Dr. Christopher S. Simmons (csim@ices.utexas.edu)

September 22, 2015

SSH, File Transfers and Version Control

CSE 380: Tools and Techniques of Computational Chemistry

SSH—Secure Shell

- Replaced telnet, rlogin, rcp, etc…
- Originated from an academic project in Finland
- Commercialized 1998: SSH Communications Security
- SSH1 expired in May 1995 because of security flaws
- SSH2 is currently used
- OpenSSH: most popular F(L)OSS implementation

SSH services/functionality

- Services
  - Confidentiality
  - Integrity
  - Authentication
- Functionality
  - Secure command shell (remote execution of commands)
  - Secure file transfer
  - Data tunneling for TCP/IP applications (TCP port forwarding)

SSH protocol suite (architecture)

- SSH transport layer protocol (TLP)
  - (server) authentication, confidentiality, and integrity
- SSH user authentication protocol (UAP)
- SSH connection protocol (CP)
  - Provides interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X11 connections.

SFTP (secure file transport protocol)

- runs over a secure channel
- Independent from other SSH protocols
  - may run on top of other secure protocols, e.g., TLS
- ICES/TACC uses SFTP on top of SSH
- Insert DEMO of sftp here

User authentication

- Protocol Basics
  - Client is a program running on behalf of the user
  - The server has complete control over authentication as it tells client which authentication methods can be used.
  - The client can choose the order making it flexible.
- Authentication methods:
  - Public key based
  - Password based
  - Host based

User Authentication cont.

- Password based authentication is what we've been using
- Host based authentication requires escalated privileges
- Public key based is most secure and has advantages over other methods

Generating a private/public key pair

- OpenSSH supports DSA and RSA keys
- Which to use?
  - RSA can be 768 – 4096 bits (default is 2048)
  - DSA must be 1024 bit
  - DSA is slightly faster than default RSA (2048)
  - RSA 2048 and DSA 1024 are both sufficient
  - DSA used to be preferred before RSA patent expired
- Use either, it doesn't matter

ssh-keygen

- Generate a SSH key on your client machine

[17:32:48][csim@hbar:~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/h1/csim/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /h1/csim/.ssh/id_rsa
Your public key has been saved in /h1/csim/.ssh/id_rsa.pub
The key fingerprint is:

43:b6:ad:61:1b:7f:8c:03:db:17:9d:bf:be:13:11:ec

csim@hbar.ices.utexas.edu

- Copy the public key to ~/.ssh/authorized_keys on the server
- Now you can log in using your passphrase

Automating remote logins

- ssh-agent – holds private keys and their passphrase to provide automatic SSH authentication
- ssh-add – adds a private key to a running ssh-agent
- ssh-agent can be automatically launched at login
- Demo

SSH tricks

- X11 forwarding: ssh -X host
- Remote command execution: ssh host [cmd] [arg]
- ssh-copy-id
- Speeding Things Up: Compression and Ciphers
  - ssh -C
  - ssh -c blowfish
- Piping through SSH
  - cat myfile | ssh user@host lpr
  - tar – cf source_dir | ssh host 'cat > dest.tar'
- authentication agent forwarding: ssh -A
- SSH clients for iPhone and Android systems

SCP – secure copy

- Local file to remote server
  - scp filename.txt hostname:/path/to/filename/filename.txt
- Remote server to local file
  - scp hostname:filename.txt .
- Recursive copies on directories also work
  - scp –r hbar.ices.utexas.edu:public_html ~/tmp

rsync - a fast, versatile, remote (and local) file-copying tool

- rsync options source destination
- Example 1. Synchronize Two Directories in a Local Server
  - $ rsync -zvr /path/to/source /path/to/copy
    - z – enable compression
    - v – verbose
    - r – recursive
- Example 2. Preserve timestamps during Sync using rsync –a
  - rsync -azv /path/to/source /path/to/copy
  - -a enters archive mode
  - Archive mode does the following:
    - Recursive mode
    - Preserves symbolic links
    - Preserves permissions
    - Preserves timestamp
    - Preserves owner and group

More rsync foo

```
[18:49:36][csim@gedanken:~]$ rsync -avz --progress ~/results hbar.ices.utexas.edu:/workspace/csim
building file list ...
5 files to consider
results/
results/betaN_kde.eps
      98763 100%    6.29MB/s    0:00:00 (xfer#1, to-check=3/5)
results/rhoch_kde.eps
      58378 100%    3.27MB/s    0:00:00 (xfer#2, to-check=2/5)
results/rhov_kde.eps
      98014 100%    3.46MB/s    0:00:00 (xfer#3, to-check=1/5)
results/unified.pdf
     305581 100%    5.95MB/s    0:00:00 (xfer#4, to-check=0/5)

sent 369183 bytes  received 114 bytes  43446.71 bytes/sec
total size is 560736  speedup is 1.52
```

Other useful rsync options

- -u skip files that are newer on the receiver
- -L transform symlink into referent file/dir
- -n, show what would have been transferred
- -x, don't cross filesystem boundaries
- -E copy extended attributes, resource forks
- -q suppress non-error messages
- --include=PATTERN include files matching PATTERN
- --exclude=PATTERN exclude files matching PATTERN
- --bwlimit=KBPS limit I/O bandwidth

wget - The non-interactive network downloader

- wget allows us to download files from the web via the command line
- It supports http, https and ftp
- Coupling wget with your ICES ~/public_html can be very powerful
- wget http://users.ices.utexas.edu/~csim/file.txt
- wget –r –l1 http://users.ices.utexas.edu/~csim/ttcs

The Multiple Roles of a Computational Scientist

- Computational Scientists are expected to be:
  - Physicist – addressed in curriculum
  - Numerical Analyst – address in curriculum
    - Software Developer? – seriously?
- Software Engineering (and creating . . .) is vital
  - understandable code

Software Engineering is not part of the formal curriculum thus most scientific software developed in academia is unverified. Furthermore, the software developed is unable to ever be verified without significant time investment.

Version Control

- Minimum Guidelines — Actually using version control is the first step
- Ideal Usage
  - Put EVERYTHING under version control
  - Consider putting parts of your home directory under VC
  - Use a consistent project structure and naming convention
  - Commit often and in logical chunks
  - Write meaningful commit messages
  - Do all file operations in the VCS
  - Set up change notifications if working with multiple people

Source Control and Versioning

- Why bother?
- Codes evolve over time
  - sometimes bugs creep in (by you or others)
  - sometimes the old way was right
  - sometimes it's nice to look back at the evolution
- How can you get back to an old version?
  - keep a copy of every version of every file
    - disk is cheap, but this could get out of hand quickly
    - is a huge pain to maintain
  - use a tool (we will focus on one in particular)

Token Soap Box Slide

- The merits of organized, source code control should be obvious
- It really could save your marriage/life one day (ok, maybe your diploma or research grant)
- You should use a src code revision tool regardless of the size or complexity of your coding efforts

Some Example Tools

- Free (who doesn't love free?)
  - RCS – Revision Control System
  - CVS – Concurrent Versions System
  - SVN – Subversion
  - git (used by linux kernel community, distributed)
- Commercial
  - MS Visual Studio Team System
  - IBM Rational Software: Clearcase
  - AccuRev
  - MKS Integrity

Two Models of Centralized Source Control

- A central repository holds the files in both of these models (eg. RCS, CVS, Subversion)
  - this means a specific computer is required with some disk space
  - it should be backed up! (you have been warned)

- Read-only Local Workspaces and Locks
  - every developer has a read-only local copy of the source files
  - individual files are checked-out as needed and locked in the repo in order to gain write access
  - unlocking the file commits the changes to the repo and makes the file read-only again

- Read/Write Local Workspaces and Merging
  - every developer has a local copy of the source files
  - everybody can read and write files in their local copy
  - conflicts between simultaneous edits handled with merging algorithms or manually when files are synced against the repository or committed to it
  - CVS and Subversion behave this way

CVS - Concurrent Versions System

- Started with some shell scripts in 1986
- Recoded in ’89
- Evolving ever since (though mostly unchanging now)
- Uses r/w local workspaces and merging
- Only stores differences between versions
  - saves space
  - basically uses diff(1) and diff3(1)
- Works with local repositories or over the network with rsh/ssh

Subversion

- Subversion is a functional superset of CVS (so if you learned cvs previously, you can also function in subversion)
  - Began initial development in 2000 as a replacement for CVS (initial version in Aug. 2001)
  - Also interacts with local copies
  - Includes directory versioning (rename and moves)
  - Truly atomic commits (ie. interrupted commit operations do not cause repository inconsistency or corruption)
  - File meta-data
  - True client-server model
  - Cross-platform, open-source

Subversion Architecture

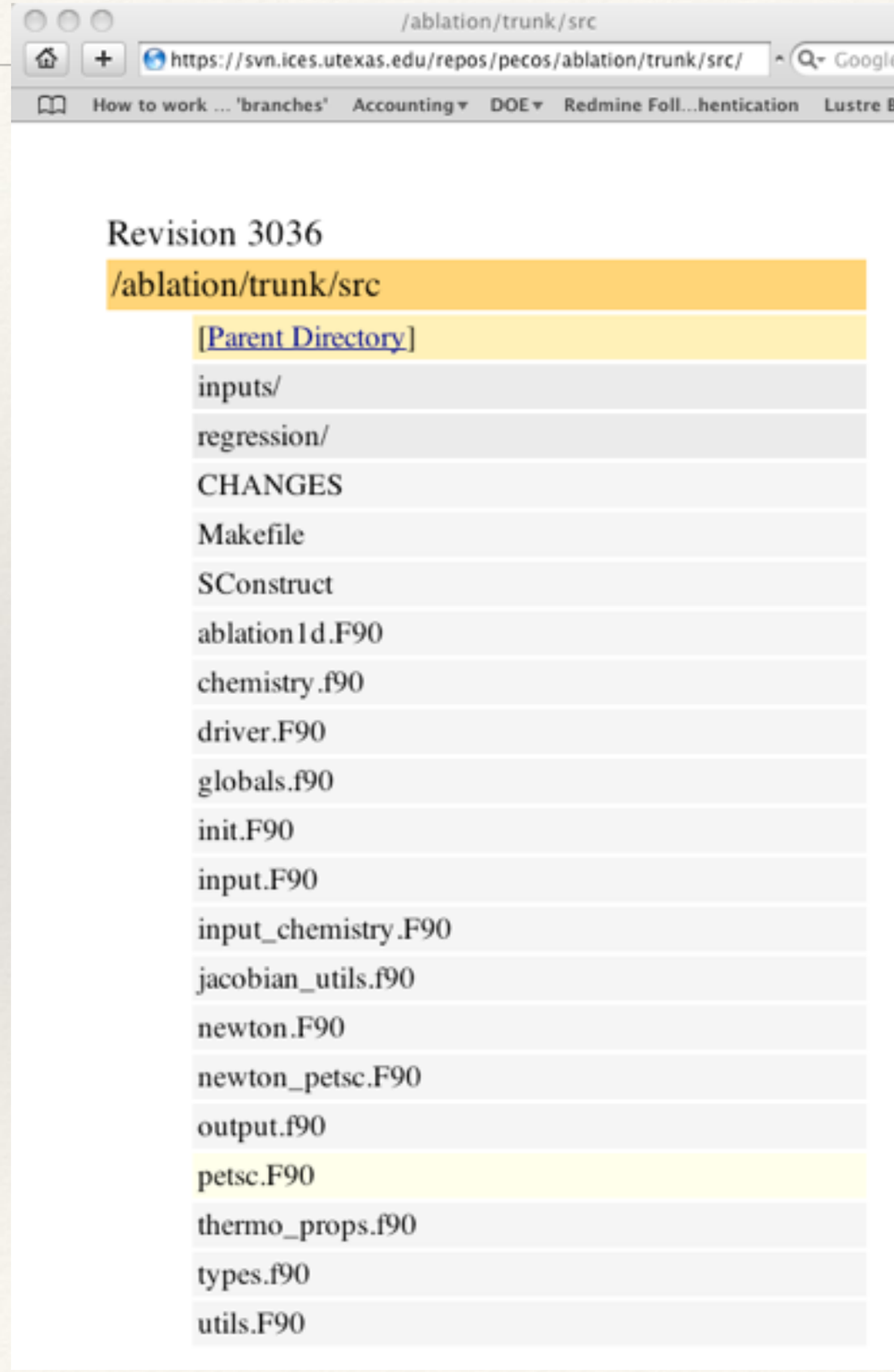
- Each working copy has a .svn directory
  - Similar to the CVS's CVS directory
  - Stores metadata about each file
- A local or network master repository is defined
- Network access over HTTP (https) or SSH
  - Apache often used for web server
- Encrypted authentication
- Has support for binary files
- Command line client is svn (this is what you use all the time)

Subversion Architecture

- Since central repository is generally accessed via http, you can also browse your repository via the web:

- URL is the same as what you would give to local command-line svn client

- Significant access control capabilities exist, but are configured through the underlying web server
  - can provide read-only access
  - can allow/disallow access to specific sub-directories of a repository



Getting Started with Subversion

Repositories

- Subversion stores files in a central location called a repository (the svn repo)
- You can create a repo anywhere you want (could be on your own local workstation)
- Generally, this should be:
  - accessible to your collaborators
  - backed up daily (you have been warned twice)
  - sufficiently large, operating on a local file system
- You don't usually need more than one repo, but if you do, multiple repo's can be defined and hosted on the same server
- Sysnet and TACC already host large SVN servers
- The remote repository will effectively store every version of each tracked file/directory you identify
  - this does not mean it has a separate "copy" of each file, so the disk space bloat is not as bad as you might suspect
  - the date and time of a each revision is maintained along with the user who committed it (along with a log entry)

Invoking the Subversion Client

- svn <subcommand> [options] [args]

- All roads lead thru the svn client
- The “subcommand” argument tells the client which operation it is to perform; the [options] and [args] values are then specific to a given command mode.
- Some common subcommands:
  - import
  - checkout (co)
  - update
  - commit
  - diff
  - log
  - info
  - add
  - delete (del, remove, rm)
  - mkdir
  - move (mv, rename, ren)
  - copy (cp)
  - status
  - list
- Plan on forgetting the above? “svn help” will show all available subcommands
- Commands are usually recursive (for example, an “svn update” will attempt to update the current directory and all sub-directories currently being managed by subversion).

Log Messages

- The “commit” and “import” commands require log messages to attach to the relevant repository changes

- Subversion uses the $EDITOR or $SVN_EDITOR environment variable to invoke your favorite editor to allow you to enter a log message when required
  - if not set, you will likely get an error
  - set in your local shell to resolve
    - > export EDITOR=emacs

- Alternatively, you can specify the log message on the command line with the –m option to commit
    - > svn commit –m “some message”

Creating a Repository

- If you want to play with SVN or host a repository on your own server, you need to first create a repository using the admin tool (svnadmin)

- Repository creation of "myrepo":

  - csim@login1> svnadmin create myrepo

  - csim@login1> ls myrepo/

  - conf/ db/ format hooks/ locks/ README.txt

- That's it, you now have a local repository created to host your development project. The next step is to import your desired files and directories into the repo.

- Note: to support remote network operations via http(s), you also need to configure a web server appropriately
- Let's assume that the web server side has been done for us and that "myrepo" above is available at:
  - http://svn.myserver.edu/myrepo

Importing Project Files

- Typically, the initial import starts from your own local copy of the code.
- If there are a lot of source files (or sub-directories), it is convenient to import the top-level directory recursively
- This means you need to clean up your source directory first
- Note that while SVN can track binary files, it does not make sense to import temporary files (or application binaries). Prior to import, you should clean:
  - All unnecessary text files (eg. emacs “~” tilde files)
  - Compiler generated object files (libraries, *.o files)
  - Any unwanted subdirectories
- With a clean local copy, it’s time for the import (starting from the top-level directory that you want to manage):
  - > cd files_to_import/
  - > svn import -m "Original import" http://svn.myserver.edu/myrepo
  - Adding inputs

Checking Out a Working Copy

- We just imported a project into svn; to begin subsequent development work, we need to first check out a working copy
- Use the “checkout” command to pull down the latest copy of all files in the repo
- svn checkout http://svn.myserver.edu/myrepo working_copy
- A working_copy/inputs/example-input1
- A working_copy/hello.c
- A working_copy/Makefile
- A working_copy/README
- Checked out revision 27.

- > cd working_copy/
- > ls
- hello.c inputs/ Makefile README

Updating to the Latest Version

- Usually, before adding anything new, or checking in your changes, you should synchronize the state of your local copy with things that may have changed in the repository version

- Use the "update" (up) command to update your local copy
  - Downloads new files
  - Downloads any differences in files between repo and the local copy
  - merges changes where possible
  - marks files that can't be merged (requires manual conflict resolution)

  - > svn up
  - A inputs/example-input2
  - A inputs/example-input3
  - A solver.c
  - Updated to revision 29.

- Similarly, you can update a specific file (eg. "svn up hello.c").

SVN Update Codes

- – U foo
  - File foo was (U)pdated (pulled from repository)
- – A foo
  - File foo was (A)dded to your working copy
- – D foo
  - File foo was (D)eleted from your working copy
- – R foo
  - File foo was (R)eplaced, that is it was deleted and a new file with the same name was added.
- – G foo
  - File foo received new changes and was also changed in your working copy. The changes did not collide and so were mer(G)ed.
- – C foo
  - File foo received (C)onflicting changes from the server. The overlap needs to be resolved by you.

Adding Files

- The “add” and “mkdir” subcommands are used to register new files and directories. The “commit” command checks them into the repo:

  - > touch CHANGELOG
  - > svn add CHANGELOG
  - A CHANGELOG
  - > svn mkdir examples
  - A examples
  - > svn commit -m "add changelog and examples/"
  - Adding CHANGELOG
  - Adding examples
  - Transmitting file data .
  - Committed revision 32.

Removing/Changing Files

- The “rm” and “mv” subcommands are used to register removes and file renames, respectively. The “cp” subcommand copies files and directories

  - > ls
  - CHANGELOG examples/ hello.c inputs/ Makefile README solver.c
  - > svn rm inputs/example-input3
  - D inputs/example-input3
  - > svn mv hello.c main.c
  - A main.c
  - D hello.c
  - > svn cp solver.c solver_new.c
  - A solver_new.c
  - > svn commit -m "file mods"
  - Deleting hello.c
  - Deleting inputs/example-input3
  - Adding main.c
  - Adding solver_new.c

Committing Local Changes

- You've seen that adding and deleting files is a two-step process
  - Step1: register the changes (add, rm, mkdir, etc)
  - Step2: commit the changes to the repo

- Similarly, any changes you make to previously existing files require a "commit" to send the changes to the central repo

  - > echo "4/1/09 Increased performance by 100X" >> CHANGELOG
  - > svn commit -m "updating changelog"
  - Sending CHANGELOG
  - Transmitting file data ..
  - Committed revision 36.

Checking Local Status

- The “status” subcommand shows the state of your local files and directories. If the local repo is in sync with the master repo, no output is shown:
  - > svn status
  - >
- Otherwise it summarizes the local changes:
  - > svn status
  - ? a.out (file is not managed by svn)
  - A regression (registered for addition)
  - M solver.c (file has local modifications)
- More details available with “-v” (specific revision number and who made the last modification)
  - > svn status –v

Viewing the Log Messages

- > svn log main.c
- Shows you:
  - all the past log messages
    - who made the commit
    - version numbers
    - total revisions
    - etc.
- Doesn't change anything (just a query)

> svn log main.c

---

r35 | karl | 2009-05-21 13:20:40 -0500 (Thu, 21 May 2009) | 2 lines

removed getenv() checks

---

r34 | karl | 2009-05-21 13:20:16 -0500 (Thu, 21 May 2009) | 2 lines

removed debug #ifdefs

---

r33 | karl | 2009-05-21 13:15:48 -0500 (Thu, 21 May 2009) | 1 line

file mods

---

r27 | karl | 2009-05-21 12:04:00 -0500 (Thu, 21 May 2009) | 1 line

Original import

Showing Differences

- The “diff” subcommand shows the file differences between two revisions (or repositories). By default, it compares your local version to the current master repository version.
- If the local repo is in sync with the master repo, no output is shown.
- Otherwise, it shows the difference between the local copy and the repo version
- differences are shown in the unified diff format
- can be used to trivially generate a patch file for use with the “patch” utility
- > svn diff main.c
- Index: main.c
- ===
- --- main.c (revision 37)
- +++ main.c (working copy)
- @@ -26,7 +26,7 @@
-
- MPI_Barrier(MPI_COMM_WORLD);

Resolving Conflicts

- The trickiest bit of using svn is often resolving conflicts when two developers have made changes to the same file
- Look for "C" when you update
  - When a conflict does occur, SVN will do its best to point you to the offending area using conflict markers (“<“, “=“, and “>”)
  - Three tmpfiles are also created. These are the original three files that could not be merged
  - SVN will not allow you to commit files with conflicts
- Method for Resolution (first, relax and be in a good mood)
  - Hand merge the two sets of changes within the conflicted file (filename)
  - Once you have resolved, use “svn resolved filename” to identify resolution and commit the results

    - <<<<<<< .mine
    - printf(" --> Process # %8i of %8i is alive. ->%s\n",
    - =======
    - printf(" --> MPI Process # %5i of %5i is alive. ->%s\n",
    - >>>>>>> .r38
    - num_local,num_procs,mach_name);

Comments on Revision Numbers

- Revision numbers are applied to an object to identify a unique version of that object.
- Although CVS and SVN have very similar approaches and syntax, there is a fundamental difference in revisioning schemes

- CVS Approach:
  - Revision numbers are per file.
  - No connection between two files with the same revision number.
  - A commit only modifies the version number of the files that were modified.

    foo.c rev 1.2 and bar.c rev 1.10.

  - After commit of bar.c:

    foo.c rev 1.2 and bar.c rev 1.11.

Comments on Revision Numbers

- SVN takes a different approach:
  - Revision numbers are global across the whole repository
  - Snapshot in time of the whole repository tree
  - A commit modifies the version number of all the files

    foo.c rev 25 and bar.c rev 25.

  - After commit of bar.c:

    foo.c rev 26 and bar.c rev 26.
    Subtle Note: foo.c rev 25 and 26 are identical.

- One benefit is that you can have a colleague checkout version r456 and that is sufficient to specify the state of the entire repository tree

Cute Tricks in Your Code

- You can keep track of the subversion revision number automatically in all your source code files
- Just add “$Id: $” into a comment in your file
- This gets expanded when you check in/out the relevant file to include the revision, last update date, and the user making the changes
- Note that the “Id” keyword has to be registered with svn to have the expansion take effect
- For example:

```
> svn propset svn:keywords 'Id Revision' solver.c
> svn commit solver.c -m "enabling keyword expansion"
// $Id: solver.c 40 2009-05-21 19:41:37Z karl $"
```

Ignoring Certain Files

- If you tire of having svn showing you “?” for files that are not part of the working repository, you can ask it to ignore certain files
- This is done by setting the svn:ignore property on the target folder with the signature of the file or folder to ignore
  - > svn status
  - ? hello
  - > svn propset svn:ignore hello .
  - property 'svn:ignore' set on '.’
  - > svn commit -m "ignoring the executable"
  - Sending trunk
  - Committed revision 45.
  - > svn status
  - >

Tags and Branches

- Subversion supports the notion of repository tags thru the “cp” command

- A common convention is to break up your top-level software directory into the following three directories
  - trunk/ (stable development)
  - tags/ (preserves a copy of specific revisions)
  - branches/ (major new developments)

- Note: no active commits typically go under the tags/ directory, only under branches/ and trunk/

Tags and Branches

- Note that the cost of branching and tagging need not be proportional to the project size
- Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time
- There is no technical distinction between tagging and branching; all you have to do is make a “copy” of your repository to create either one:

  - > svn cp . http://svn.myserver.edu/myrepo/tags/0.10-release
  - Committed revision 43.
  - > cd ../tags/
  - > svn up
  - A 0.10-release
  - A 0.10-release/regression
  - A 0.10-release/solver_new.c
  - A 0.10-release/main.c
  - A 0.10-release/CHANGELOG
  - A 0.10-release/solver.c
  - A 0.10-release/Makefile
  - A 0.10-release/README

Tags and Branches

- Once new functionality or code refactoring has been completed satisfactorily in a branch, it is time to merge back to trunk/
- The “merge” subcommand is used to do this (it applies the differences between two sources to a working copy path)
  - > svn merge http://svn.myserver.edu/myrepo//branches/0.11 .
  - --- Merging r42 through r45 into '.':
  - G   .
  - --- Merging r46 through r49 into '.':
  - A   vis_interface.c
  - A   io_interface.c
  - A   solver_petsc.c
  - D   solver_new.c
  - > svn commit -m "Merge branch 0.11 into trunk/"
  - Adding   trunk/io_interface.c
  - Deleting   trunk/solver_new.c
  - Adding   trunk/solver_petsc.c

When to Commit?

- Committing too often may leave the repo in a state where the current version doesn't compile
- Committing too infrequently means that collaborators are waiting for your important changes, bug fixes, etc. to show up
  - makes conflicts much more likely
- Common policies
  - committed files must compile and link
  - committed files must pass some minimal regression test(s)
- Come to some agreement with your collaborators about the state of the repo

Getting Started with Git

http://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide#323764

http://git-scm.com/downloads

https://bitbucket.org/ https://github.com/

History

Created by Linus Torvalds for work on the Linux kernel ~2005

Some of the companies that use git:

Git is a …

DISTRIBUTED Version Control System

OR

DIRECTORY Content Management System

OR

TREE history storage system

Distributed

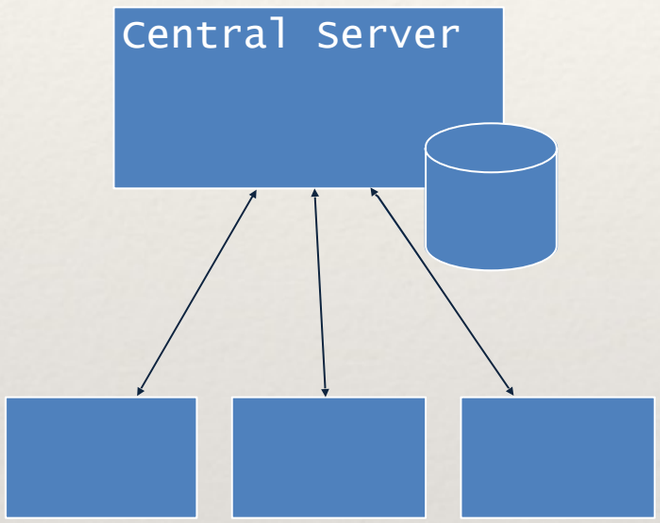
Everyone has the complete history

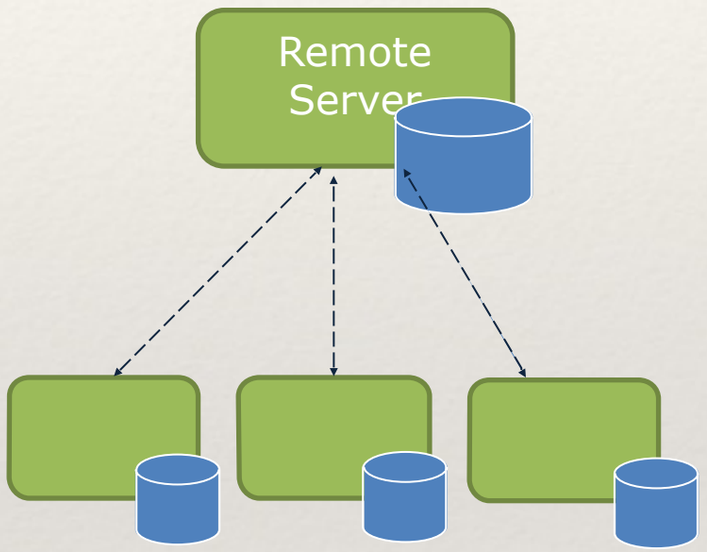
Everything is done offline

No central authority

Changes can be shared without a server

Centralized VC vs. Distributed VC





Branching

Forget what you know from Central VC

Git branch is “Sticky Note” on the graph

All branch work takes place within the same folder within your file system.

When you switch branches you are moving the “Sticky Note”

Initialization

```
csim@hbar /scratch/class $ mkdir CoolProject
csim@hbar /scratch/class $ cd CoolProject/
csim@hbar /scratch/class/CoolProject $ git init
Initialized empty Git repository in /scratch/class/CoolProject/.git/
csim@hbar /scratch/class/CoolProject (master) $ echo "hi there" > README
csim@hbar /scratch/class/CoolProject (master) $ git add .
csim@hbar /scratch/class/CoolProject (master) $ git commit -m "my first commit"
[master (root-commit) f253f1b] my first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README
csim@hbar /scratch/class/CoolProject (master) $
```

Introduce yourself to git

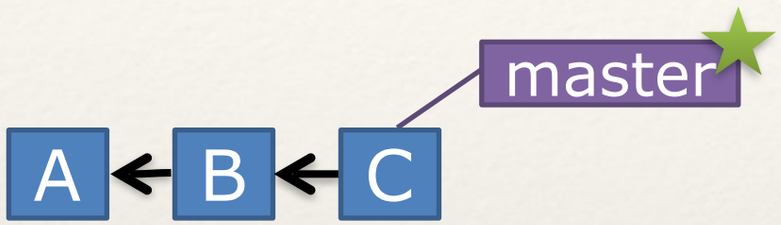
```
csim@hbar /scratch/class/CoolProject (master) $
csim@hbar /scratch/class/CoolProject (master) $ curl -l http://install.hovr2pi.org/git.config
#!/bin/bash

git config --global user.name "Christopher Simmons"
git config --global user.email csim@hovr2pi.org
case `uname` in

Darwin)
        git config --global credential.helper osxkeychain
        ;;

Linux)
        git config --global credential.helper "cache --timeout=3600"
        echo "caching does no good if git-daemon is not installed"
;;
esac

csim@hbar /scratch/class/CoolProject (master) $ curl -l http://install.hovr2pi.org/git.config | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
113   339  113   339    0     0   1296      0 --:--:-- --:--:-- --:--:--  2607
caching does no good if git-daemon is not installed
csim@hbar /scratch/class/CoolProject (master) $
```

Branches Illustrated


```
> git commit –m ‘my first commit’
```

- On my first commit I have A.
- The default branch that gets created with git is a branch named Master.
- This is just a default name. As I mentioned before, most everything in git is done by convention. Master does not mean anything special to git.

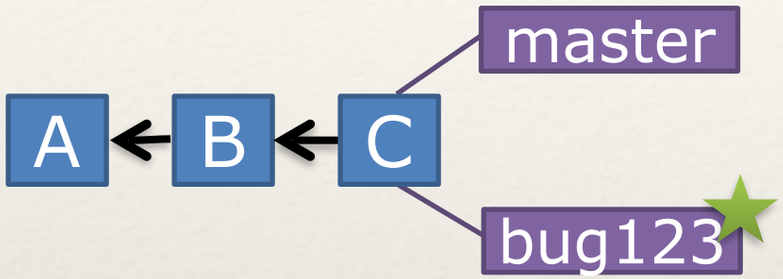
Branches Illustrated




```
> git commit (x2)
```

- We make a set of commits, moving master and our current pointer (*) along

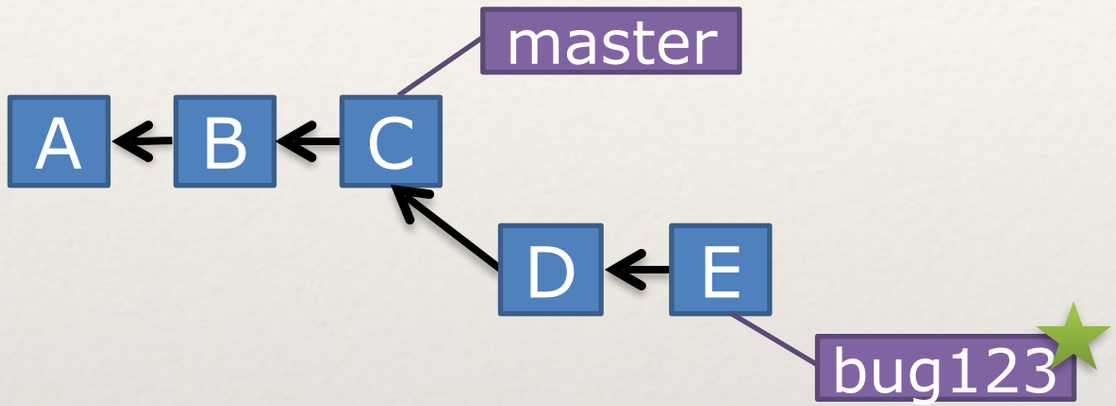
Branches Illustrated




```
> git checkout -b bug123
```

- Suppose we want to work on a bug. We start by creating a local “story branch” for this work
- Notice that the new branch is really just a pointer to the same commit (C) but our current pointer (*) is moved

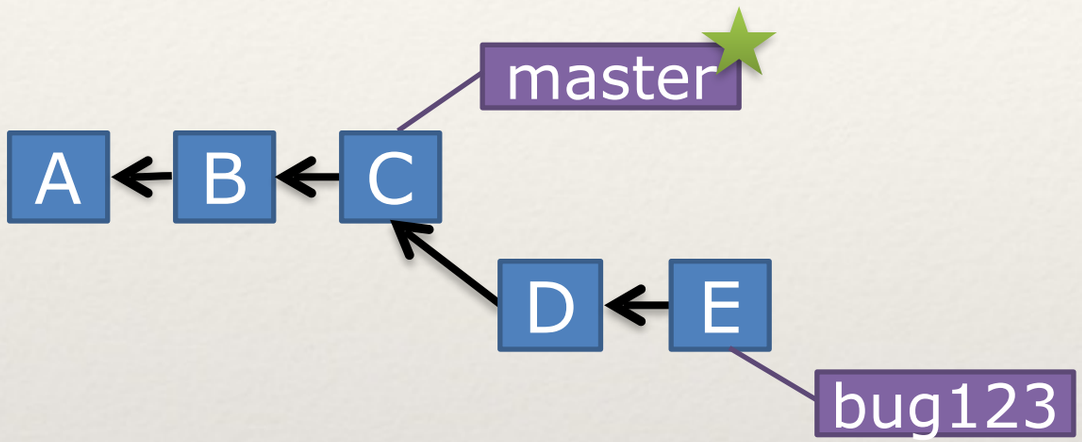
Branches Illustrated




```
> git commit (x2)
```

- Now we make commits and they move along, with the branch and current pointer following along

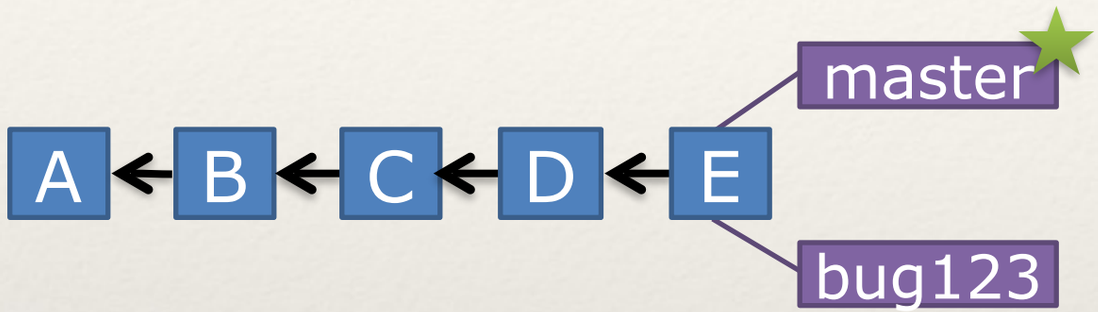
Branches Illustrated




```
> git checkout master
```

- We can “checkout” to go back to the master branch
- This is where you might freak out. The changes you made are no longer in the current checkout but this is the correct git workflow

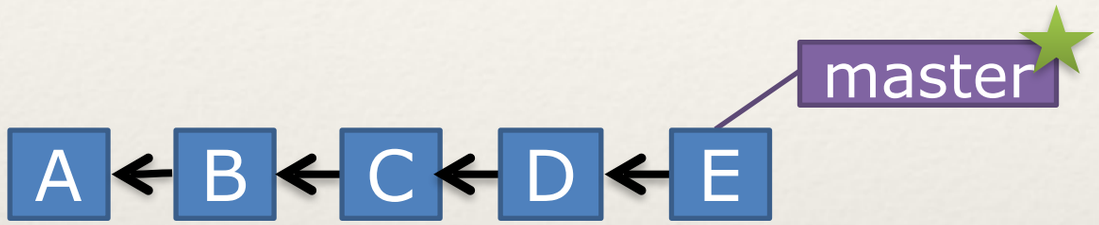
Branches Illustrated




```
> git merge bug123
```

- Now we merge from the story branch, bringing those change histories together

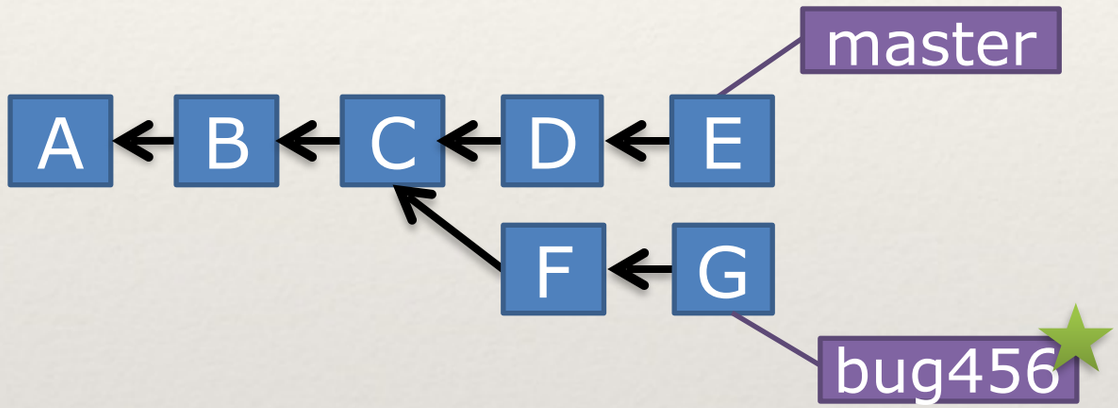
Branches Illustrated




```
> git branch -d bug123
```

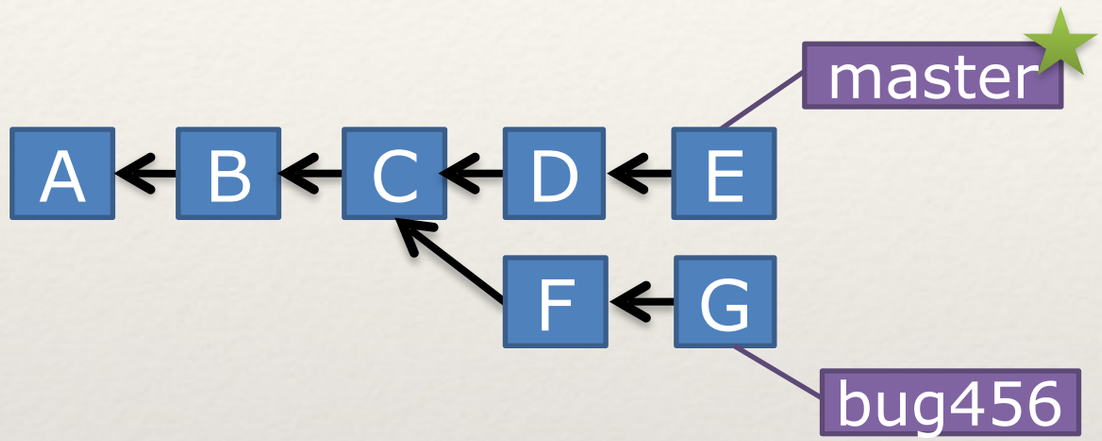
- And since we're done with the story branch, we can delete it. This all happened locally, without affecting anyone upstream.

Branches Illustrated



- Another common scenario is as follows:
  - We created our “story branch” off of C.
  - But some changes have happened in master (bug123 which we just merged) since then
  - And we’ve made a couple of commits in bug456

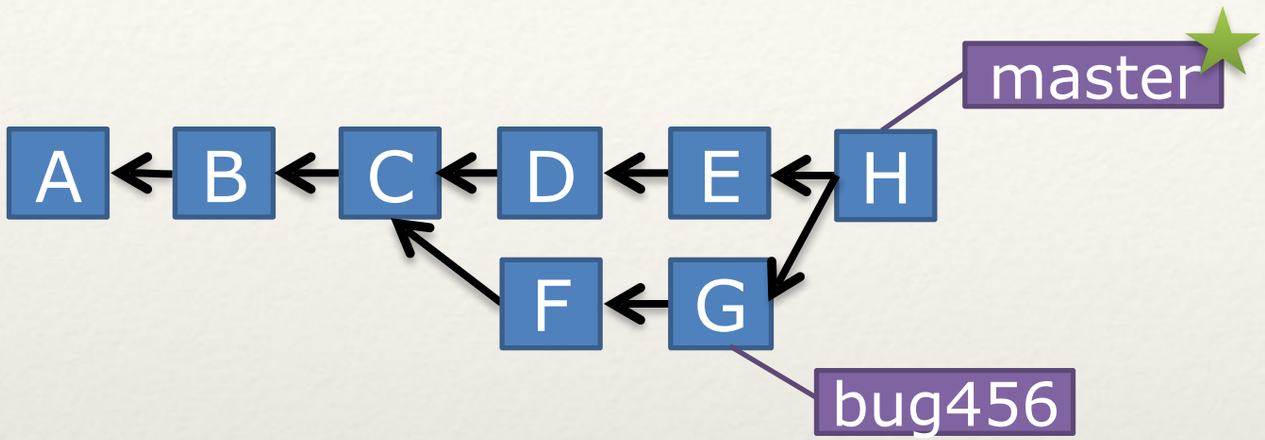
Branches Illustrated




```
> git checkout master
```

- Again to merge, we checkout back to master which move our * pointer

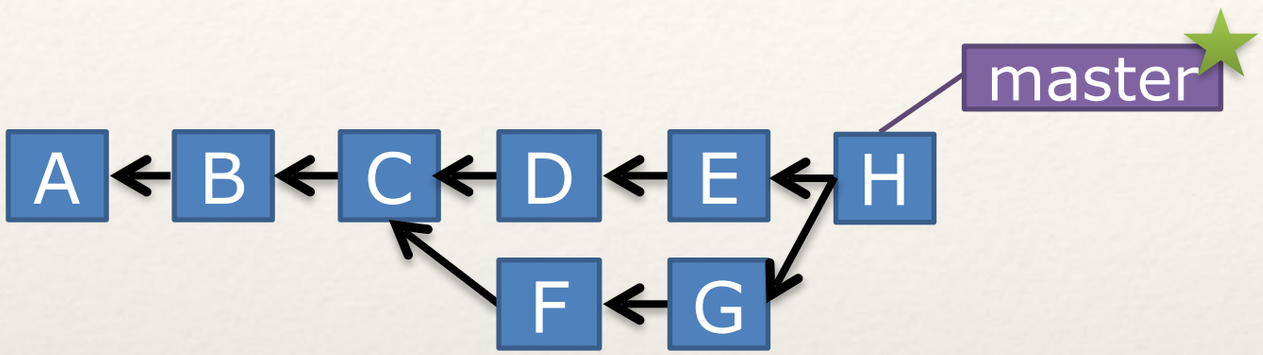
Branches Illustrated




```
> git merge bug456
```

- Again now we merge, connecting the new (H) to both (E) and (G)
- Note that this merge, especially if there are conflicts, can be unpleasant to perform

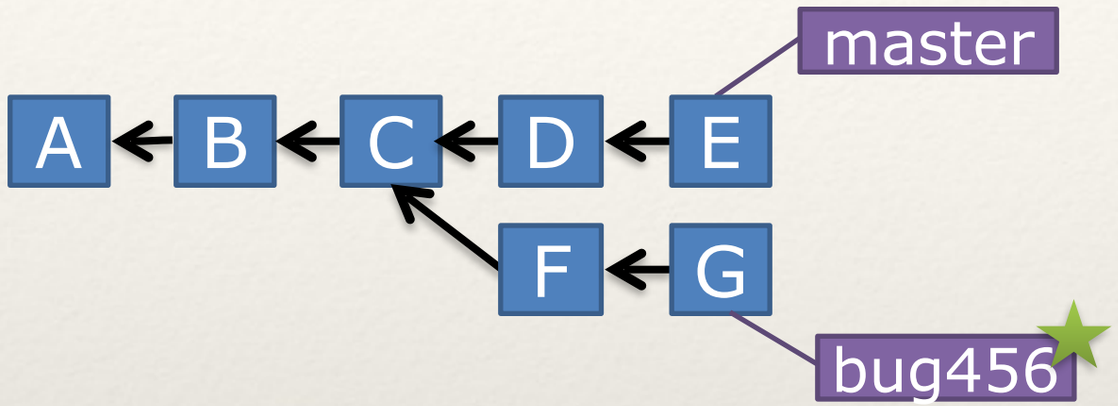
Branches Illustrated




```
> git branch -d bug456
```

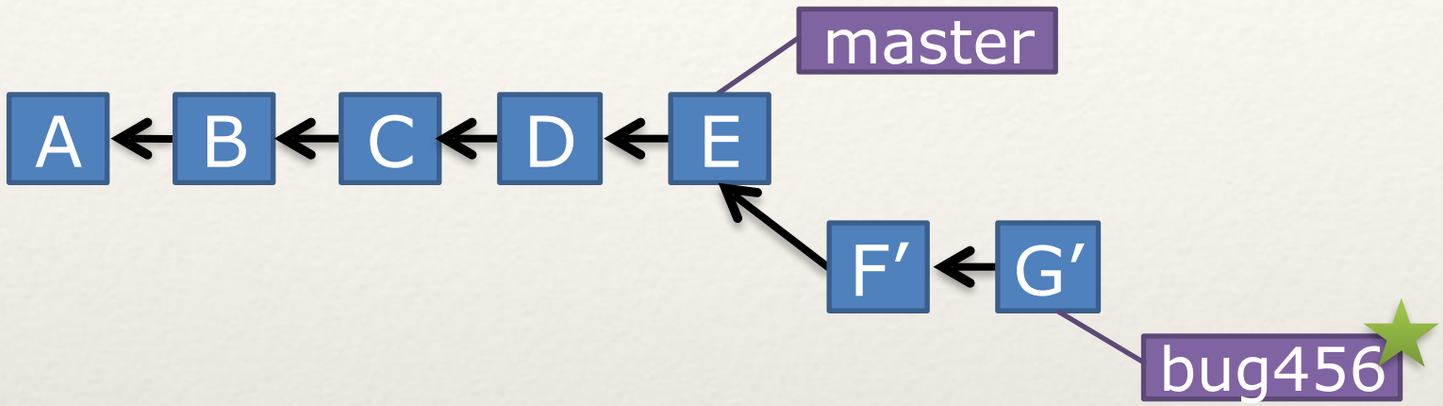
- Now we delete the branch pointer
- But notice the structure we have now. This is non-linear. That will make it hard to see the changes independently and can get very messy over time

Branches Illustrated



- Let's try this again but using the Rebase workflow
- So we are ready to merge bug456 once again

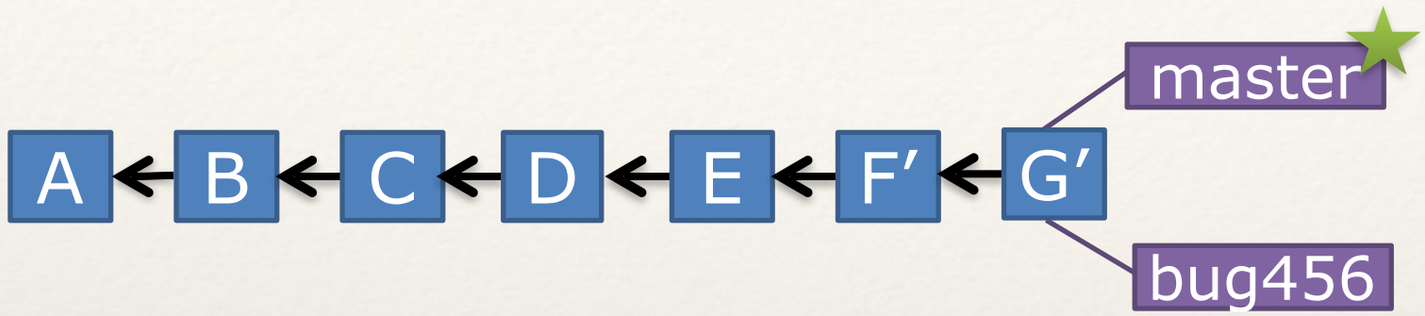
Branches Illustrated




```
> git rebase master
```

- Instead of merging, we “rebase”
- What this does is:
  - Take the changes we had made against (C) and undo them, but remember what they were
  - Re-apply them on (E) instead

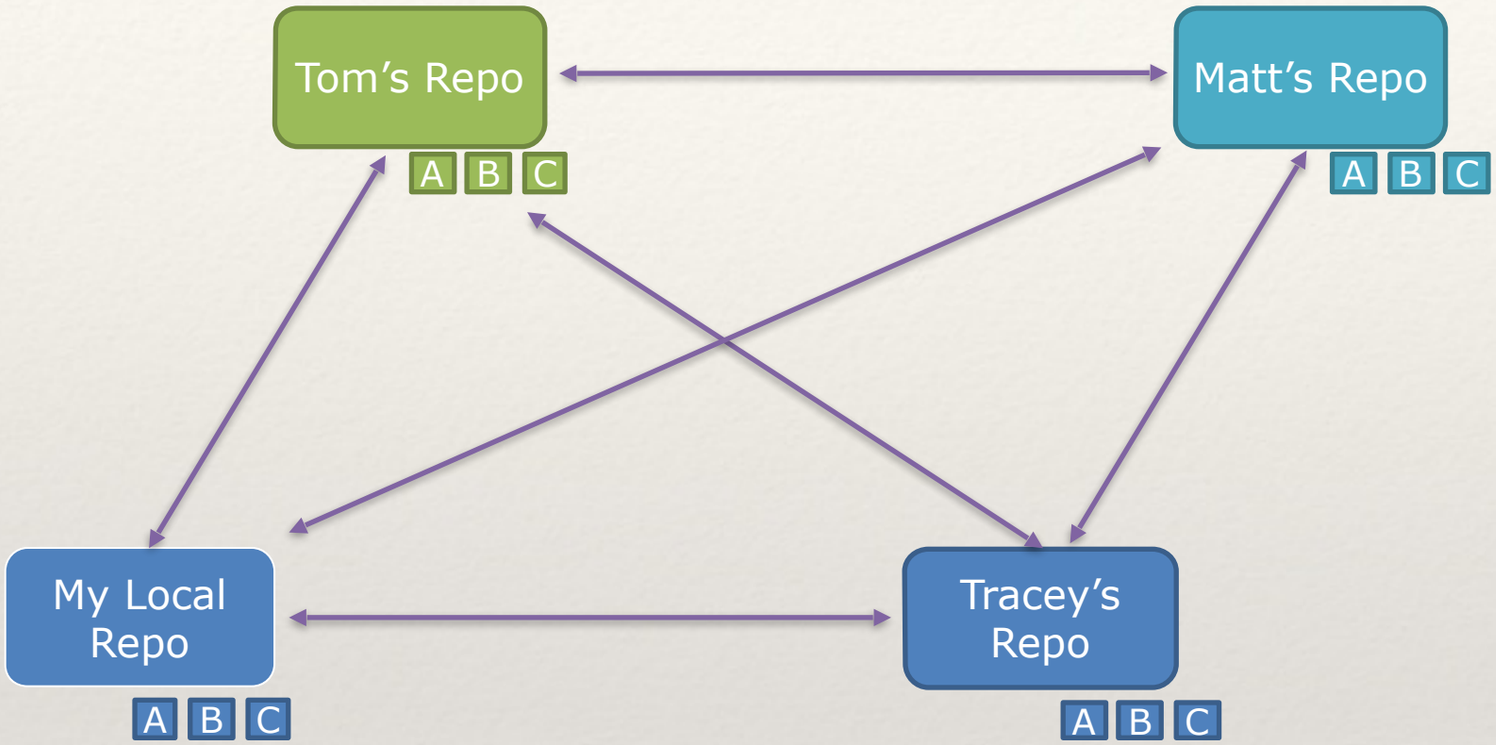
Branches Illustrated




```
> git checkout master
> git merge bug456
```

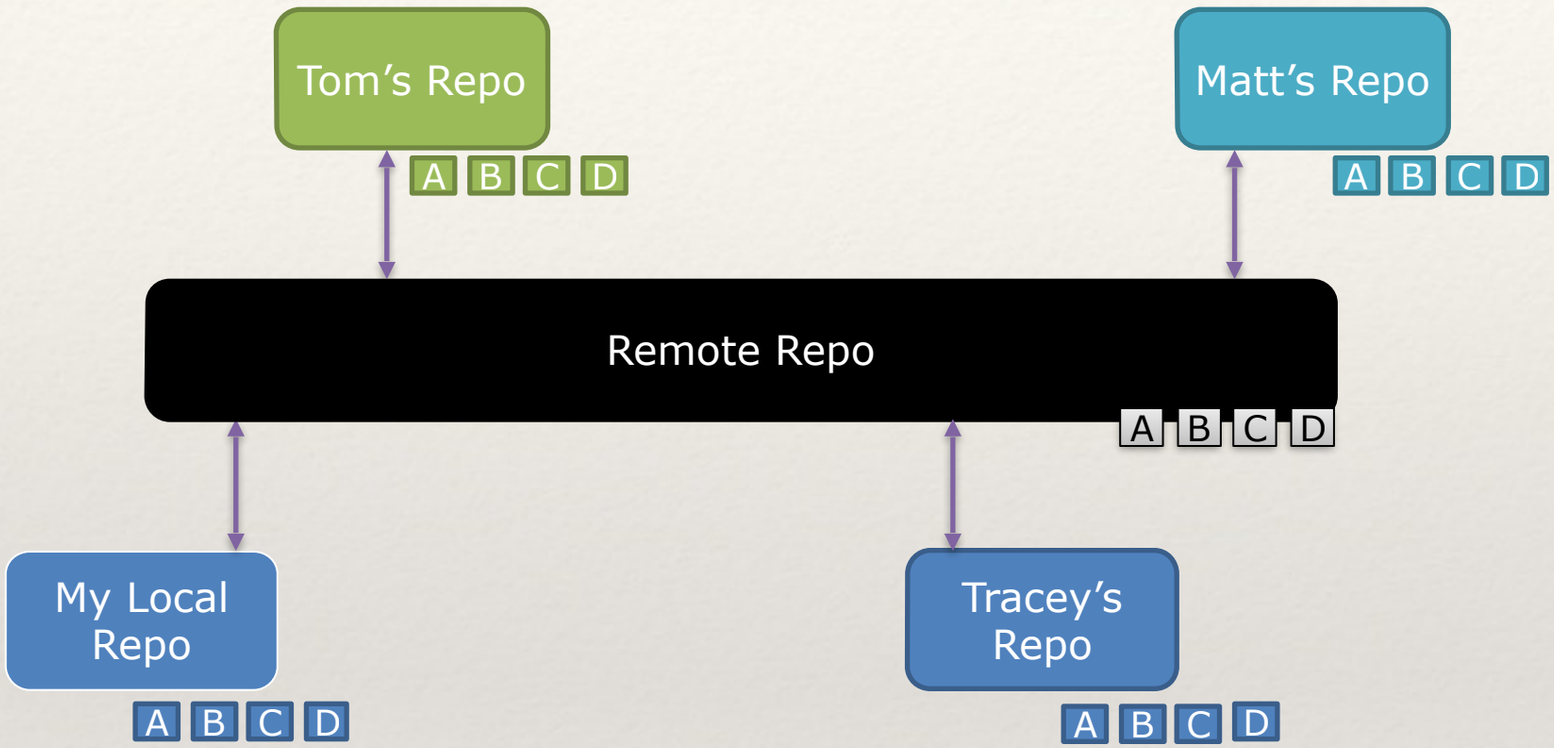
- Now when we merge them, we get a nice linear flow
- Also, the actual change set ordering in the repo mirrors what actually happened. F’ and G’ come after E rather than in parallel to it.

Sharing commits



- Since everyone on the team has a complete copy of the repo, you can do things like this
- But there is a better way …

Sharing commits



- There's nothing special about a remote server. You can have as many as you want
- But it enables a nice integration location, just like you are used to in centralized version control

Setting up a remote

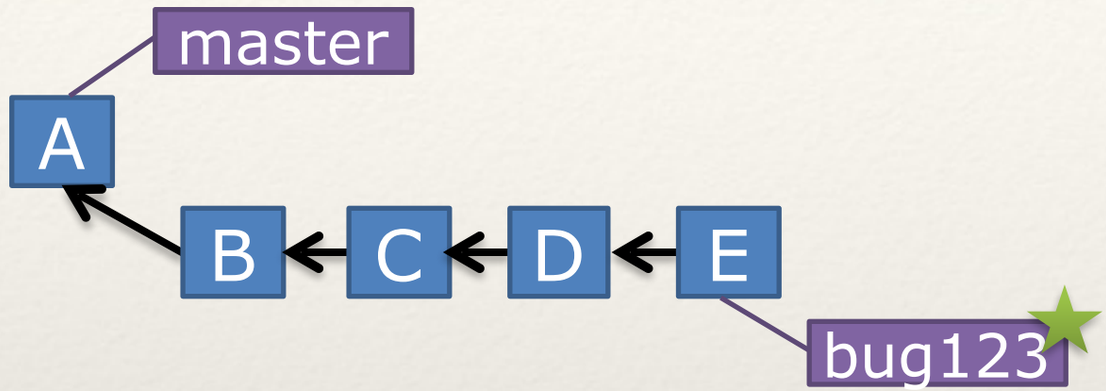
```
csim@hbar ~/git/cfour (master) $ git remote -v
origin  https://hovr2pi@bitbucket.org/hovr2pi/cfour.git (fetch)
origin  https://hovr2pi@bitbucket.org/hovr2pi/cfour.git (push)
csim@hbar ~/git/cfour (master) $
csim@hbar ~/git/cfour (master) $ git remote add germany simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git
csim@hbar ~/git/cfour (master) $ git remote -v
germany simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git (fetch)
germany simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git (push)
origin  https://hovr2pi@bitbucket.org/hovr2pi/cfour.git (fetch)
origin  https://hovr2pi@bitbucket.org/hovr2pi/cfour.git (push)
csim@hbar ~/git/cfour (master) $
```

git clone sets up remote

```
csim@hbar ~/git $ pwd
/h1/csim/git
csim@hbar ~/git $ git clone simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git dascfour
Cloning into 'dascfour'...
simmons@dirac.chemie.uni-mainz.de's password:
Warning: No xauth data; using fake authentication data for X11 forwarding.
remote: Counting objects: 60340, done.
remote: Compressing objects: 100% (21557/21557), done.
remote: Total 60340 (delta 45618), reused 51253 (delta 38349)
Receiving objects: 100% (60340/60340), 45.92 MiB | 10.15 MiB/s, done.
Resolving deltas: 100% (45618/45618), done.
Checking connectivity... done
Checking out files: 100% (11937/11937), done.
csim@hbar ~/git $ cd dascfour/
csim@hbar ~/git/dascfour (master) $ git remote -v
origin  simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git (fetch)
origin  simmons@dirac.chemie.uni-mainz.de:/mnt/software/git/aces2.git (push)
csim@hbar ~/git/dascfour (master) $
```

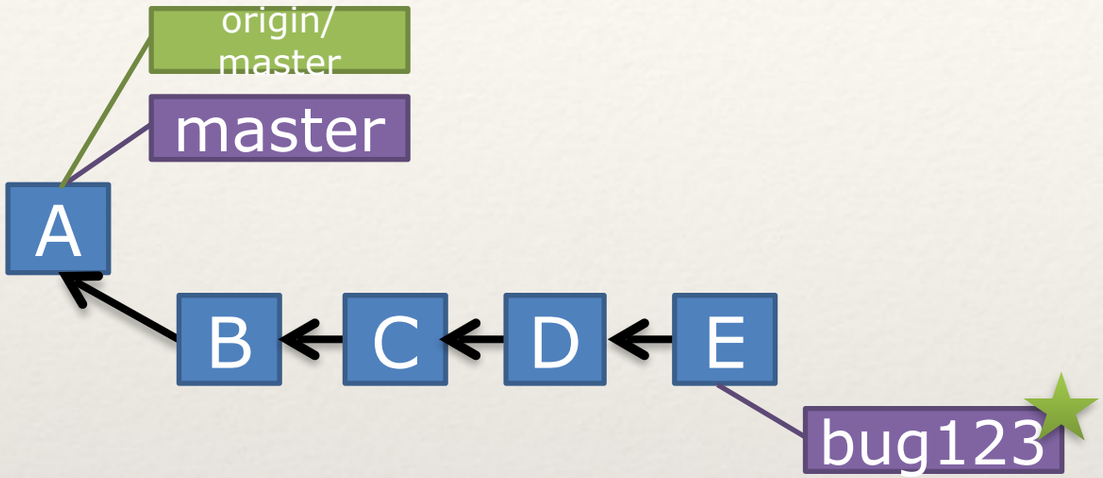
- Name remotes whatever you want
- Origin is only a convention

Branches Illustrated



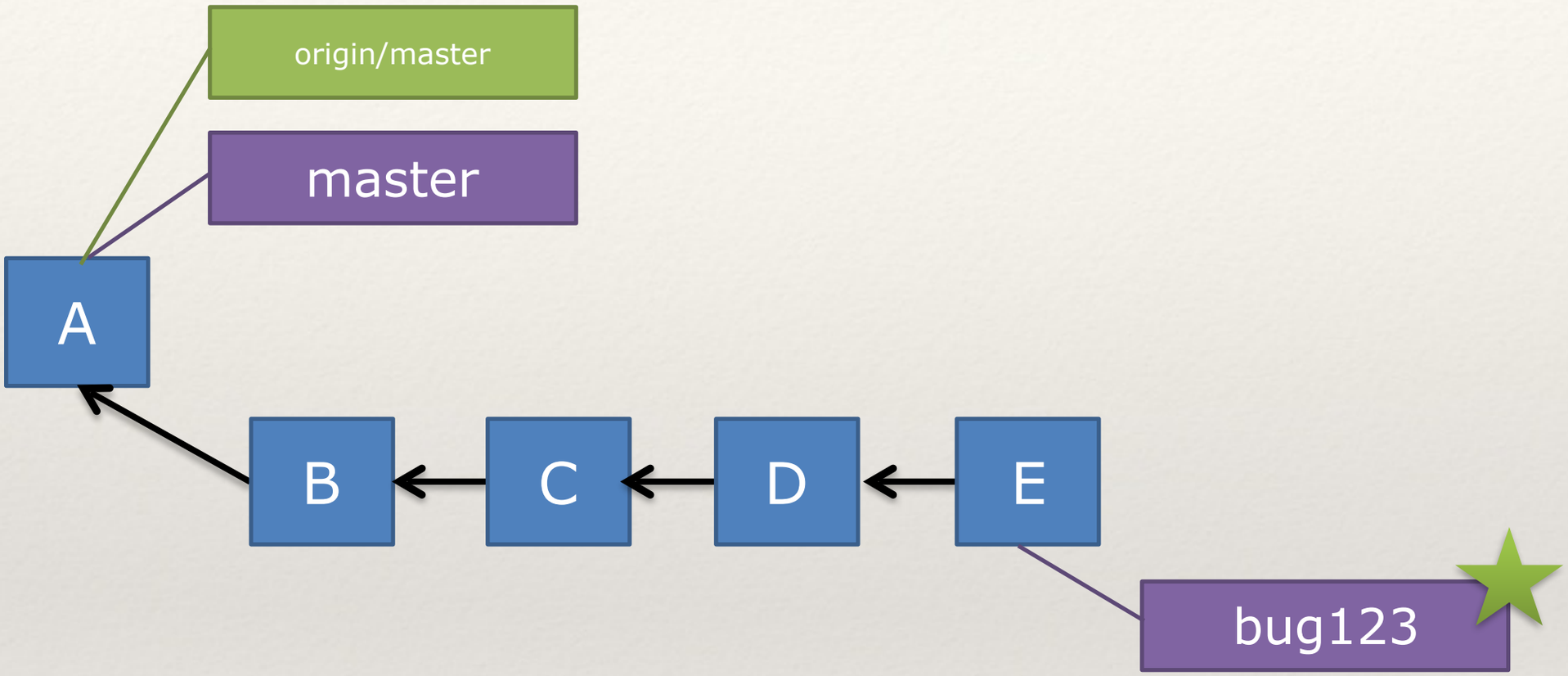
- Suppose we are here:
- We cloned master on (A) and have been fixing bug123 in our story branch

Branches Illustrated



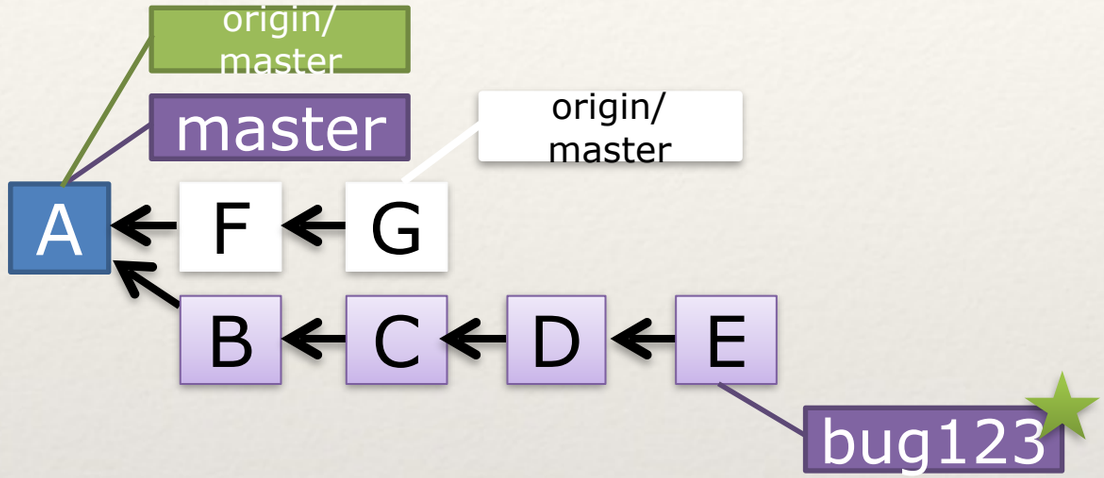
- This is the same as above except we are going to track an upstream remote as well; here it is the remote origin/master

Branches Illustrated



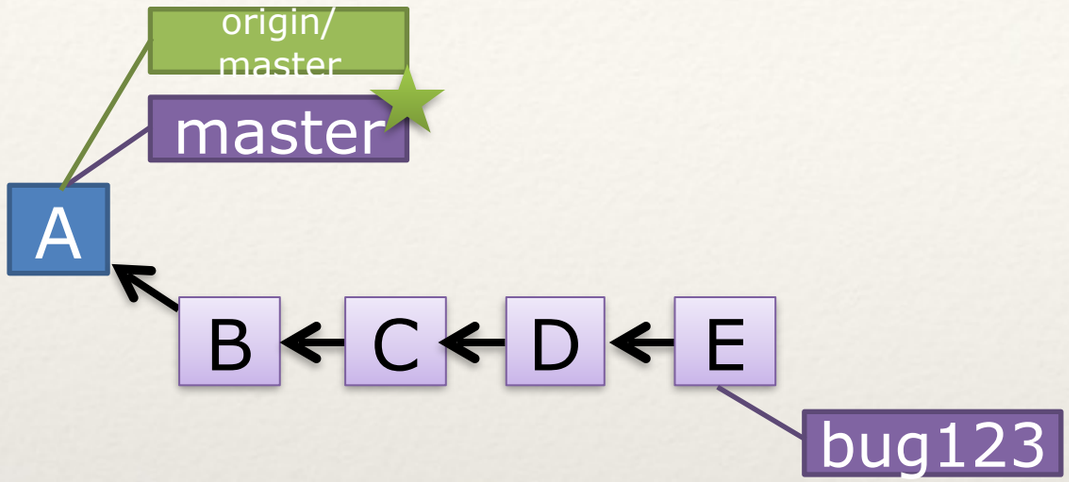
- This is the same as above except we are going to track an upstream remote as well; here it is the remote origin/master
- The changes on the bug123 branch are only know to the local repo. The remote server does not have these changes (or the bug123 branch for that matter)

Branches Illustrated



- In reality, there are two version of the origin/master pointer
  - One we know about for our “local” version of origin master
  - The other one is the actual point on the remote server which may or may not be at the same location

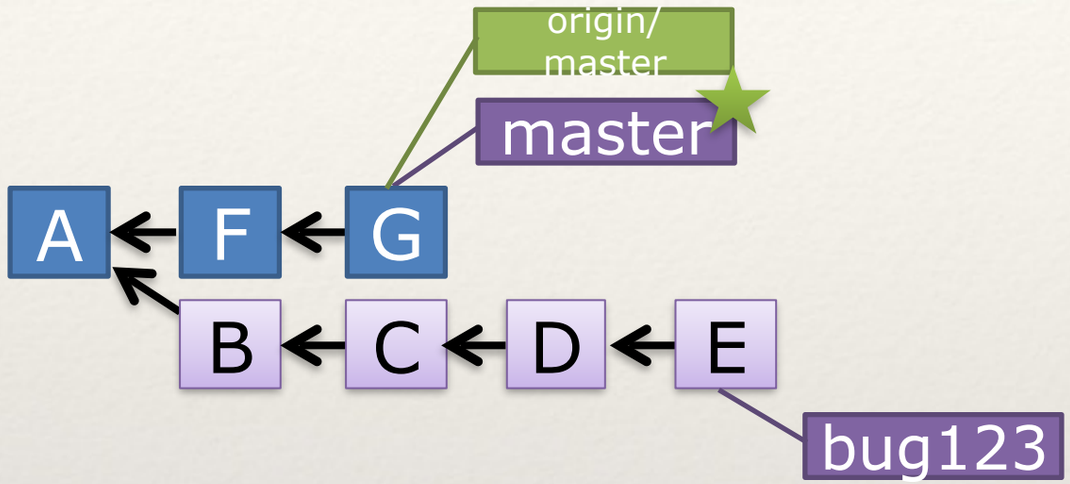
Branches Illustrated




```
> git checkout master
```

- So if this is what we know locally, we can update our master to catch up
- First we checkout master which move our current (*) to there. Note that we are actually on OUR master not the upstream one. This is always true. But the tracking branch is also pointing to (A) at this point.

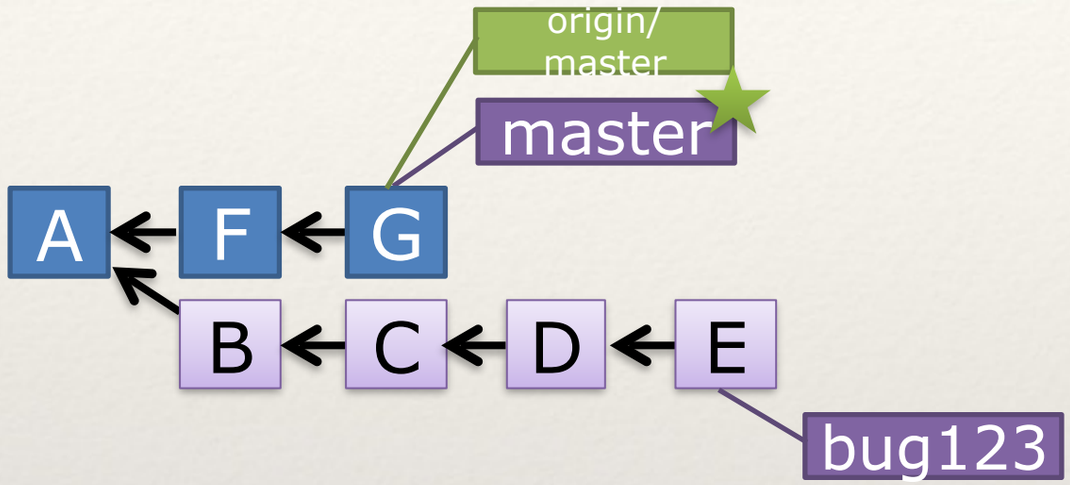
Branches Illustrated




```
> git pull origin
```

- Now, we can pull down the origin and move both along to their new place

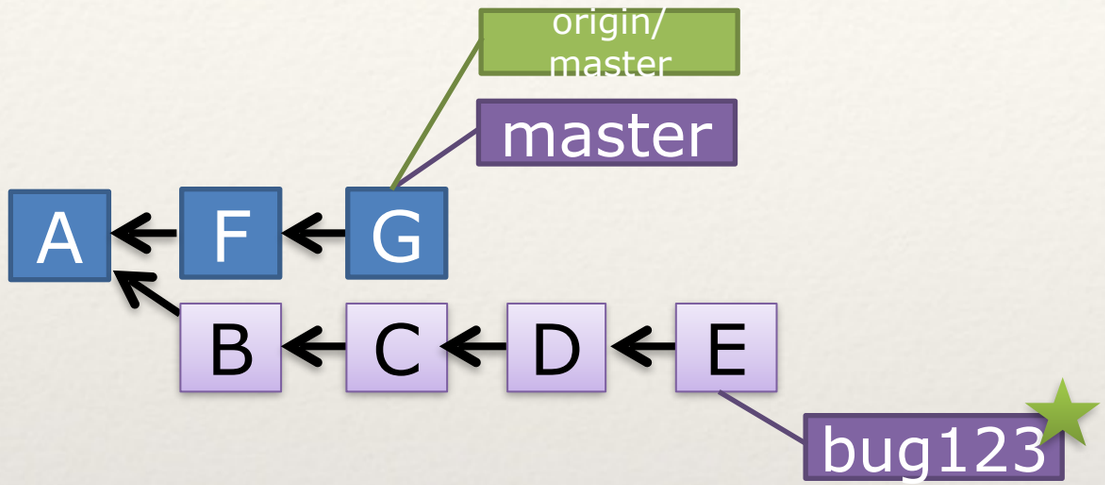
Pull = Fetch + Merge




```
> git pull origin
```

- Fetch — update your local copy of the remote branch
- Pull essentially does a fetch and then a merge in one step

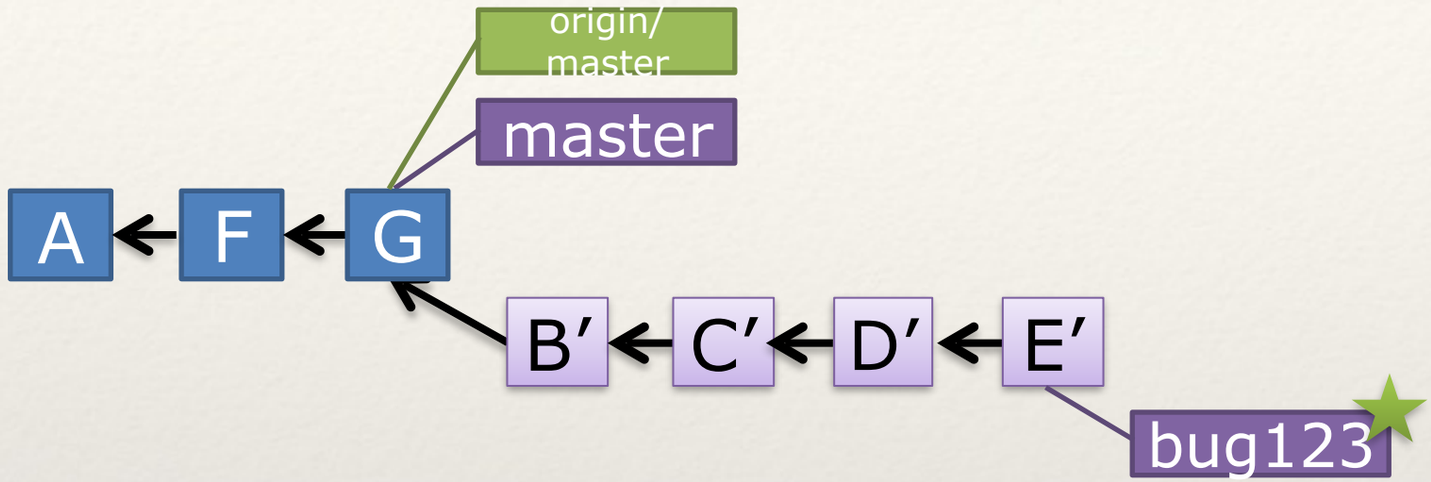
Branches Illustrated




```
> git checkout bug123
```

- Returning to our bug fix “story branch”, we have a similar problem to what we saw before
- B-C-D-E all come before F and G
- Merging now would create (non-linear) issues

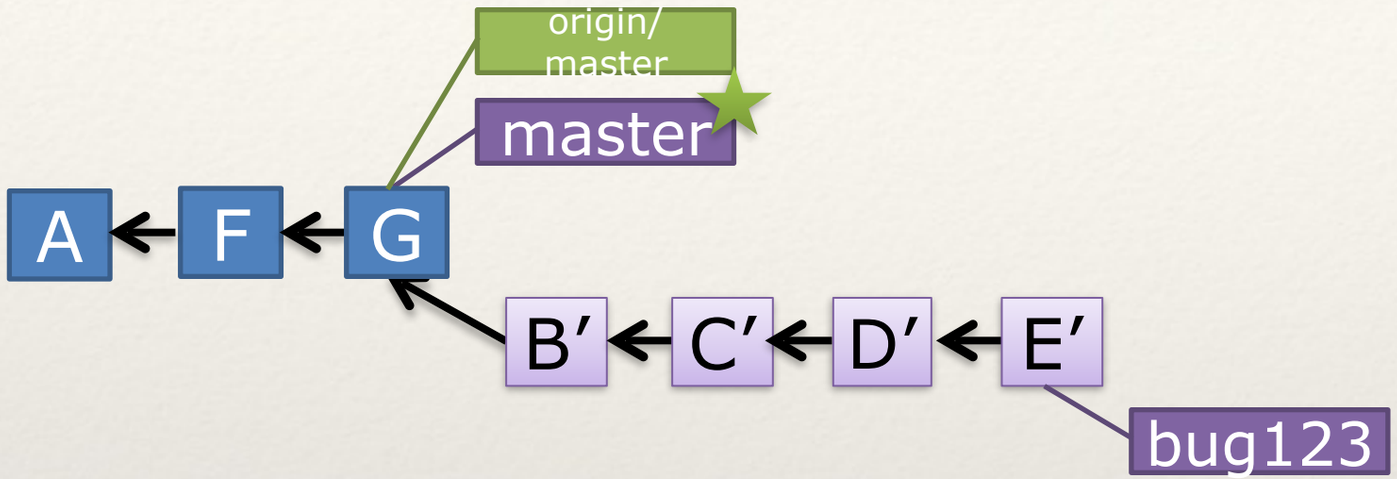
Branches Illustrated




```
> git rebase master
```

- So, we rebase our bug123 branch onto master

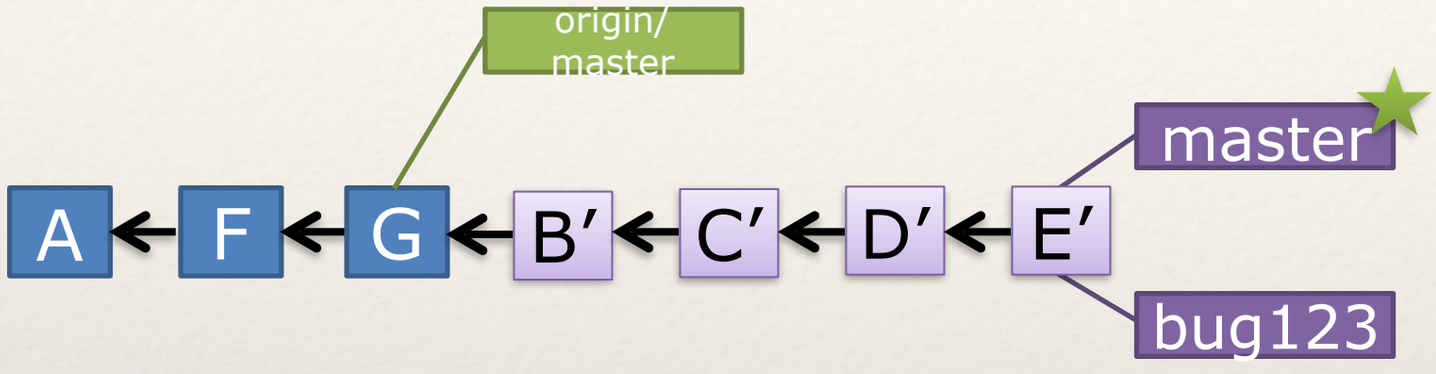
Branches Illustrated




```
> git checkout master
```

- We checkout master, once again

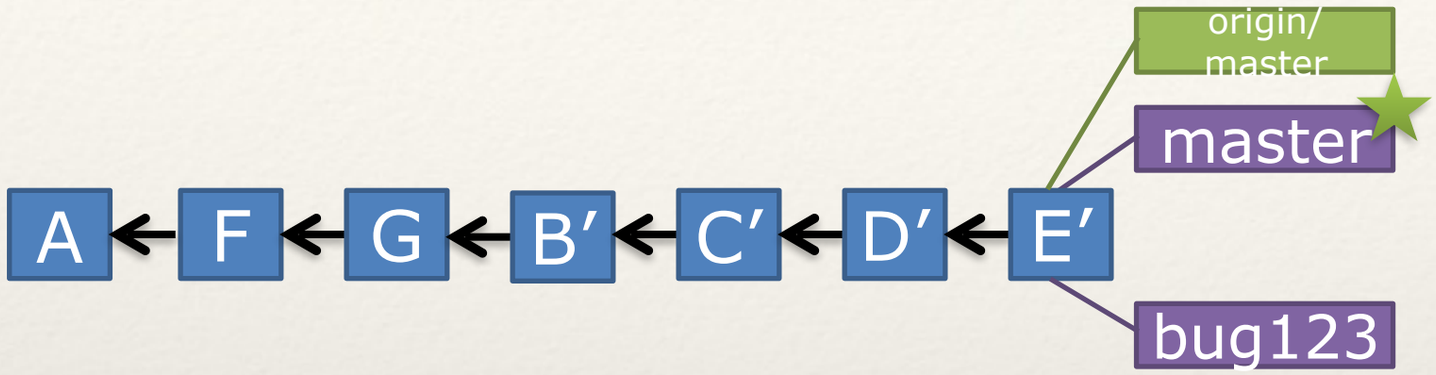
Branches Illustrated




```
> git merge bug123
```

- And merge bug123 branch with master and get our nice linear repo back

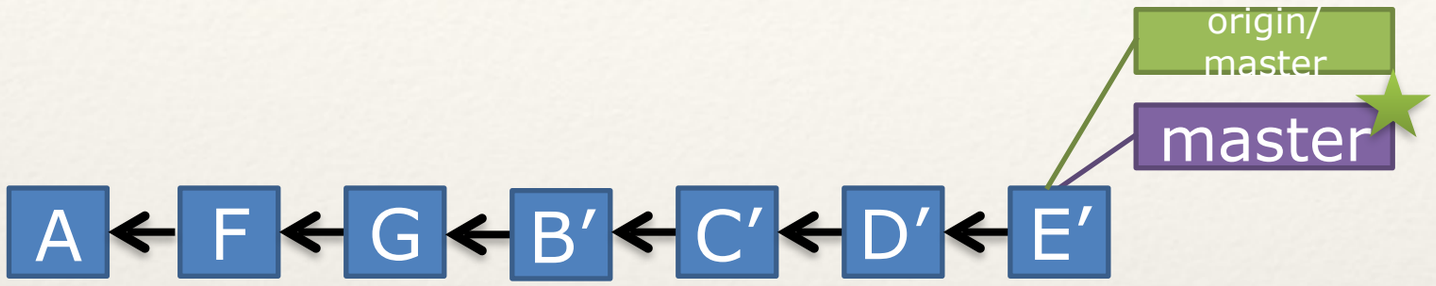
Branches Illustrated




```
> git push origin
```

- And finally, because we want to publish these changes upstream, we push to the origin, moving the pointer along to the same place as our local repo
- Push — pushes your changes upstream
- Git will reject pushes if newer changes exist on the remote
  - Good practice: Pull then Push

Branches Illustrated




```
> git branch -d bug123
```

- And finally, we can now delete our “story branch” bug123

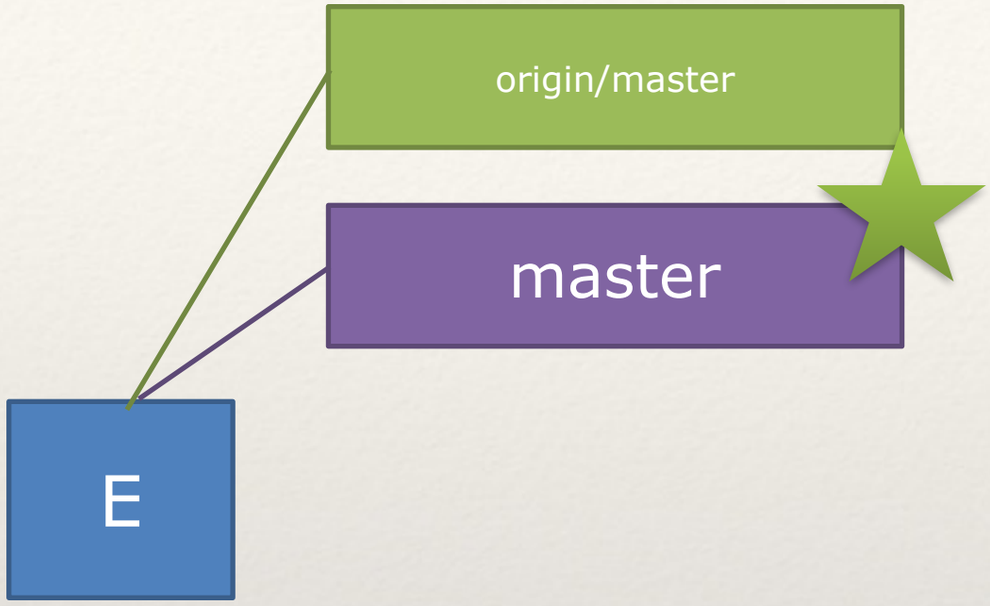
Short vs. Long-Lived Branches

- Local branches are short lived
- Staying off master keeps merges simple
- Enables working on several changes at once
  - Create
  - Commit
  - Merge
  - Delete

Short vs. Long-Lived Branches

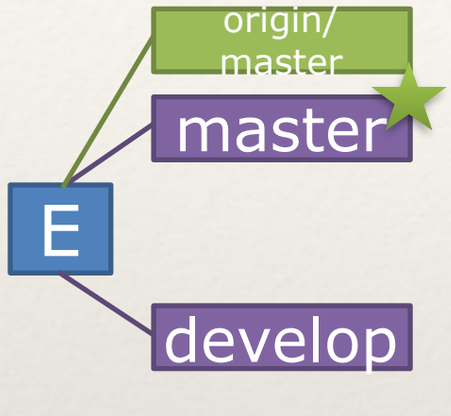
- Great for multi-version work
- Follow same rules as Master / Story branches
- Integrate frequently
- Pushed to Remotes

Branches Illustrated



- Say we want to start working on the next version of our project

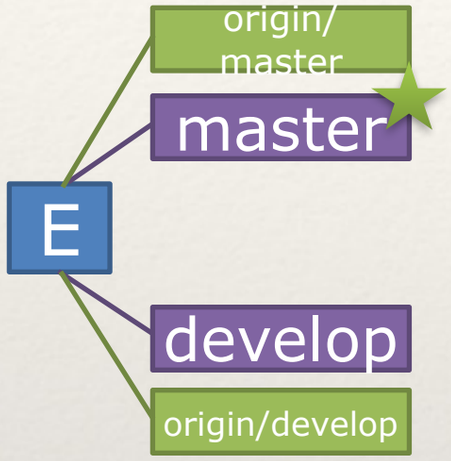
Branches Illustrated




```
> git branch develop
```

- We, will want to create a develop branch

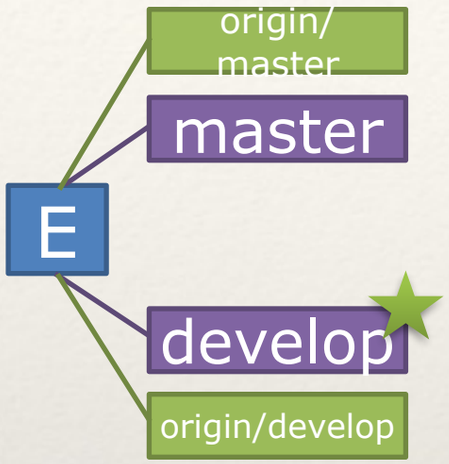
Branches Illustrated




```
> git push origin develop
```

- To share this with our team, we need to push this “long-lived” branch up to the remote

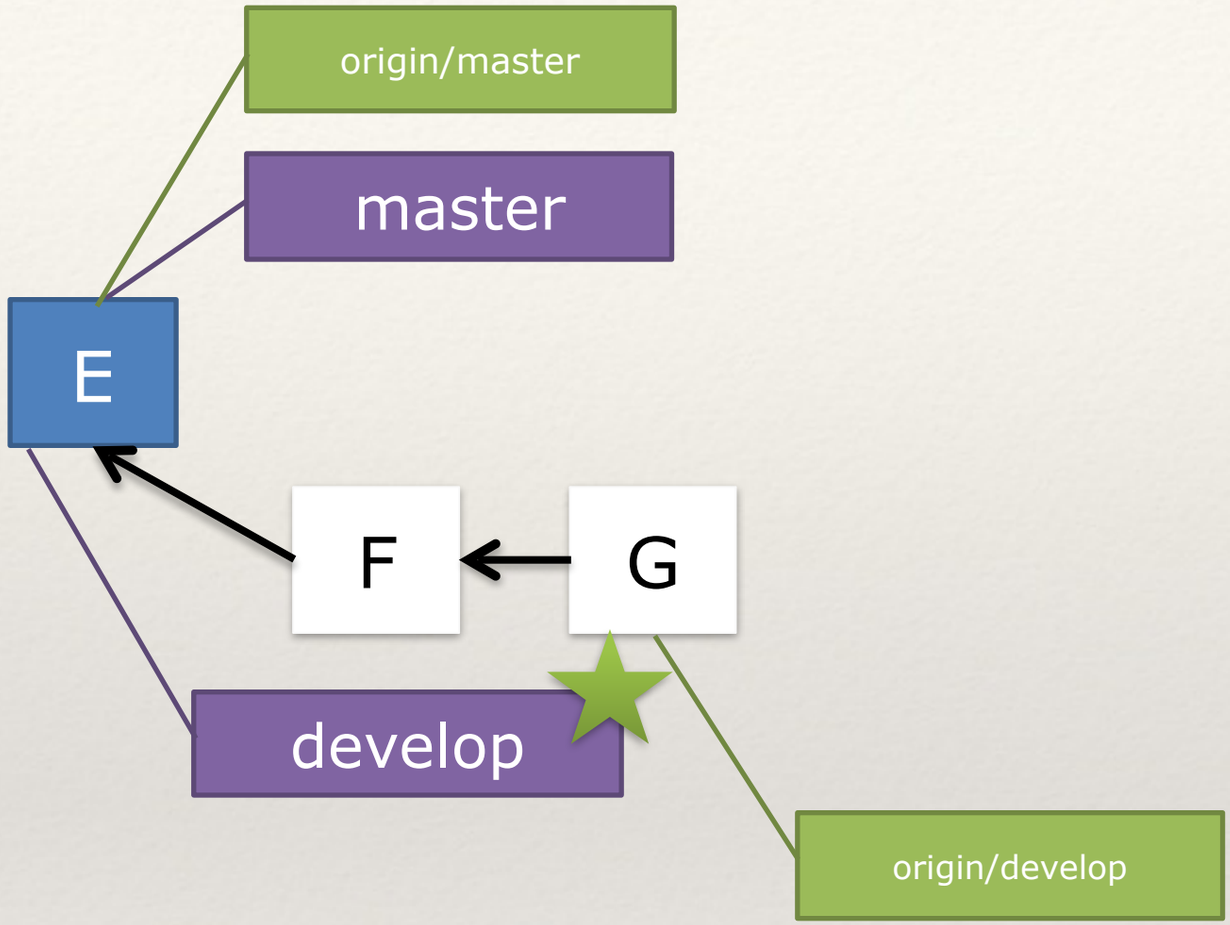
Branches Illustrated




```
> git checkout develop
```

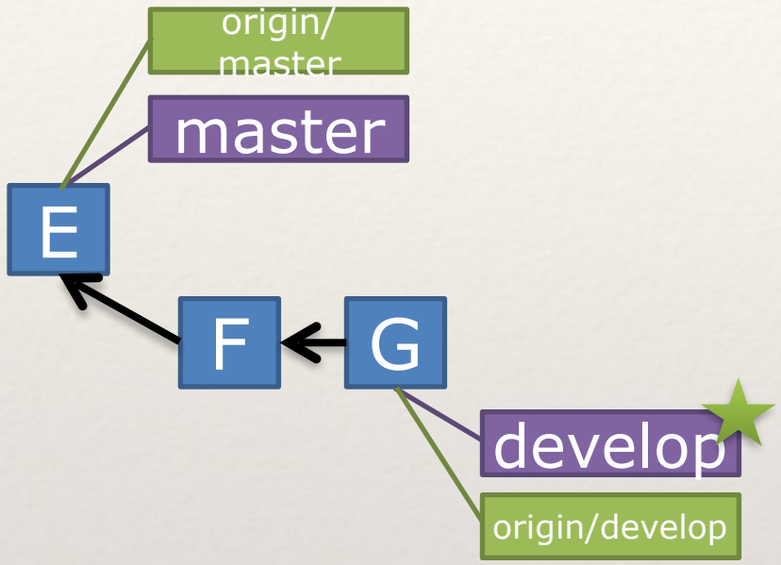
- Now, we switch our local working copy to the local develop branch

Branches Illustrated



- However, someone on our team did some development on this branch also and pushed it to the remote origin/develop branch
- We don't see these changes locally until we pull them down

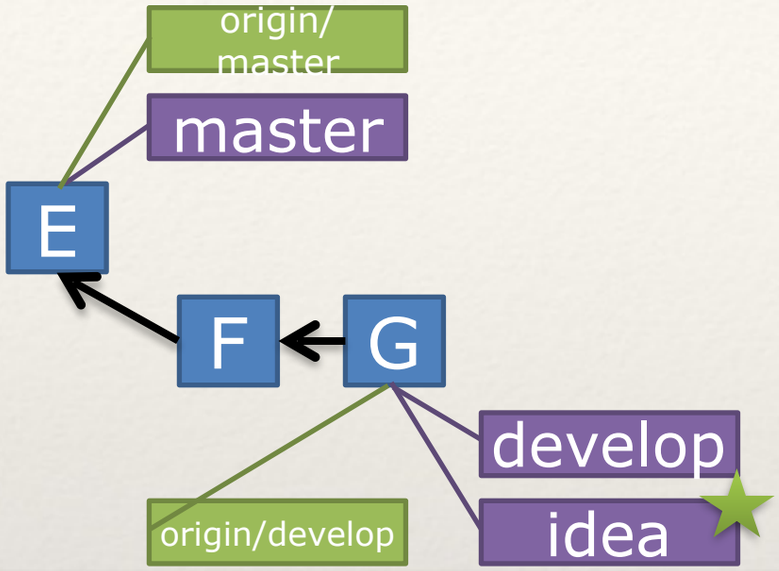
Branches Illustrated




```
> git pull origin develop
```

- So, before we start our work for the day, we pull down the latest changes to origin/develop
- Remember, pull = fetch + merge

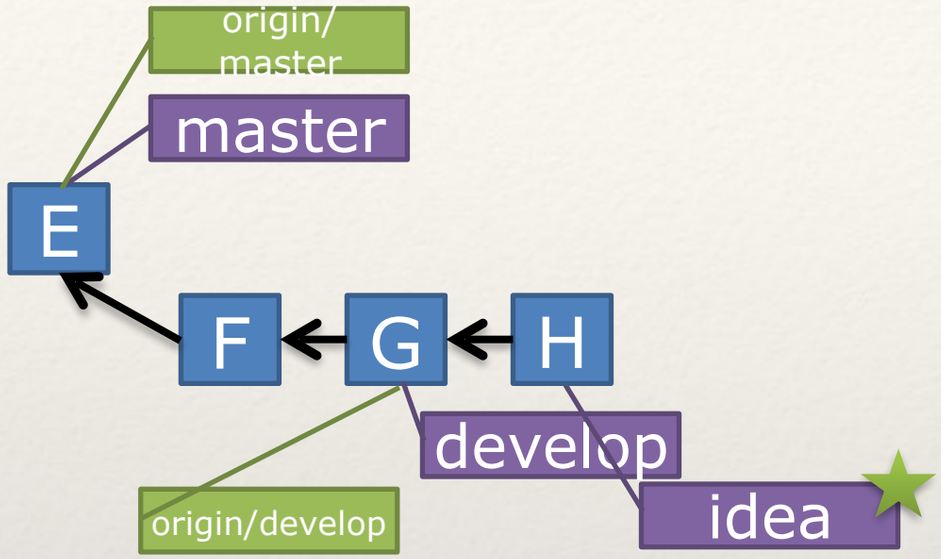
Branches Illustrated




```
> git checkout -b idea
```

- I have a great idea. So, I create a working branch “idea” off of develop

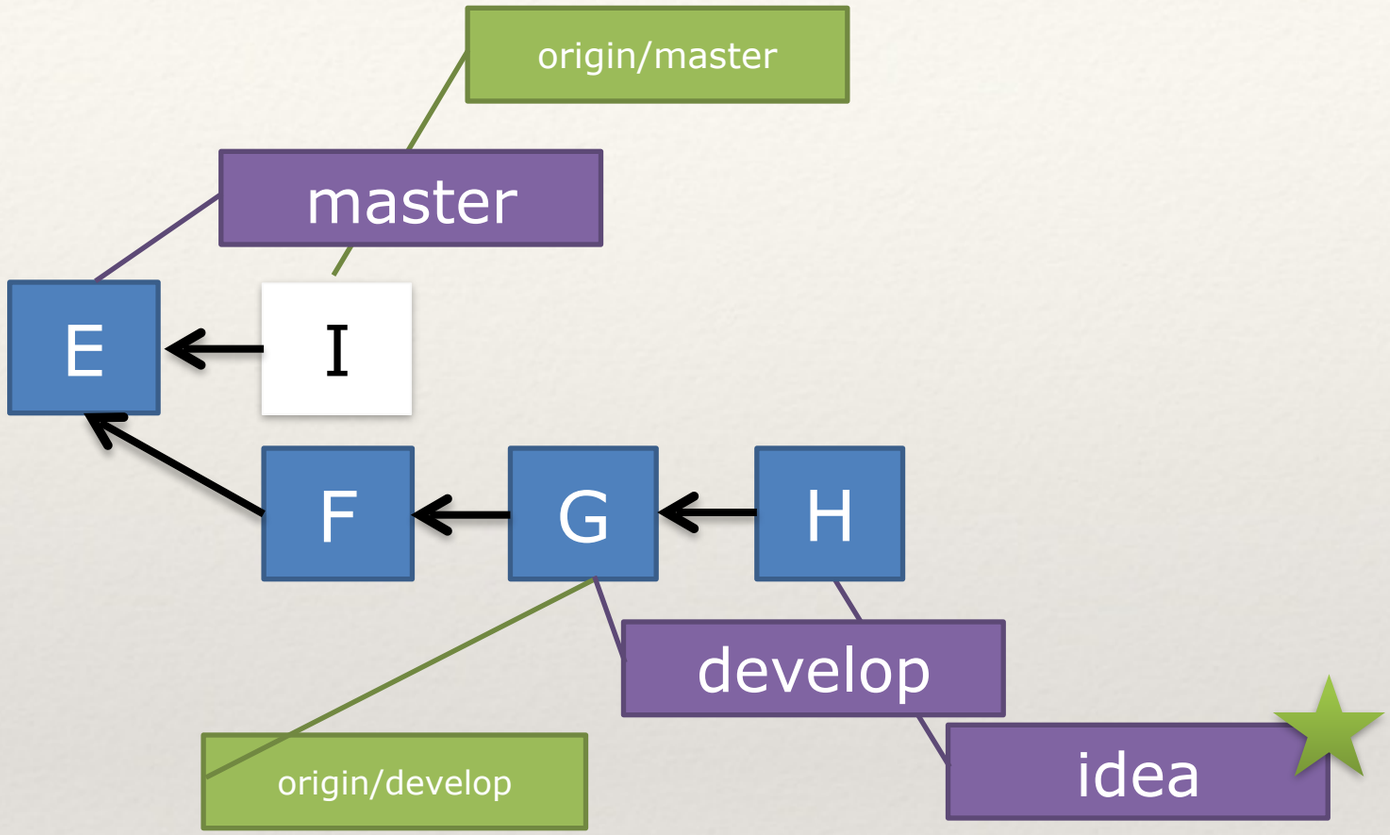
Branches Illustrated




```
> git commit
```

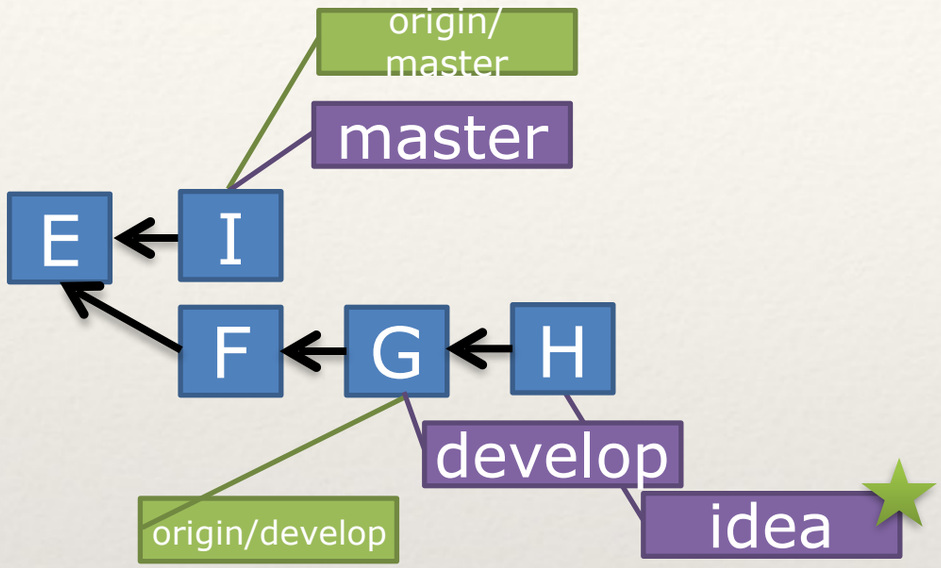
- I did some work in “idea” and moved my local copy along a commit

Branches Illustrated



- In the meantime, one of my group mates did a hot fix to the production "master" branch

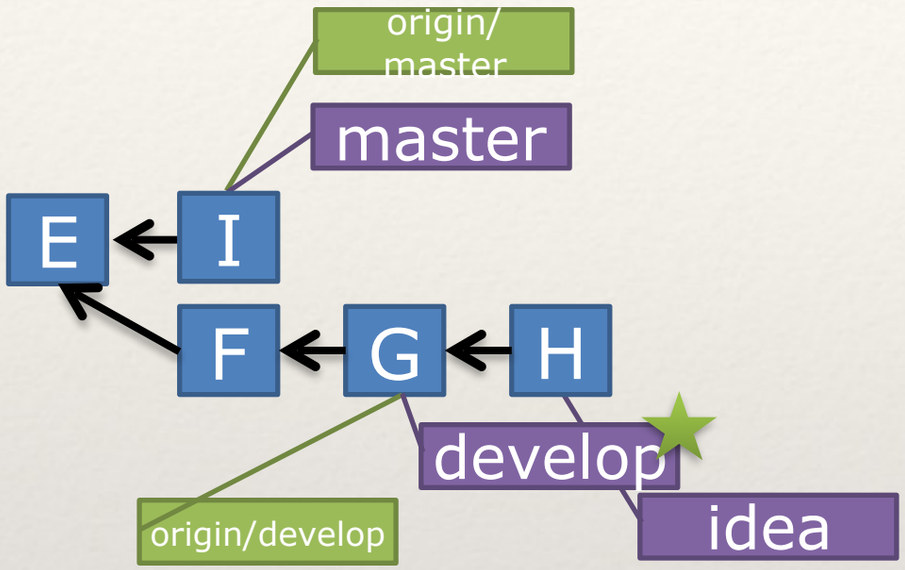
Branches Illustrated




```
> git pull (at least daily)
```

- Since I like to keep up to date, I pull down the changes while I'm working (perhaps I got a notification via email that a hot fix went in)

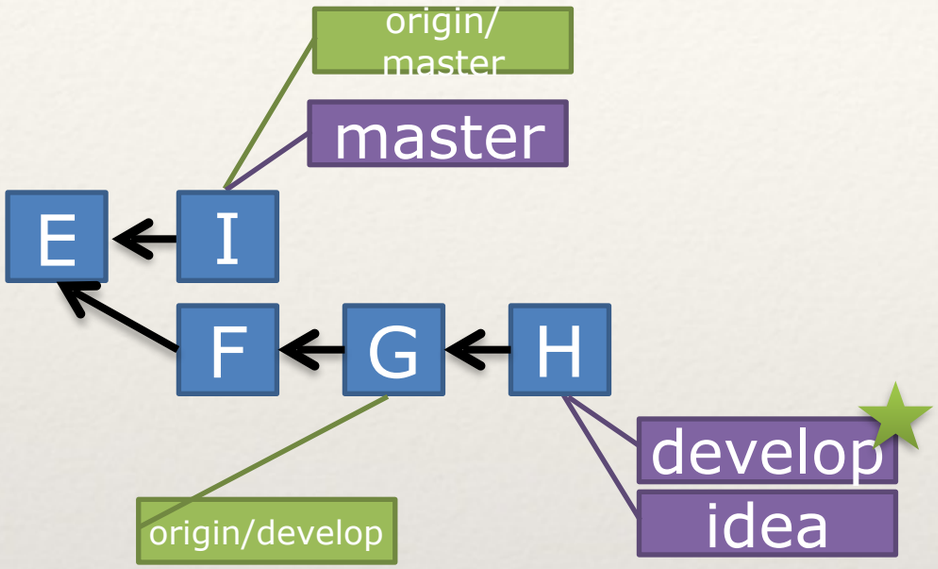
Branches Illustrated




```
> git checkout develop
```

- Now, I'm going to merge "idea" back into develop, so I switch back my local working copy to the develop branch

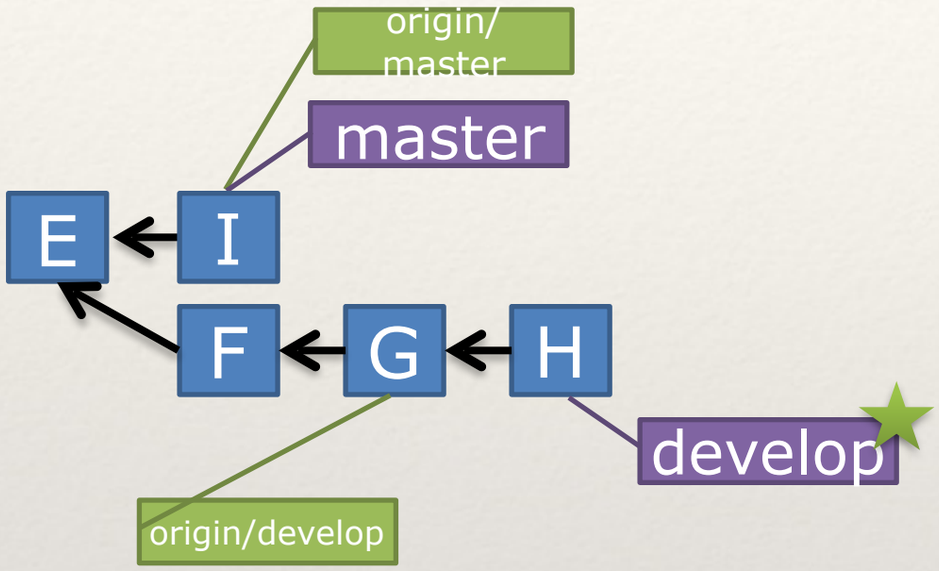
Branches Illustrated




```
> git merge idea (fast forward merge)
```

- Now, I merged my idea changes into the local develop branch. Since there are no other changes to develop, this is called a “fast forward” merge

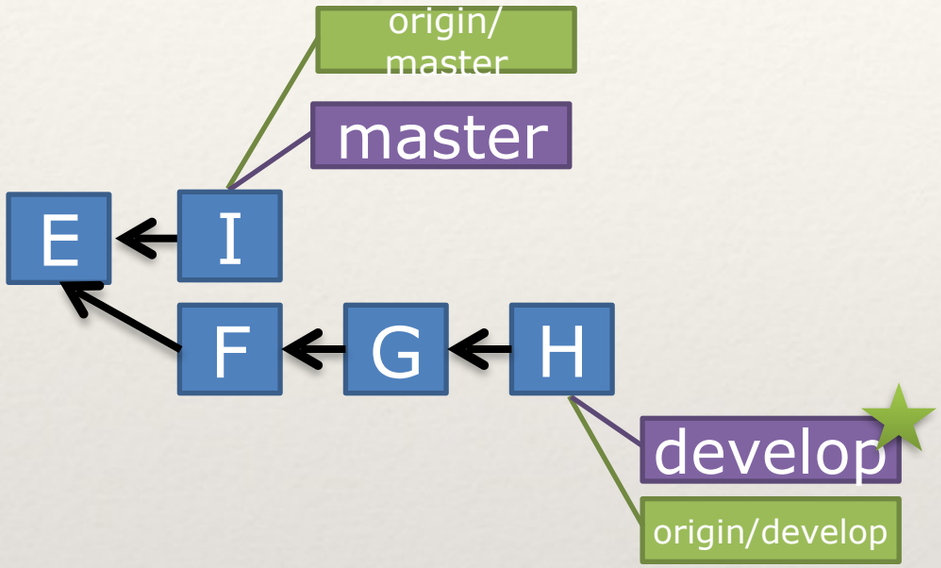
Branches Illustrated




```
> git branch -d idea
```

- I clean up my idea branch (delete it)

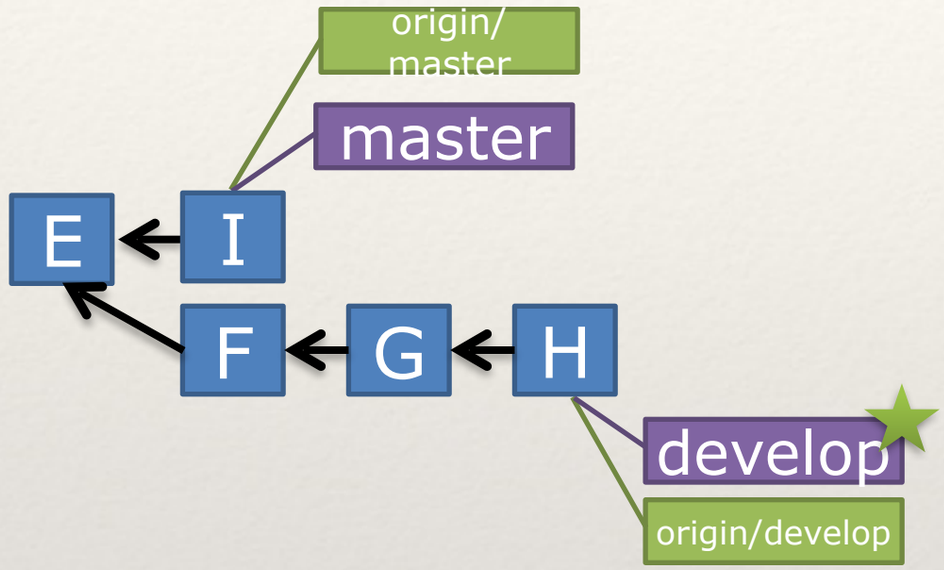
Branches Illustrated




```
> git push origin develop
```

- And share my latest changes of develop to the remote (origin) repo

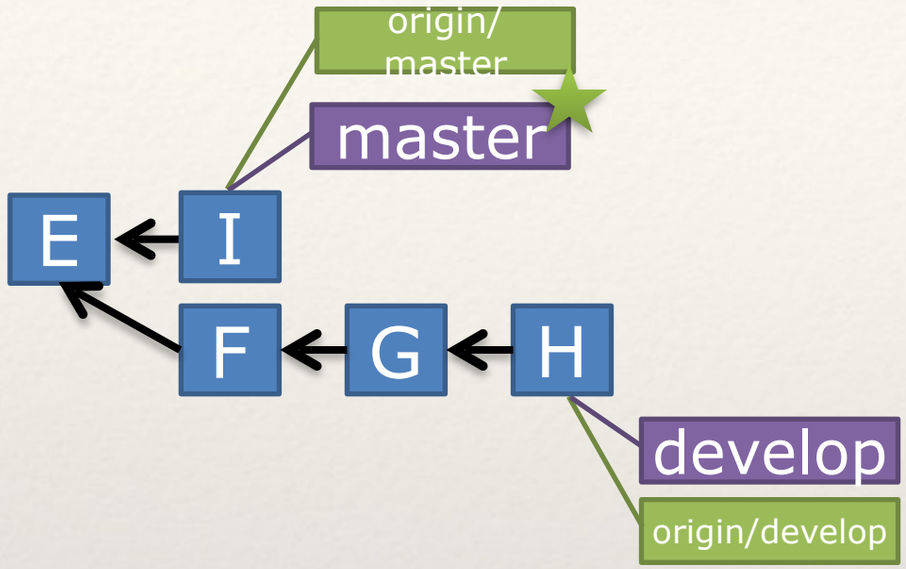
Merge vs. Rebase Workflows




```
> git push origin develop
```

- I'm now ready to move the develop branch to production (master)
  - I can either use a:
    - merge workflow
    - or rebase workflow

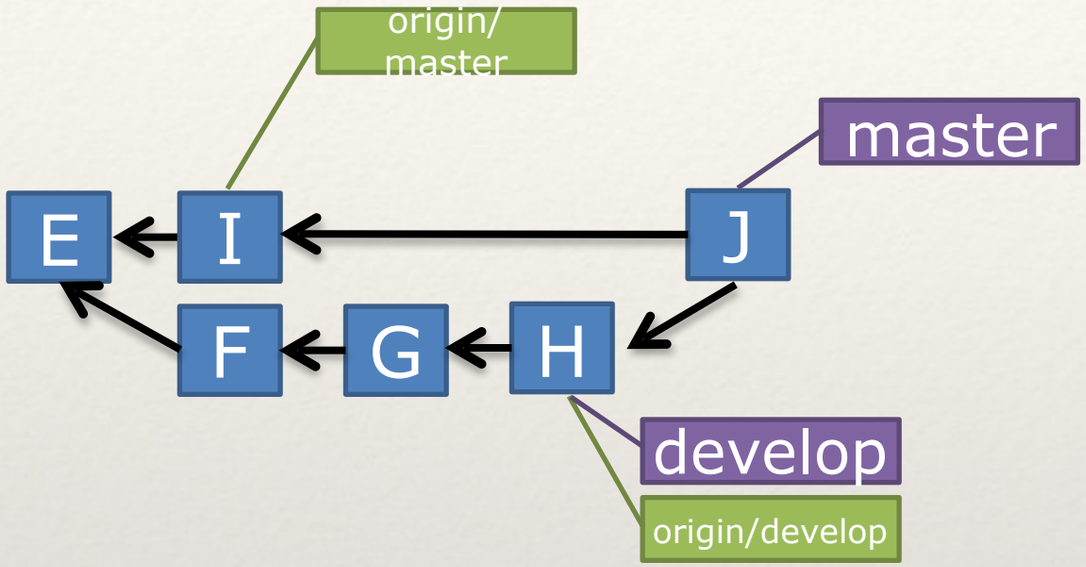
Merge Flow




```
> git checkout master
```

- Move local working copy to master

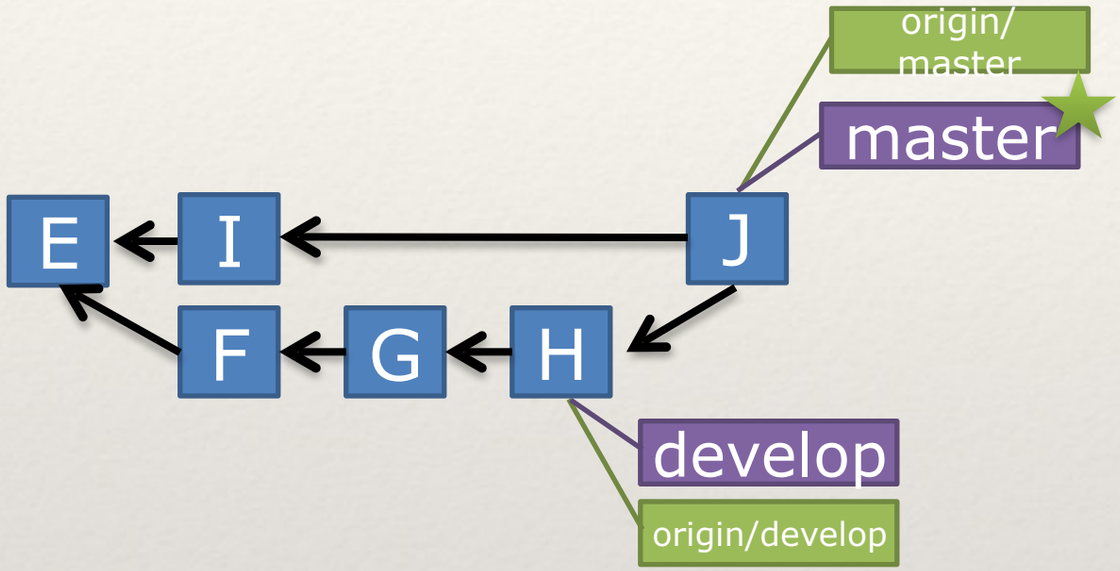
Merge Flow




```
> git merge develop
```

- Merge develop into master

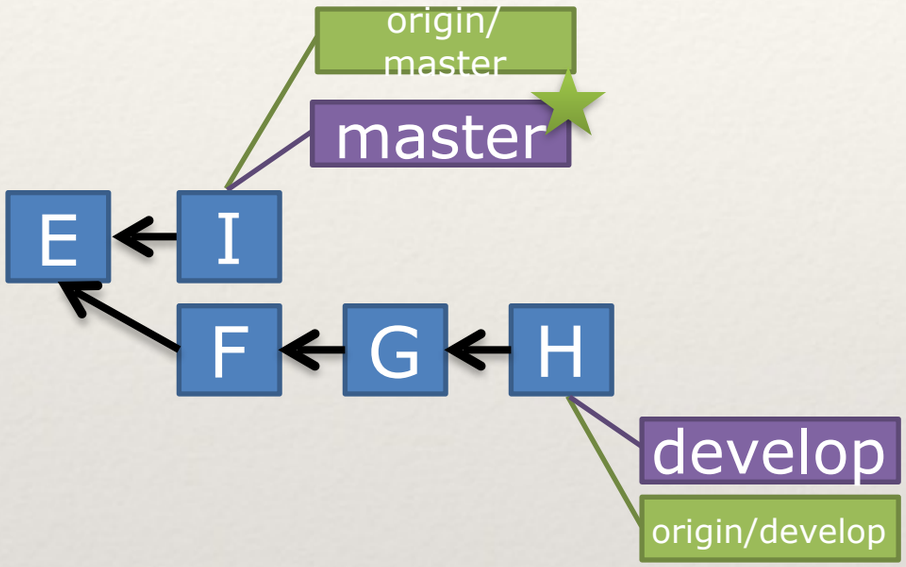
Merge Flow




```
> git push origin
```

- Push my local master changes to origin/master

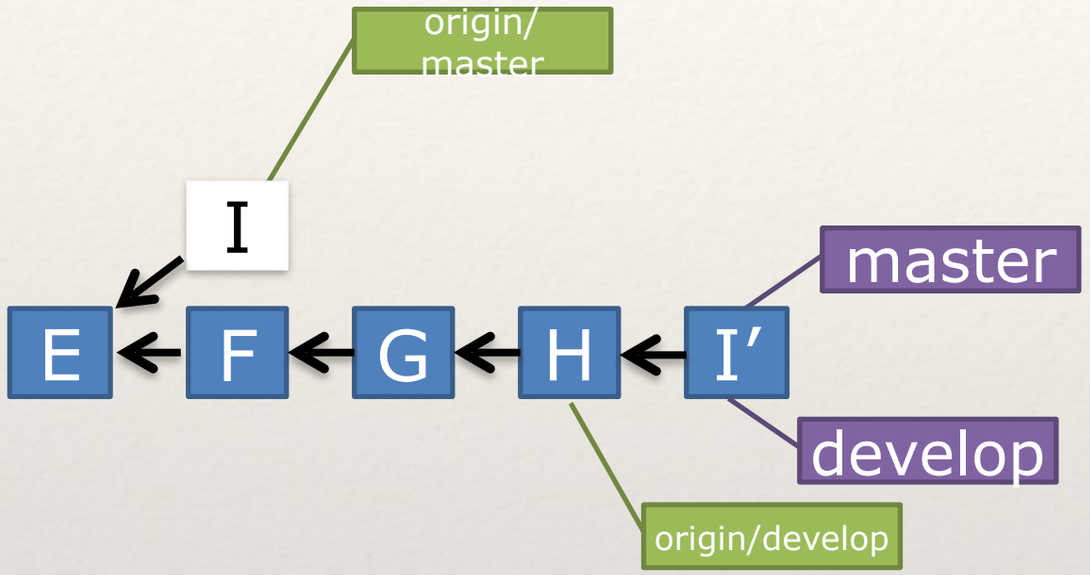
Rebase Flow




```
> git checkout master
```

- Move onto master

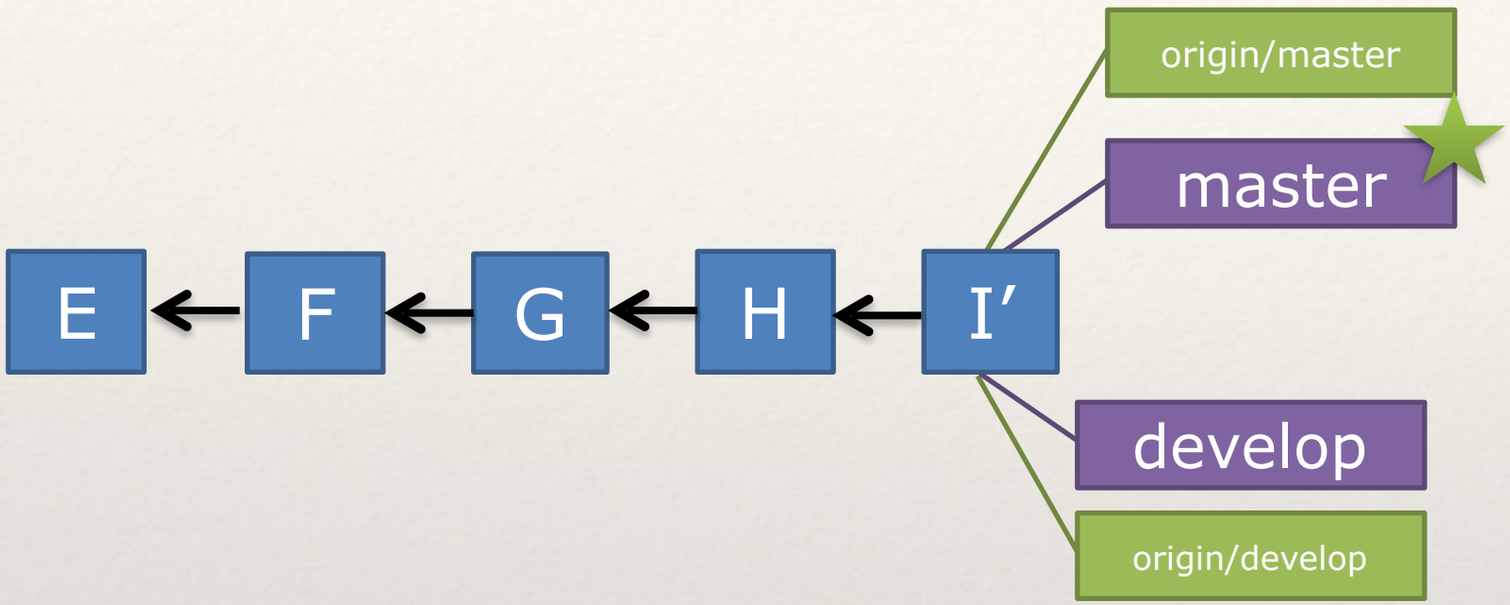
Rebase Flow




```
> git rebase develop
```

- Rebase develop onto master

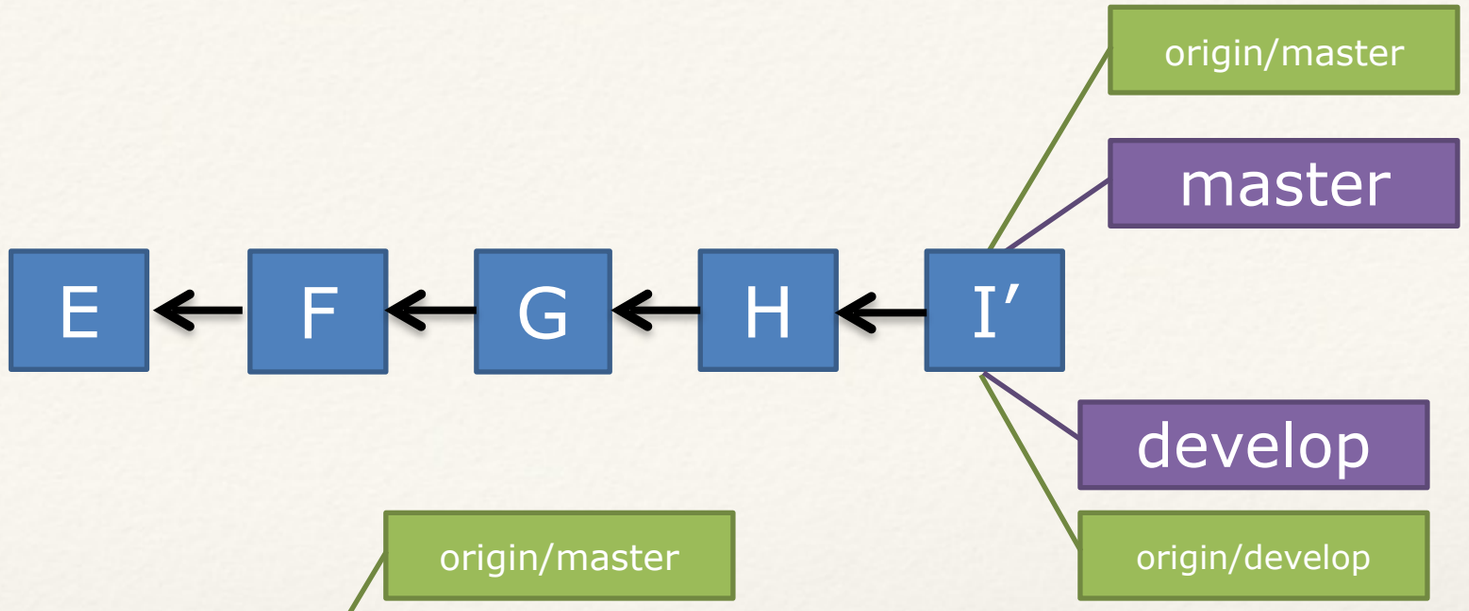
Rebase Flow



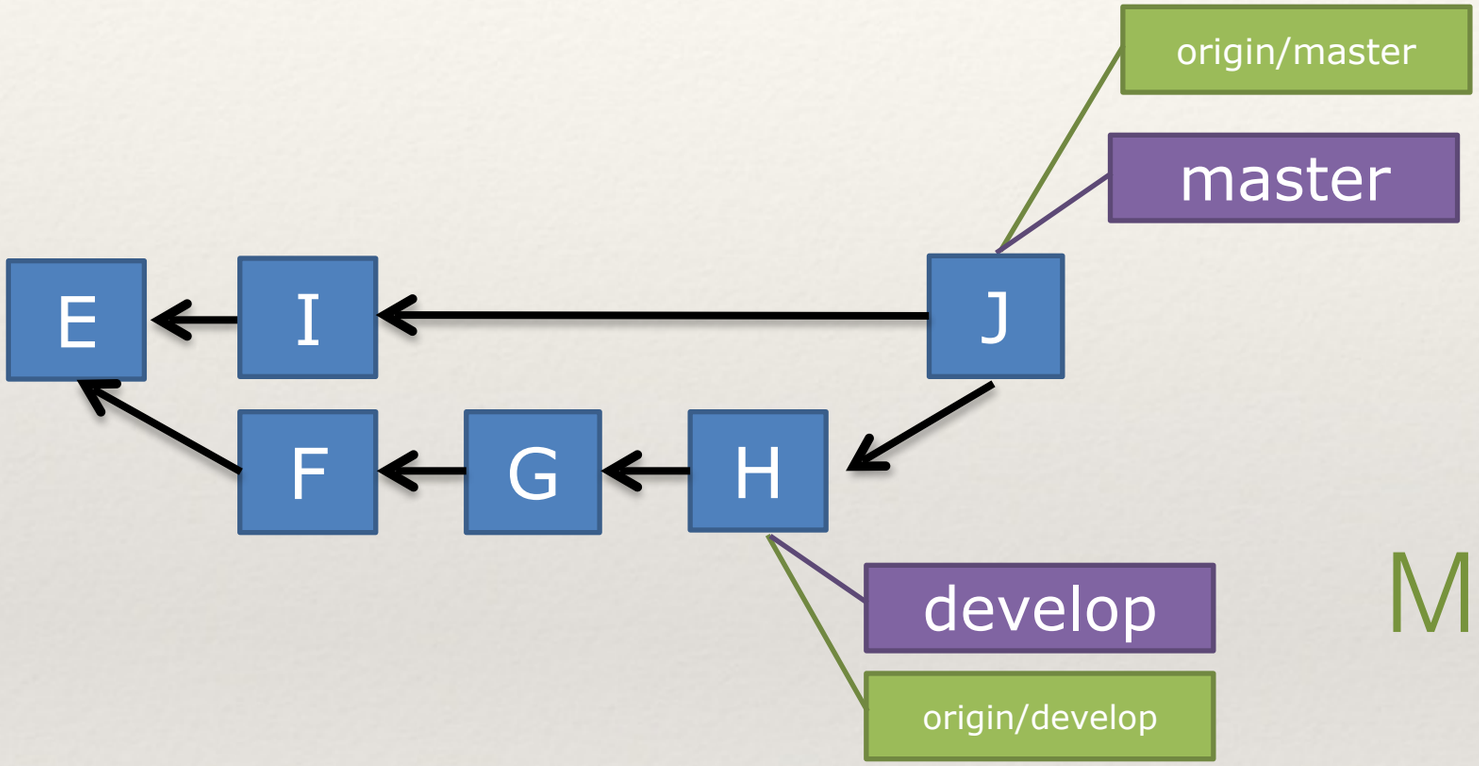

```
> git push origin
```

- Get origin up to date

Rebase Flow



Merge Flow



- As with all things Git, there are multiple ways of doing things
- Using a merge workflow rather than Rebase is a matter of preference
- Rebase looks cleaner and in large projects is usually desired
- Rebase workflow is an example of rewriting history in git

Questions?

- CVS
  - FAQ:http://ximbiot.com/cvs/wiki/index.php?title=CVS_FAQ
  - Manual: http://ximbiot.com/cvs/manual/
- Subversion
  - http://subversion.tigris.org/
  - http://svnbook.red-bean.com/