbucket, Gitorious, or your own and then push your changes to that repo.
- Go back to "Work on some files."

Actually working with git

- git init
- git add -A [add all files in this directory.]
- git commit -m “This is my first check in.”
- [Edit some files.]
- git status [check and see which files you've worked on since your last commit.]
- [Go to step 2.]

Actually working with git

- git init
- git add -A [add all files in this directory.]
- git commit -m “This is my first check in.”
- [Edit some files.]
- git status [check and see which files you've worked on since your last commit.]
- [Go to step 2.]

Branching

Aside from making commits, VCS enables branching.

Remember, the basic structure is a tree

Got a new feature? Just make a branch to see if it will work.

git checkout -b newbranch

Of course, branching is pointless unless you can merge your changes back to your master branch. If only....

Branch like its going out of style.

git checkout -b newbranch
"newbranch" now has the same SHA1 as the previous branch (say "master").
However HEAD now points to newbranch, and new commits apply to newbranch, and not master.
You can switch back with:
git checkout master [Bam, you're back to your original set of files]
You can even have multiple branches:
git checkout -b branchitymcbranch master
will start a new branch at whatever the last commit of master is.

Git happens to make branching very easy compared to other VCS

- managing (and moving between) branches is easy
- requires no special permissions to create or delete a branch
- effectively does not lead to duplicate files (git only stores one file per SHA1)
- merging them back later is relatively easy (we'll get to that)

Git happens to make branching very easy compared to other VCS

- managing (and moving between) branches is easy
- requires no special permissions to create or delete a branch
- effectively does not lead to duplicate files (git only stores one file per SHA1)
- merging them back later is relatively easy (we'll get to that)

Branch like its going out of style.

git checkout -b newbranch

"newbranch" now has the same SHA1 as the previous branch (say “master”).

However HEAD now points to newbranch, and new commits apply to newbranch, and not master.

You can switch back with:

git checkout master [Bam, you're back to your original set of files]

You can even have multiple branches:

git checkout -b branchitymcbranch master

will start a new branch at whatever the last commit of master is.

Branching

Aside from making commits, VCS enables branching.

Remember, the basic structure is a tree

Got a new feature? Just make a branch to see if it will work.

git checkout -b newbranch

Of course, branching is pointless unless you can merge your changes back to your master branch. If only....

Branch like its going out of style.

git checkout -b newbranch
"newbranch" now has the same SHA1 as the previous branch (say "master").
However HEAD now points to newbranch, and new commits apply to newbranch, and not master.
You can switch back with:
git checkout master [Bam, you're back to your original set of files]
You can even have multiple branches:
git checkout -b branchitymcbranch master
will start a new branch at whatever the last commit of master is.

Git happens to make branching very easy compared to other VCS

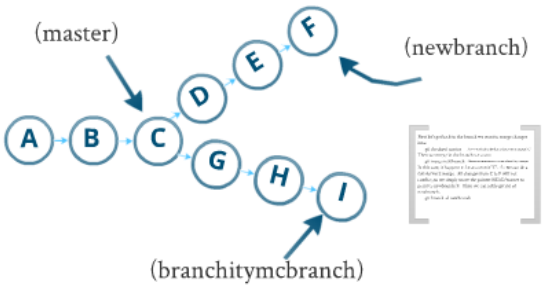
- managing (and moving between) branches is easy
- requires no special permissions to create or delete a branch
- effectively does not lead to duplicate files (git only stores one file per SHA1)
- merging them back later is relatively easy (we'll get to that)

OMG Merging

Two basic kinds of merge, "fast-forward" and "three-way merge".
Branches are kept in git as changes from the master branch.

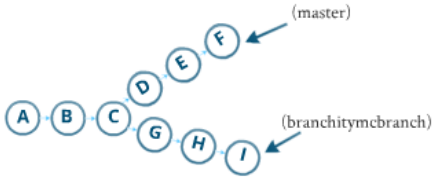
Example

The circles are commits. Notice that we branched at commit C, twice. How do we merge this?



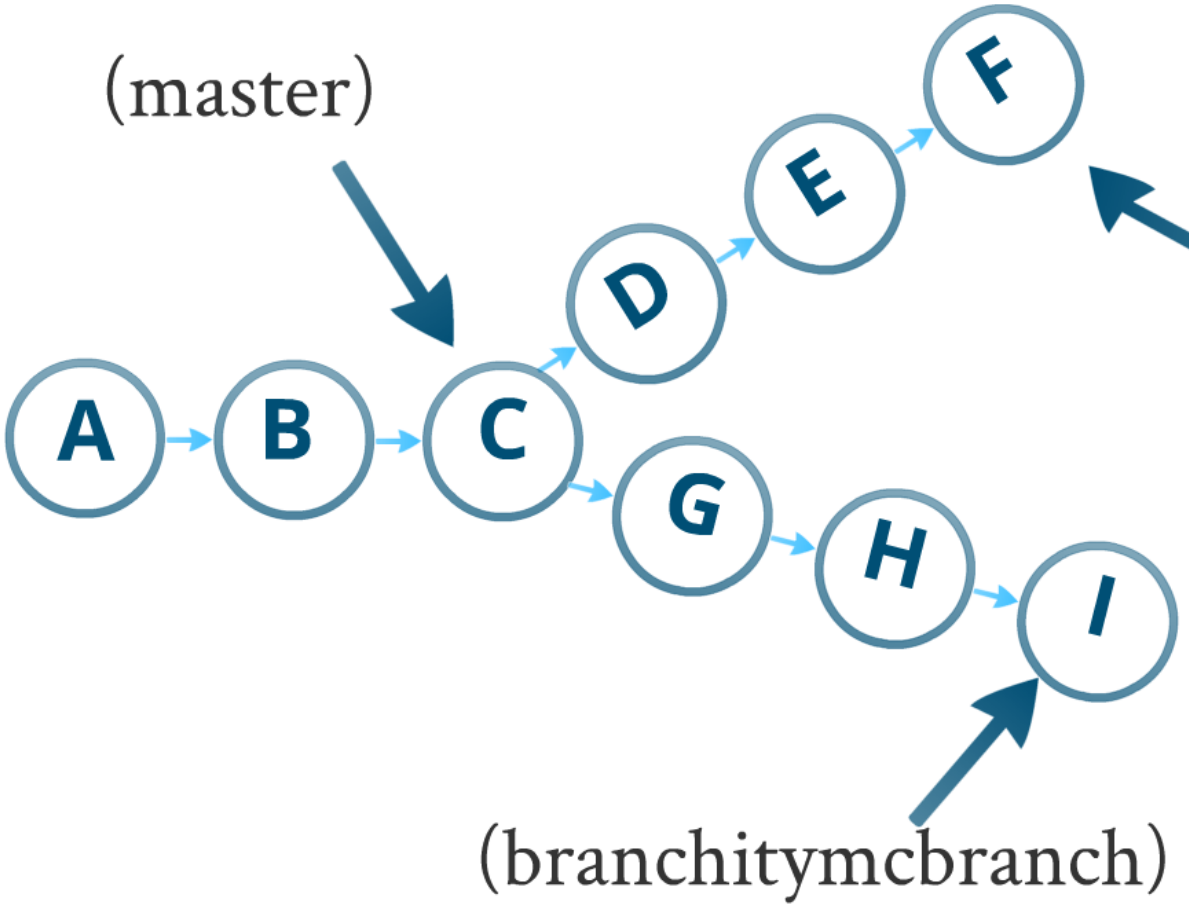
Example, Cont'd

This is where it gets harder. How do we merge this?



Example

The circles are commits. Notice that we branched at commit C, twice. How do we merge this?



First let's get back to the branch we want to merge changes into:

git checkout master Now we've back to the files as they were at commit "C"

Then we merge in the branch we want:

git merge newbranch Checks common ancestry to see where they diverged

In this case, it happens to be at commit "C". So we can do a fast-forward merge. All changes from C to F will not conflict, so we simply move the pointer HEAD/master to point to newbranch/F. Then we can safely get rid of newbranch.

git branch -d newbranch

First let's get back to the branch we want to merge changes into:

git checkout master Now we're back to the files as they were at commit "C"

Then we merge in the branch we want:

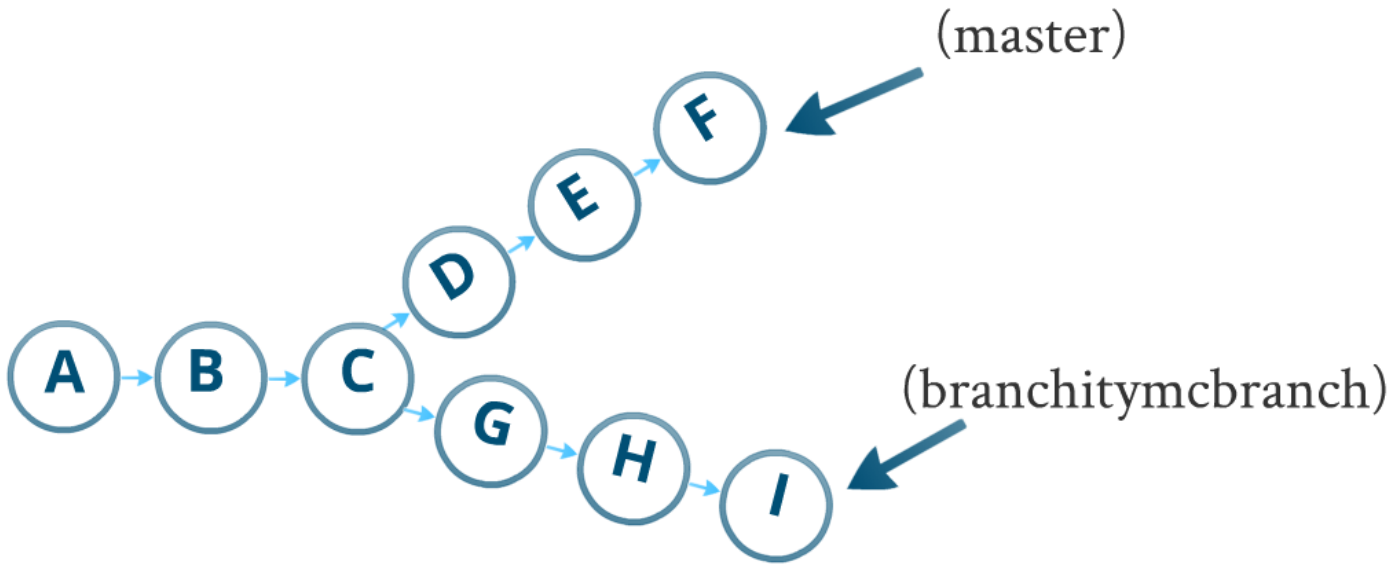
git merge newbranch Checks common ancestry to see where they diverged

In this case, it happens to be at commit "C". So we can do a fast-forward merge. All changes from C to F will not conflict, so we simply move the pointer HEAD/master to point to newbranch/F. Then we can safely get rid of newbranch.

git branch -d newbranch

Example, Cont'd

This is where it gets harder. How do we merge this?



Merging the ugly stuff

The commands are the same:

```
git checkout master
git merge branchitymcbranch
```

This is called a three-way merge. branchitymcbranch and master share the common ancestor of C. We can't simply move the pointer this time, though.

In some cases, this merge can still be done automatically, but often two commits will have conflicts. At that point it is up to you to resolve (git will prompt you with the conflicts)

Merging the ugly stuff

Our history is no longer linear, however, and this isn't nice.

Instead we could do something like:

```
git checkout newbranch; git rebase master
```

This applies G H I on top of A B C D E F.

Why bother? Linear history makes working with certain tools in the git toolchain (not to mention your brain) easier.

Merging the ugly stuff

The commands are the same:

```
git checkout master
git merge branchitymcbranch
```

This is called a three-way merge. branchitymcbranch and master share the common ancestor of C. We can't simply move the pointer this time, though.

In some cases, this merge can still be done automatically, but often two commits will have conflicts. At that point it is up to you to resolve (git will prompt you with the conflicts)

Merging the ugly stuff

Our history is no longer linear, however, and this isn't nice.

Instead we could do something like:

    git checkout newbranch; git rebase master

This applies G H I on top of A B C D E F.

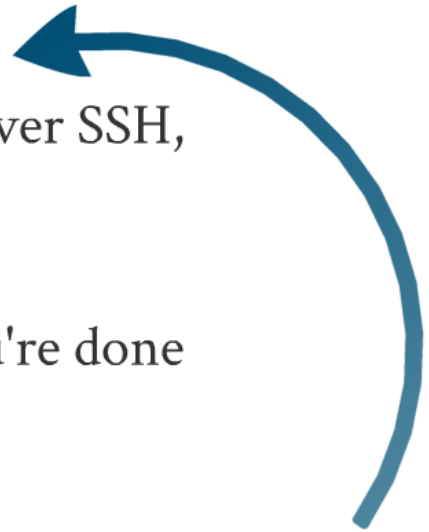
Why bother? Linear history makes working with certain tools in the git toolchain (not to mention your brain) easier.

Playing with others

Git works without a server, and you can simply copy your directory (which contains a .git directory) and give it to someone else.

But you can also have a git server (such as the ones I already mentioned) on a remote host. Most of these git hosts work over SSH, so that your files are transmitted securely.

Working with hosts is as simple as adding "git push" once you're done with committing.



For the love of all that you hold dear, do this. All the version control in the world does nothing if your hard drive crashes with the only copy of your repo on it.

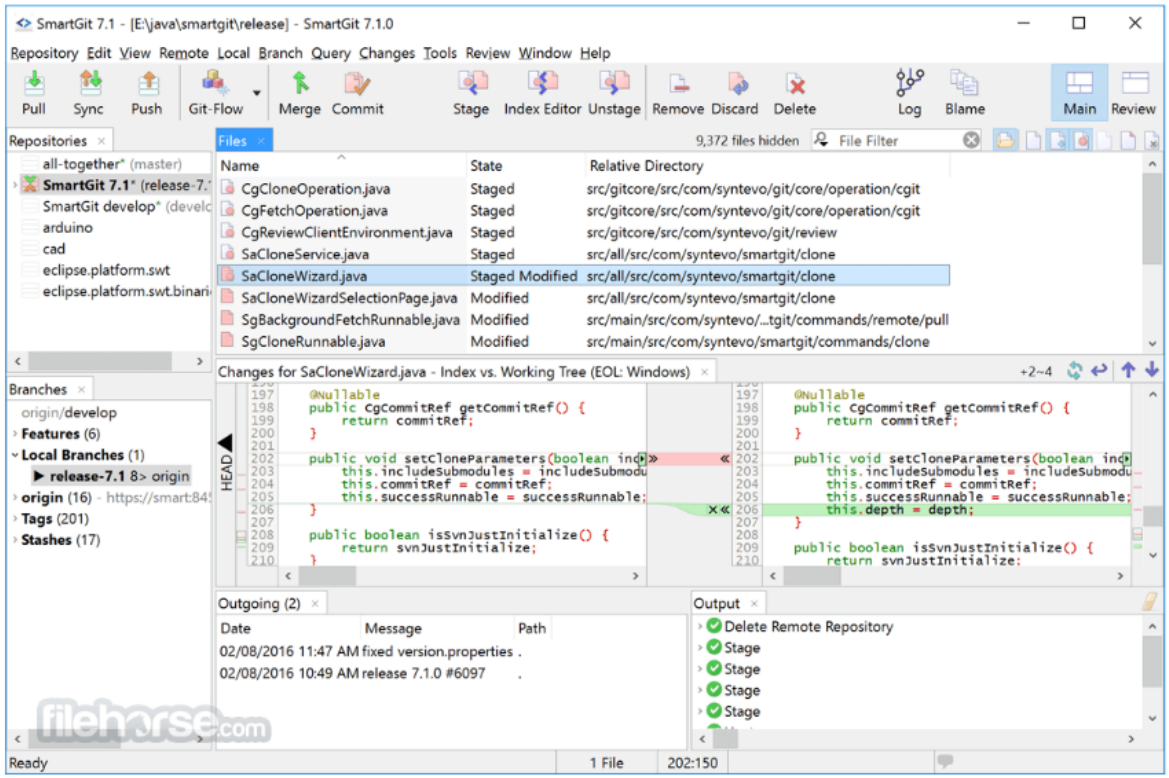
More cool commands

- git diff # show the difference between various files and even trees
- git archive --output foo.tar HEAD (or git archive --output foo.zip HEAD)
- git rm filename (or git mv filename) # use this to actually delete from the repo. Cleaner than just deleting through the filesystem.
- git checkout foo.cpp # Checkout an individual file from the HEAD. This is great if you made a bunch of changes to different files, but want one file in particular back (or if you deleted that file on accident via the filesystem!)
- git log # to show the history of commits.
- .gitignore # not a command but a file. Put filenames (or regular expressions that map to filenames in here) and git will never add them to the repo or commits. This is useful for temp files, binary files, or files with sensitive data in them.
- git reset HEAD filename # removes that file from the next commit
- git branch # lists all local branches

Git Client

- SmartGit

is a also a cross-platform, powerful, popular GUI Git client for Linux, Mac OS X and Windows.



Git Client

- GitKraken
- Git-cola
- Giggle
- Gitg