TACC Training: git

Dr. Antía Lamas-Linares alamas@tacc.utexas.edu
Dr. Damon McDougall dmcdougall@tacc.utexas.edu

What is this?

- Absolute beginner's git
  - Git as a single user
  - Synchronizing your work across multiple machines
  - Backups, who needs backups? … EVERYONE
- Advanced git
  - Collaborators
  - Multiple remotes
  - Complex merges
  - Best practices

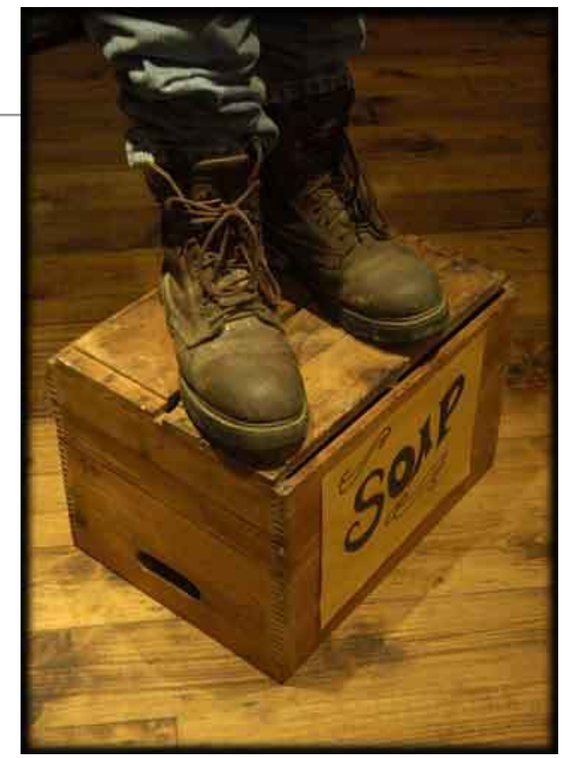
Version Control

- 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

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

Open a terminal and log in to Stampede2

```
[alamas@dhcp-146-6-176-48 [3] ssh alamas@stampede2.tacc.utexas.edu
To access the system:

1) If not using ssh-keys, please enter your TACC password at the password prompt
2) At the TACC Token prompt, enter your 6-digit code followed by <return>.

[Password:
[TACC Token Code:
Last login: Fri Sep 28 11:58:20 2018 from 134.134.139.93
--------------------------------------------------------------------------------
                    Welcome to the Stampede2 Supercomputer
      Texas Advanced Computing Center, The University of Texas at Austin
--------------------------------------------------------------------------------
```

Create a directory with a couple of files and follow along

Creating a local repo: init

```
[alamas@Sstaff(77)$ ls                                                          ]
file1  file2
[alamas@Sstaff(78)$ git init                                                    ]
Initialized empty Git repository in /scratch/03302/alamas/presentations/git-tut/.git/
[alamas@Sstaff(79)$ git status                                                  ]
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1
        file2

nothing added to commit but untracked files present (use "git add" to track)
alamas@Sstaff(80)$
```

The repo structure exists but nothing else

Local workflow: add and commit, status and log

```
[alamas@Sstaff(80)$ git add file1 file2                                    ]
You have mail in /var/spool/mail/alamas
[alamas@Sstaff(81)$ git status                                             ]
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file1
        new file:   file2
```

```
[alamas@Sstaff(83)$ git commit -m "the start of a great repo"              ]
 [master (root-commit) 1152c7f] the start of a great repo
  2 files changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 file1
  create mode 100644 file2
[alamas@Sstaff(84)$ git status                                             ]
On branch master
nothing to commit, working directory clean
[alamas@Sstaff(85)$ git log                                                ]
commit 1152c7fc772876cb5325fc7b14ce5672326e0fb6
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 08:38:37 2018 -0500

    the start of a great repo
```

Changing your mind: revert

```
[alamas@Sstaff(153)$ git log                                                    ]
commit 86e563f4a8d24f3bc00d8ad6b812e781936e21d6
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 15:55:09 2018 -0500

    important changes of the second kind

commit 5a9c2cd537ca4bd96718c9c2f02fcde29b940317
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 15:54:06 2018 -0500

    important file added

commit 7ce0e0eea4a14c910aeaddc4a275aa51688235a5
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 13:46:03 2018 -0500

    changes in feature1 branch

commit 1152c7fc772876cb5325fc7b14ce5672326e0fb6
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 08:38:37 2018 -0500

    the start of a great repo
alamas@Sstaff(154)$
```

Changing your mind: revert

I made a terrible error in my commits …

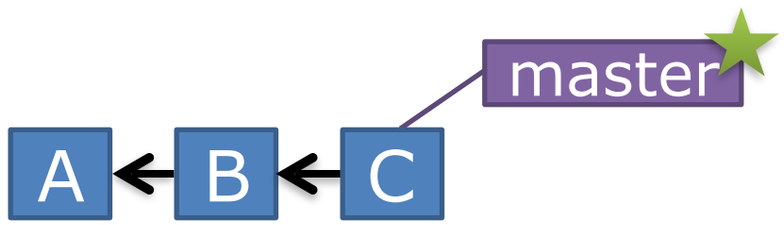
```
[alamas@Sstaff(181)$ git revert HEAD                                    ]
 [master fda1654] Revert "modification2"
  1 file changed, 1 deletion(-)
[alamas@Sstaff(182)$ git log                                            ]
commit fda16548a5210456379f0f3b325fdcfbd8e46bb3
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 16:37:17 2018 -0500

    Revert "modification2"

    This reverts commit b4c9fa473a64fbf43378e06986d29c829dc5abfe.

commit b4c9fa473a64fbf43378e06986d29c829dc5abfe
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 16:31:47 2018 -0500

    modification2

commit d76afe45b02b945173e68e080fdeea0df86b2b58
Author: alamas <alamas@tacc.utexas.edu>
Date:   Fri Oct 12 16:31:25 2018 -0500

    modification1
```

Branching out: branch


```
> 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.

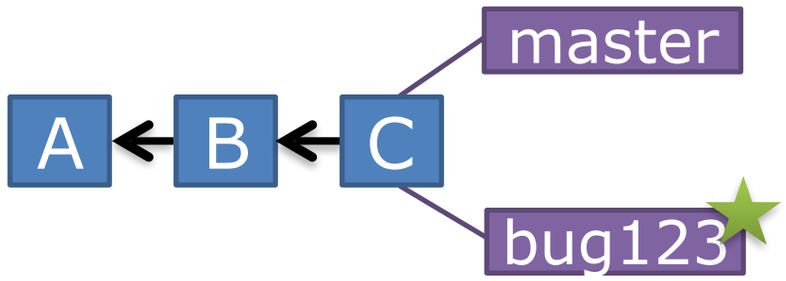
Branches Illustrated




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

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

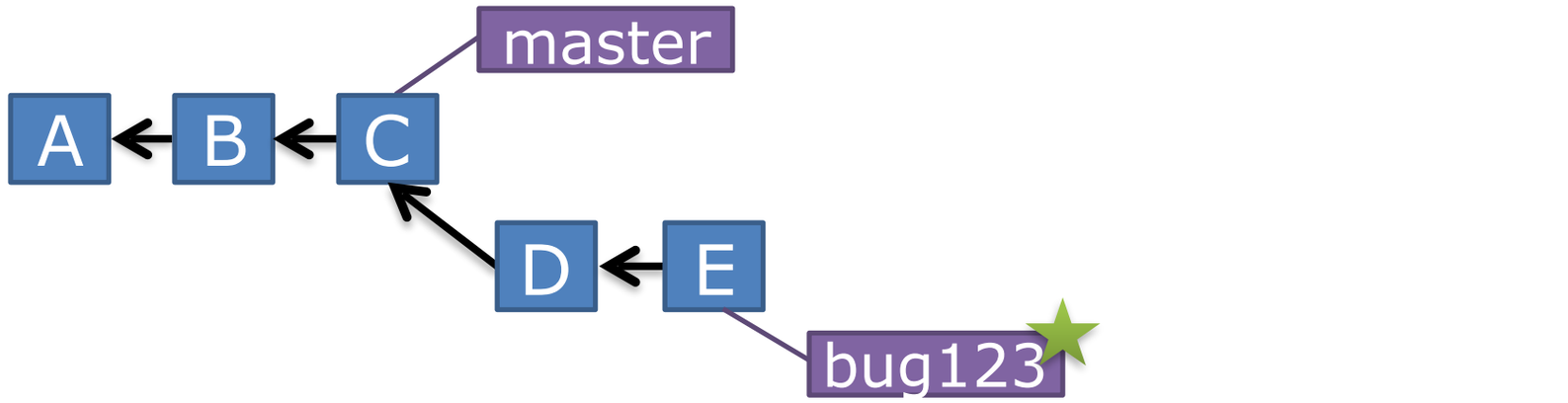
Branches Illustrated




```
> 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

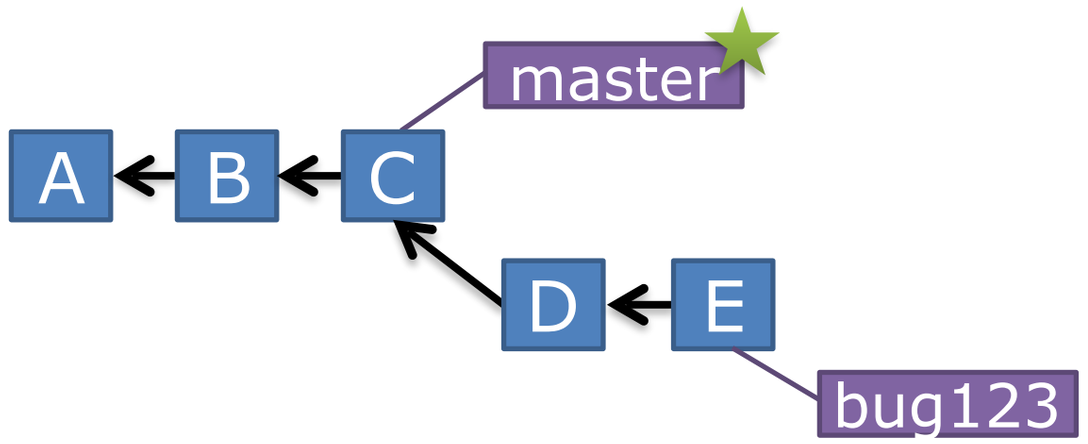
Branches Illustrated



> git commit (x2)

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

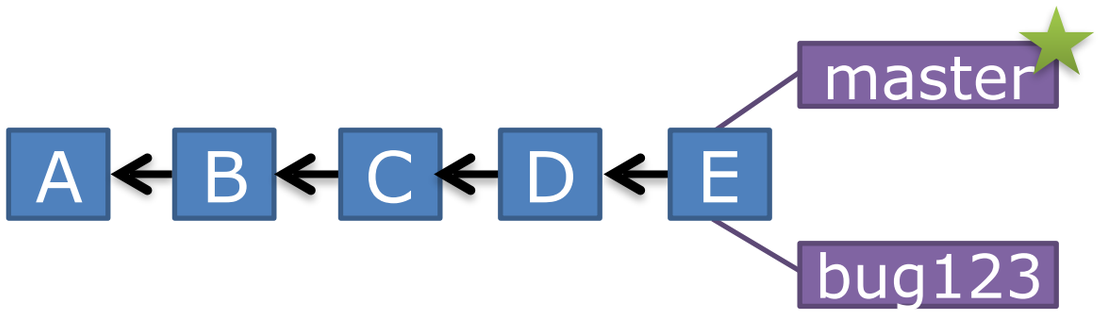
Branches Illustrated



> 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

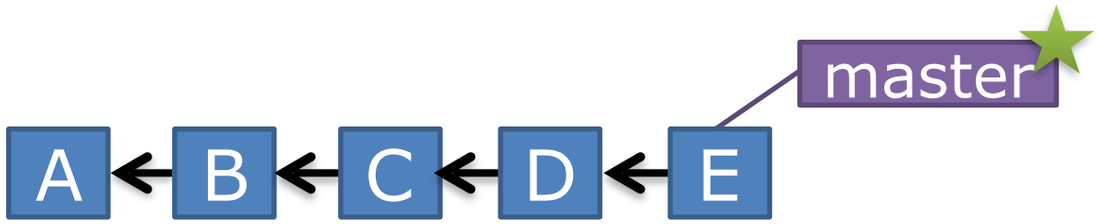
Branches Illustrated




```
> 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 any other repos.

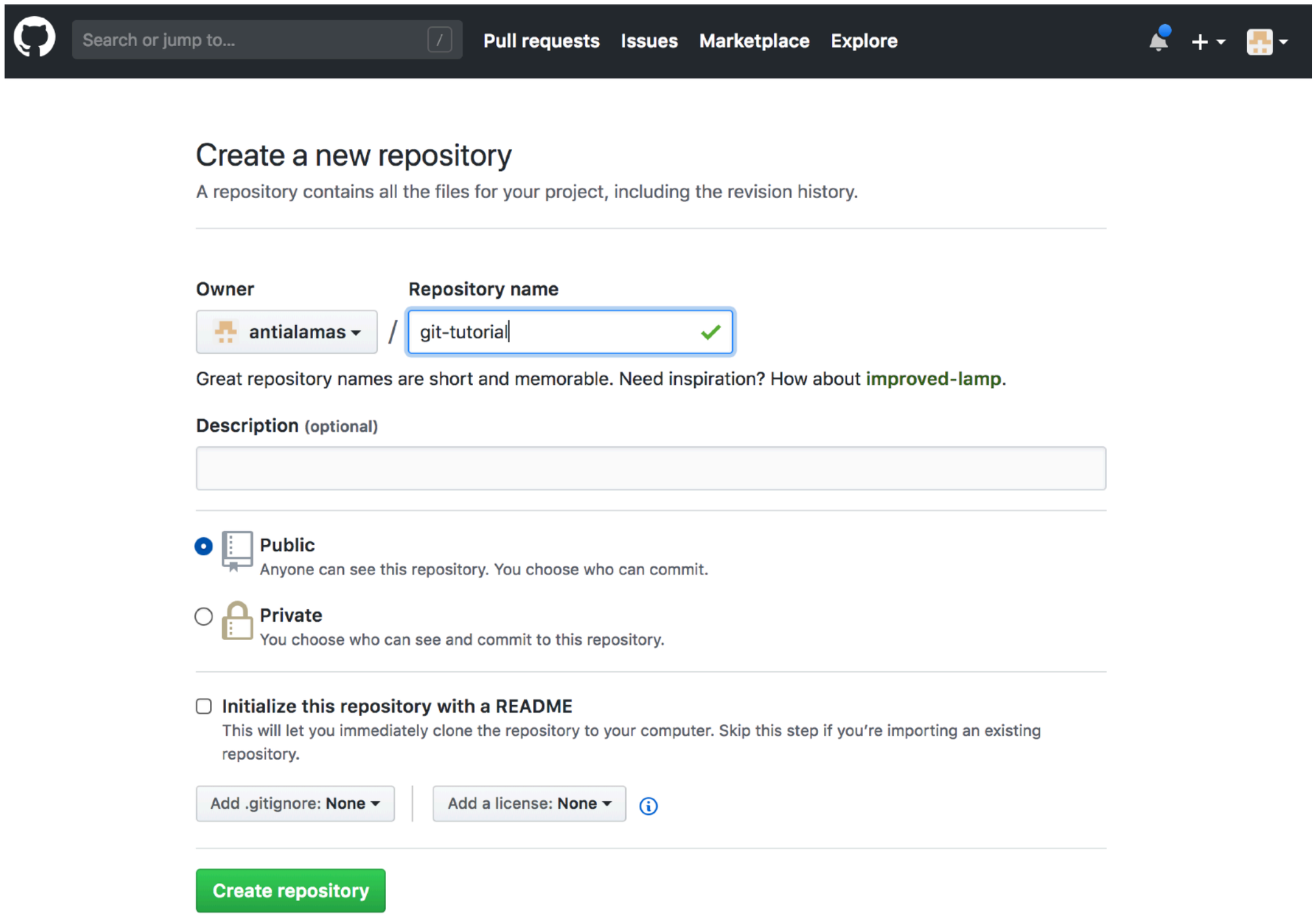
Branches Illustrated

```
alamas@Sstaff(20)$ git checkout -b bug123
Switched to a new branch 'bug123'
alamas@Sstaff(21)$ vim file1
alamas@Sstaff(22)$ git add file1
alamas@Sstaff(23)$ git commit -m "a fix for bug123"
[bug123 0ffab53] a fix for bug123
 1 file changed, 1 insertion(+)
alamas@Sstaff(24)$ git status
On branch bug123
nothing to commit, working directory clean
alamas@Sstaff(25)$ git checkout master
Switched to branch 'master'
alamas@Sstaff(26)$ git merge bug123
Updating fda1654..0ffab53
Fast-forward
 file1 | 1 +
 1 file changed, 1 insertion(+)
alamas@Sstaff(27)$ git status
On branch master
nothing to commit, working directory clean
alamas@Sstaff(28)$
```

Creating a remote repo: Why??

- Tracking changes in a single place is good …
  - Synchronizing across multiple machines is BETTER
    - Takes the guess work out of status at different locations
    - Incidentally, it is also a great BACKUP strategy
    - If you need to set up a new working location, it is trivial!

Creating a remote repo (github)



Creating a remote repo (github)

antialamas / git-tutorial

Unwatch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/antialamas/git-tutorial.git

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# git-tutorial" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/antialamas/git-tutorial.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/antialamas/git-tutorial.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Creating a remote repo (github)

```
[alamas@Sstaff(30)$ git remote add origin https://github.com/antialamas/git-tutorial.git
[alamas@Sstaff(31)$ git push -u origin master
[Username for 'https://github.com': antialamas
[Password for 'https://antialamas@github.com':
 remote: Invalid username or password.
 fatal: Authentication failed for 'https://github.com/antialamas/git-tutorial.git/'
[alamas@Sstaff(32)$ git push -u origin master
[Username for 'https://github.com': antialamas
[Password for 'https://antialamas@github.com':
 Counting objects: 23, done.
 Delta compression using up to 28 threads.
 Compressing objects: 100% (16/16), done.
 Writing objects: 100% (23/23), 1.88 KiB | 0 bytes/s, done.
 Total 23 (delta 6), reused 0 (delta 0)
 remote: Resolving deltas: 100% (6/6), done.
 remote:
 remote: Create a pull request for 'master' on GitHub by visiting:
 remote:      https://github.com/antialamas/git-tutorial/pull/new/master
 remote:
 To https://github.com/antialamas/git-tutorial.git
  * [new branch]      master -> master
 Branch master set up to track remote branch master from origin.
```

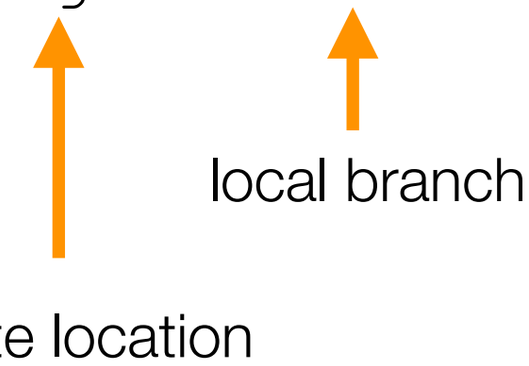
Do yourself a favor and set up ssh keys …

```
[alamas@Sstaff(39)$ git remote set-url origin git@github.com:antialamas/git-tutorial.git
[alamas@Sstaff(40)$ git remote -v
 origin  git@github.com:antialamas/git-tutorial.git (fetch)
 origin  git@github.com:antialamas/git-tutorial.git (push)
```

Remote repo: What is going on?

- Remote location called “origin” (default name), could be something else.
- When local repo is in a state we like to preserve, we push the changes upstream to the remote

```
> git push origin master
```



Changing working location: clone

Leaving the office for a late night of extra work …

```
[alamas@Sstaff(57)$ cd git-tut-home/
/scratch/03302/alamas/presentations/git-tut-home
[alamas@Sstaff(58)$ git clone https://github.com/antialamas/git-tutorial.git
Cloning into 'git-tutorial'...
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 26 (delta 6), reused 26 (delta 6), pack-reused 0
Unpacking objects: 100% (26/26), done.
Checking connectivity... done.
[alamas@Sstaff(59)$ ls
git-tutorial
[alamas@Sstaff(60)$ cd git-tutorial/
/scratch/03302/alamas/presentations/git-tut-home/git-tutorial
[alamas@Sstaff(61)$ ls
file1  file2  file3
[alamas@Sstaff(62)$ git log -n 3
commit c0efcf459e53d771e5d8fdf75ea0fa17274cbcec
Author: antialamas <alamas@tacc.utexas.edu>
Date:   Sun Oct 14 21:28:05 2018 -0500

    some additions in documentation of file2
```

Workflow local-remote

Before closing up the session in one location:

```
> git add <whatever files>
> git commit -m “last edits of the day”
> git push origin master
```

Before even thinking about editing a file of existing repo in new location:

```
> git pull origin master
```

This ensures that you will be able to transition seamlessly from one location to another

It is even more critical if you are collaborating with others

Special commits - tags

It can be useful to mark special points in the project: e.g. released version, submitted, …

```
> git tag -a “sub_prl” -m “Submitted PRL”
```

By default, tags do NOT get pushed to the remote repo; the tags flag uploads all the info

```
> git push origin --tags
```

Going to a specific tag works the same as a branch:

```
> git checkout sub_prl
```

Special commits - tags

```
alamas@Sstaff(79)$ git checkout sub_rmp
Note: checking out 'sub_rmp'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at c0efcf4... some additions in documentation of file2
```

If you do decide to work from a tag location, create a new branch!

Basic individual workflow

- Track any project in development
- Create a remote to provide synchronization and backup
- Use local branches and merging to develop from stable positions
- Tags are useful for marking special points in a project

init, add, commit, status, log, revert, checkout, merge, remote, push, pull, tag

TACC Training: Advanced git

Dr. Antía Lamas-Linares alamas@tacc.utexas.edu
Dr. Damon McDougall dmcdougall@tacc.utexas.edu

What can you expect to get out of this?

- `git` has a lot of features, and you probably won't use all of them—you probably won't use 15% of them
- Enough knowledge to set up a repo with colleagues and work together on a project
- Resources to refresh your memory—these slides
- Resources to point you to more advance material for self-learning—stack overflow/google/colleagues
- Version control is not a replacement for speaking to your colleagues

What we're going to cover

- Customising `git`
- Merging branches and conflict resolution
- More complicated merges and merging workflows
- Multiple remotes (collaborators)
- Some (opinionated) best practices

Introduce yourself to git

```
damon@quagmire:(master) ~/CoolProject
$ git config --global user.name "Damon McDougall"
damon@quagmire:(master) ~/CoolProject
$ git config --global user.email "dmcdougall@tacc.utexas.edu"
```

- Commits can contain information like the username and email address of the person that made the commit.
- This sets a global email address for all repos on my machine
- Working on a repo for work? Then you can use your work email address for that repo. There is a `--local` flag. This configures only the repo my current working directory is in

git aliases

- You can set aliases for commands you type often

```
damon@dhcp-146-6-176-221:(master) ~/tacc/training
$ git config --global alias.s "status"
damon@dhcp-146-6-176-221:(master) ~/tacc/training
$ git config --global alias.d "diff"
damon@dhcp-146-6-176-221:(master) ~/tacc/training
$ git s
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file1

no changes added to commit (use "git add" and/or "git commit -a")
damon@dhcp-146-6-176-221:(master) ~/tacc/training
$ git d
diff --git a/file1 b/file1
index e69de29..e965047 100644
--- a/file1
+++ b/file1
@@ -0,0 +1 @@
+Hello
```

Ignoring files

- A lot of untracked files can clutter up `git status`. You can tell `git` to ignore files

- The `.gitignore` file lives at the root of the repository

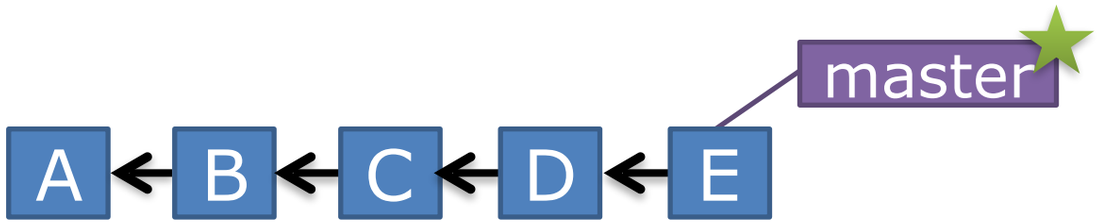
- Commit it just like any other file

```
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ git s
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be

        temp_file

nothing added to commit but untracked files present (
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ echo "temp_file" >> .gitignore
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ git add .gitignore
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ git commit -m "Make git ignore temp_file"
[master aef2bc6] Make git ignore temp_file
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ git s
On branch master
nothing to commit, working tree clean
```

Branching

- `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”

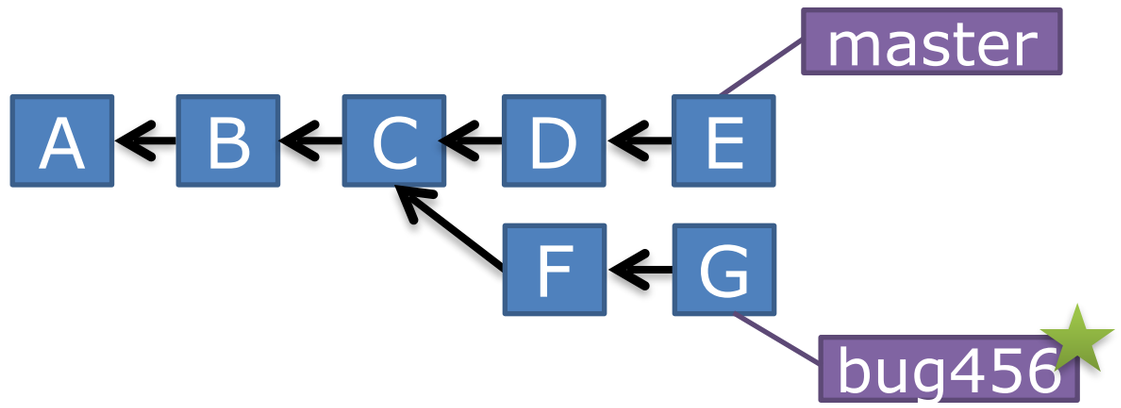
Branches Illustrated — this is where Antía ended




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

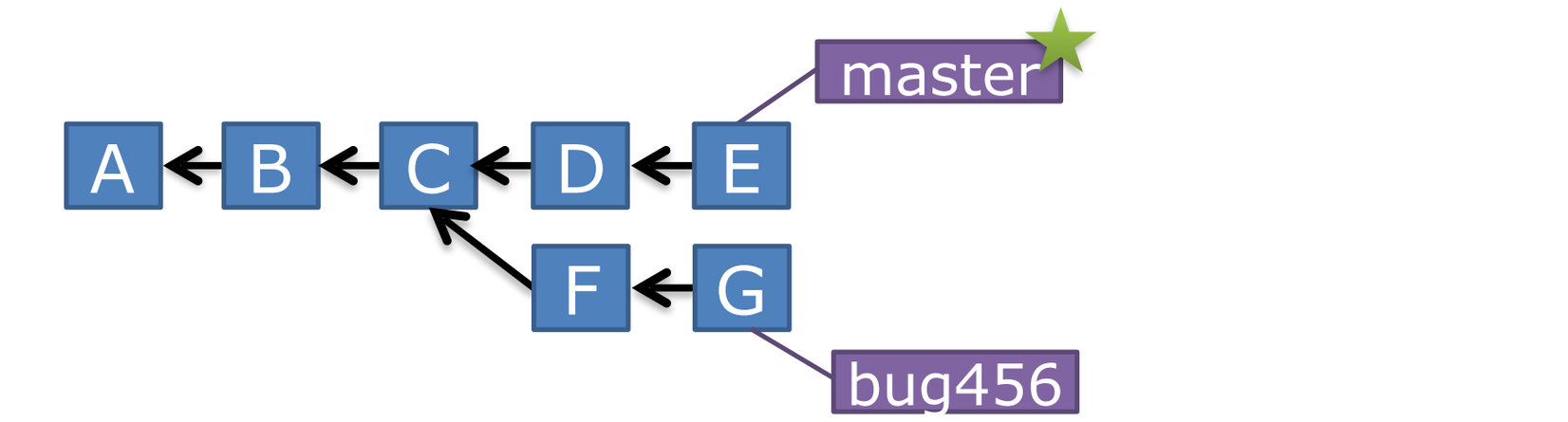
- And since we're done with the story branch, we can delete it. This all happened locally, without affecting anyone upstream

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`

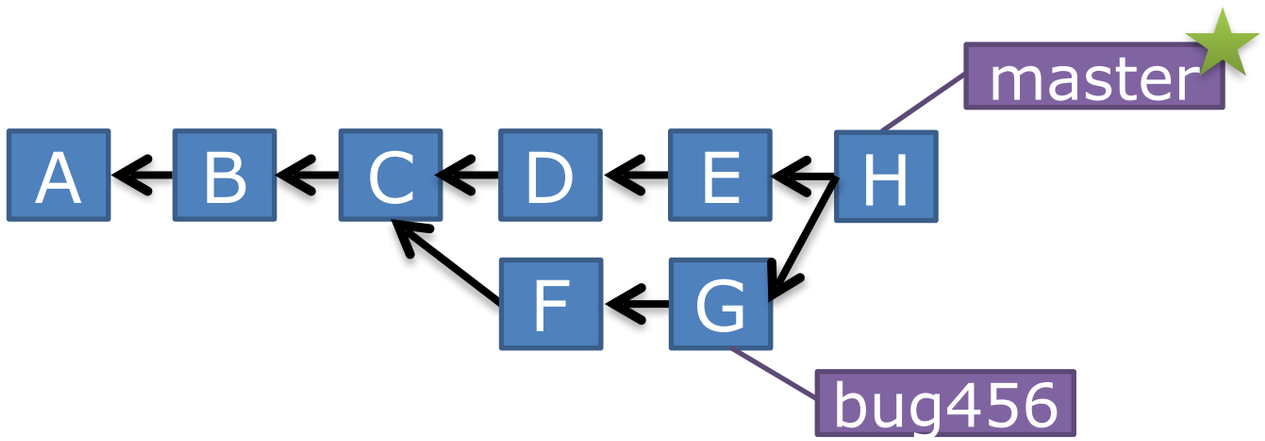
Branches Illustrated



> 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

Merge conflicts

- Merging happens in the *current* branch. Always know what your current branch is!

- `git` will tell you if there are conflicts:

```
damon@dhcp-146-6-176-221:(master) ~/tacc/test_repo
$ git merge bug456
Auto-merging file2
CONFLICT (content): Merge conflict in file2
Automatic merge failed; fix conflicts and then commit the result.
damon@dhcp-146-6-176-221:(master|MERGING) ~/tacc/test_repo
$ git s
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   file2

no changes added to commit (use "git add" and/or "git commit -a")
```

Merge conflicts

- Look at the file with the conflict(s)
- Conflict markers:
  - `<<<<<<<`
  - `=======`
  - `>>>>>>>`

```
damon@dhcp-146-6-176-221:(master|MERGING) ~/tacc/test_repo
$ cat file2
Here is some
text I
have written.
This could be C
code or
perhaps
<<<<<<< HEAD
C++ or LaTeX, who
=======
FORTRAN or LaTeX, who
>>>>>>> bug456
knows.

This is the second paragraph where we talk
about some new things.
```

Merge conflicts

- Conflict markers tell you what change came from where
- Between <<<<<<< and =======
  - This is from `HEAD` (aka where we are *now*, `master`)
- Between ======= and >>>>>>>
  - This is from branch `bug456`

```
damon@dhcp-146-6-176-221:(master|MERGING) ~/to
$ cat file2
Here is some
text I
have written.
This could be C
code or
perhaps
<<<<<<< HEAD
C++ or LaTeX, who
=======
FORTRAN or LaTeX, who
>>>>>>> bug456
knows.

This is the second paragraph where we talk
about some new things.
```

Resolving merge conflicts

- Edit the file — it is your job to make sure the end result is “correct” in some sense
  - Perhaps the changes in master are wrong and you only want the changes in `bug456`.
  - Perhaps the changes in `bug456` are wrong and you only want the changes in `master`
  - Perhaps both are wrong and the “correct” result is something different.
- How do you know what to do?
- Since you are you are the one that owns both branches, you probably know what the answer is.
- Sometimes you are merging someone else’s branch, though. You need to go and talk to that person and work together to determine what the “correct” result is.
- `git add` and `git commit` the result — and write a good message

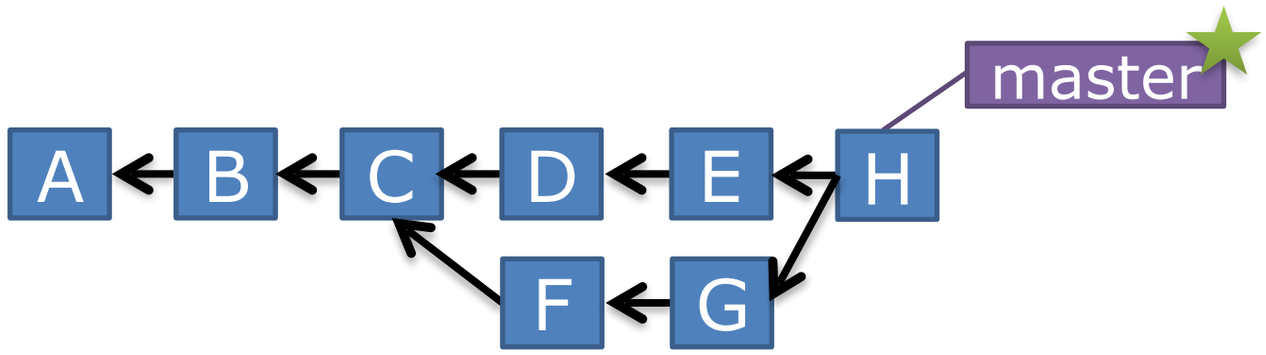
Resolving merge conflicts

- Edit the file to fix it. Make sure to remove the conflict markers; the resulting file should have sensible and well-formed contents.

- The commit is special it is a merge commit

```
damon@dhcp-146-6-176-221:(master|MERGING) ~/tacc/test_repo
$ git add file2
damon@dhcp-146-6-176-221:(master|MERGING) ~/tacc/test_repo
$ git s
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   file2

damon@dhcp-146-6-176-221:(master|MERGING) ~/tacc/test_repo
$ git commit
[master 4dcb6c3] Merge branch 'bug456'
```

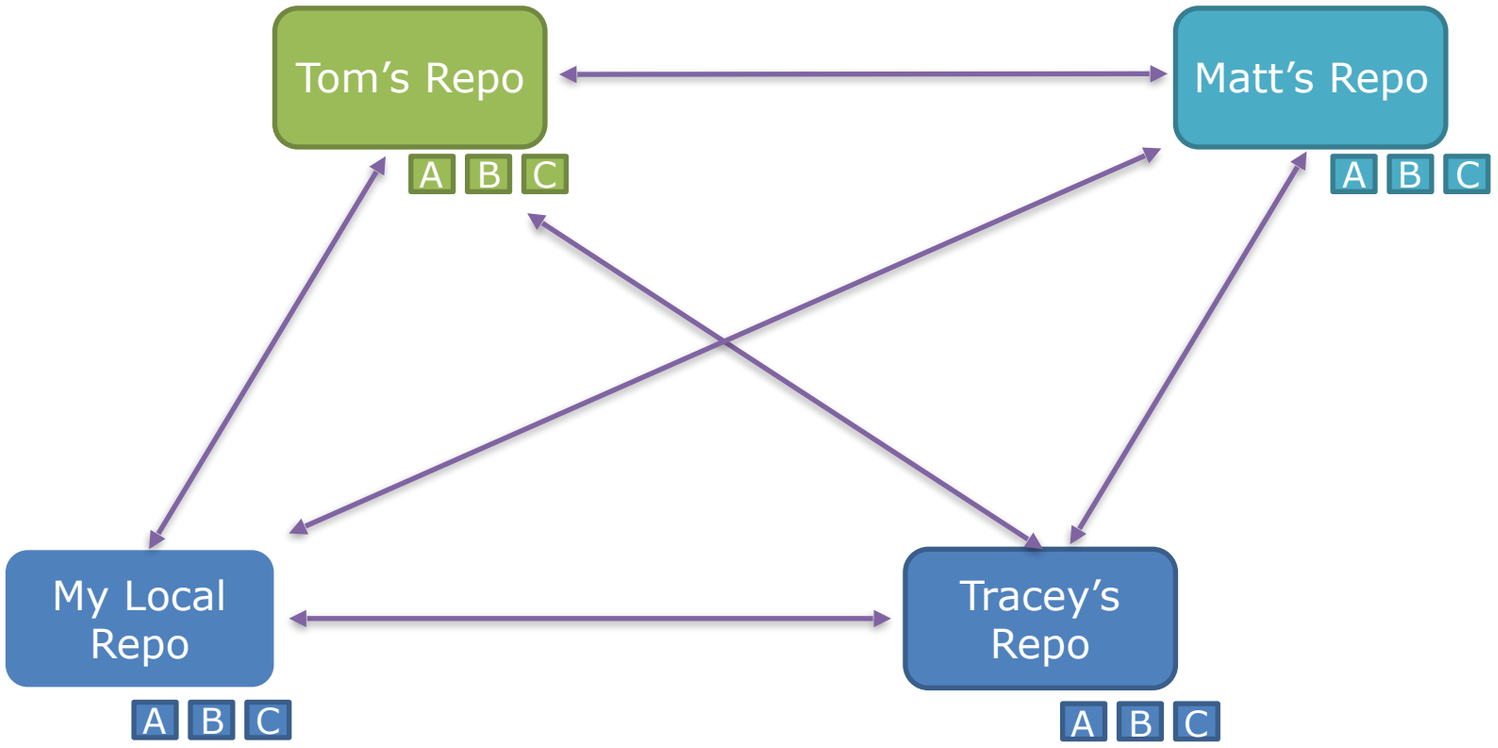
Branches Illustrated




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

- Now we delete the branch pointer
- But notice the structure we have now. This is non-linear. Sometimes this can make it hard to see the changes independently
- “Rebase workflow” is a different way to deal with multiple branches. Beyond the scope of today’s training

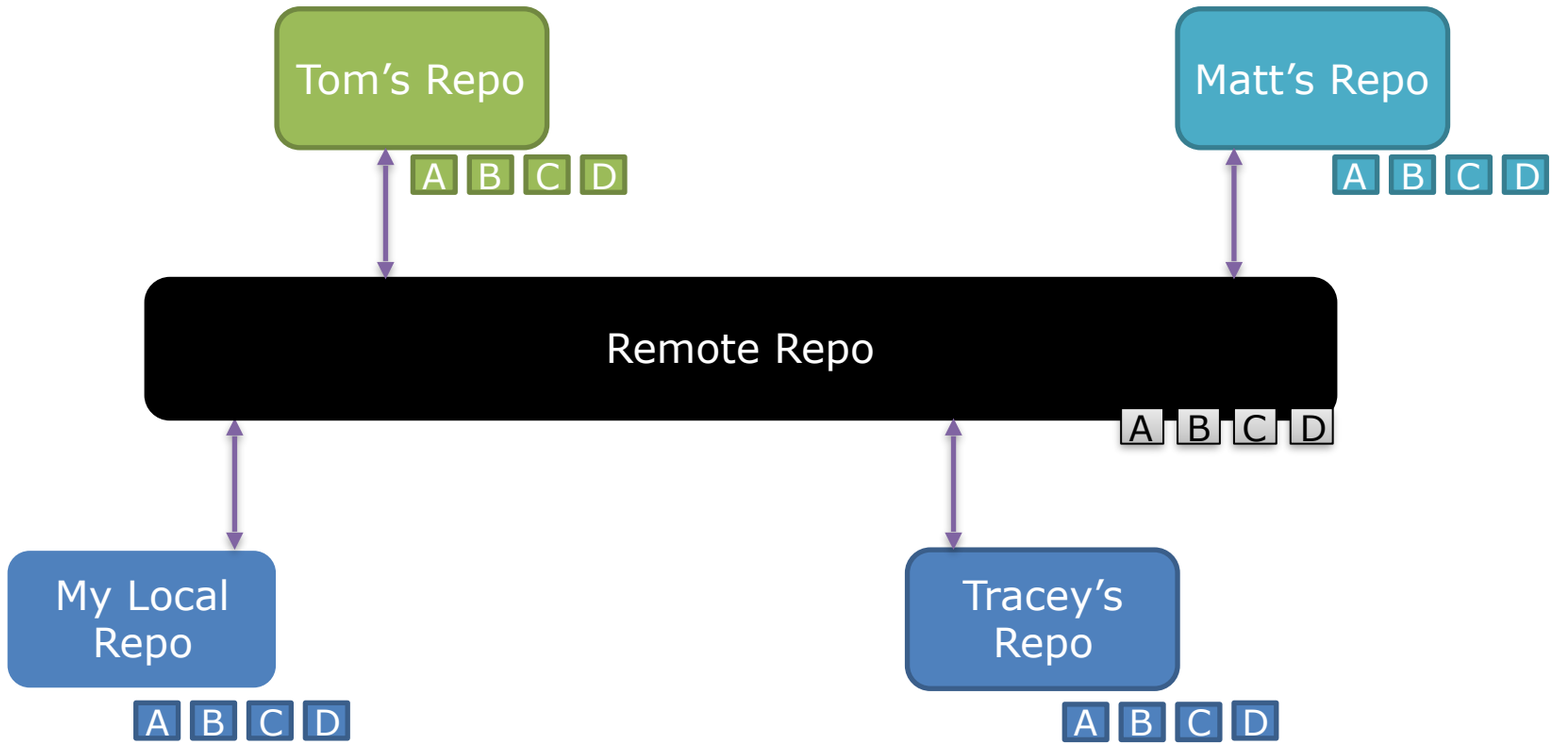
Sharing commits



- Since everyone on the team has a complete copy of the repo, you can do things like this

- But there is another way …

Sharing commits/multiple remotes



- 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

We are going to try sharing commits

- Let's pair up. If there's an odd number of people in the room then one of you can pair with me.

- Give yourselves a label. One of you is "Person 1", and the other is "Person 2". It doesn't matter which way round

- Remember your label

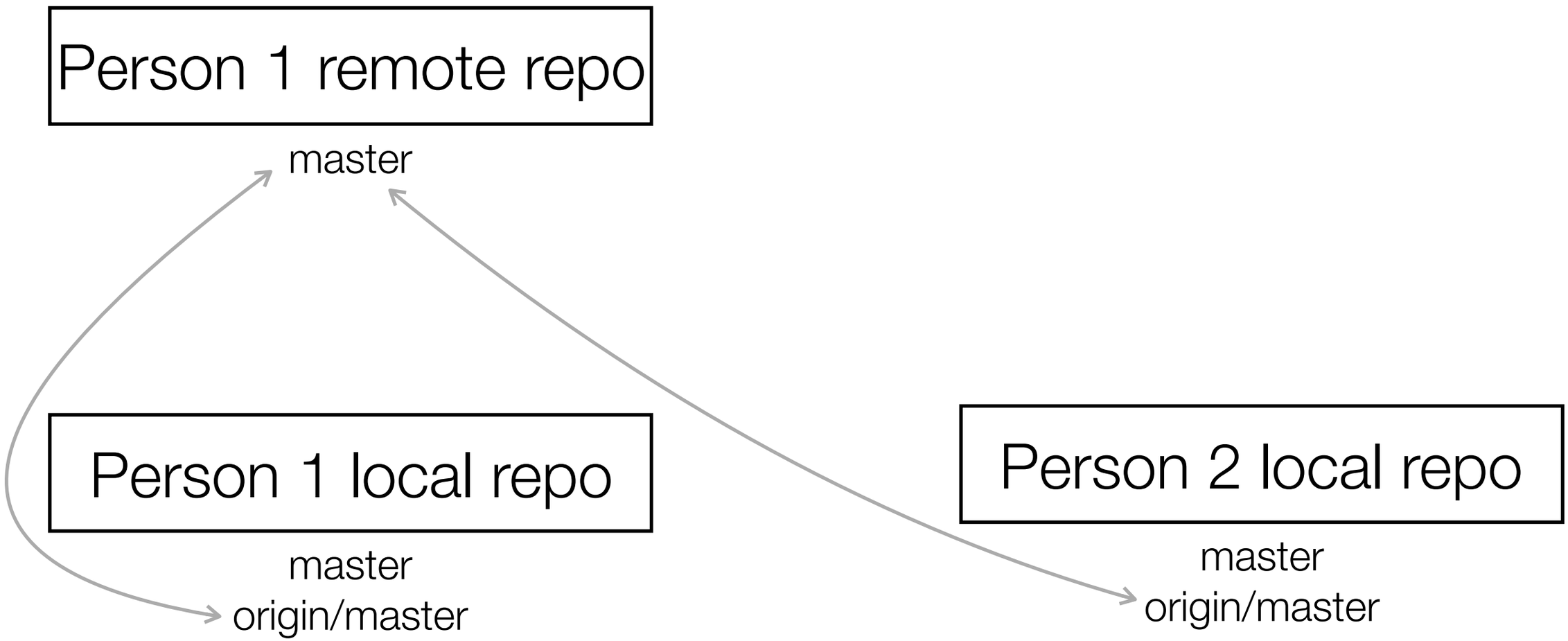
Instructions for “Person 1”

- You have a local `git` repo
- Create a new GitHub repo. This will be a remote (you might have already done this).
- In your local repo, create a remote that points to your GitHub remote (you might already have this set up)
- Make sure you’ve committed everything you need to your local repo
- Push your local repo’s master branch to your GitHub remote

Instructions for “Person 2”

- Ask “Person 1” for their GitHub username: **`x`**
- Ask “Person 1” for the name of their repository: **`y`**
- In your terminal, navigate somewhere outside of a repository
- Type **`git clone https://github.com/x/y.git`**
- Notice that **`git clone`** creates a remote for you called **`origin`**. This should point to Person 1’s GitHub repo. Verify with **`git remote -v`**

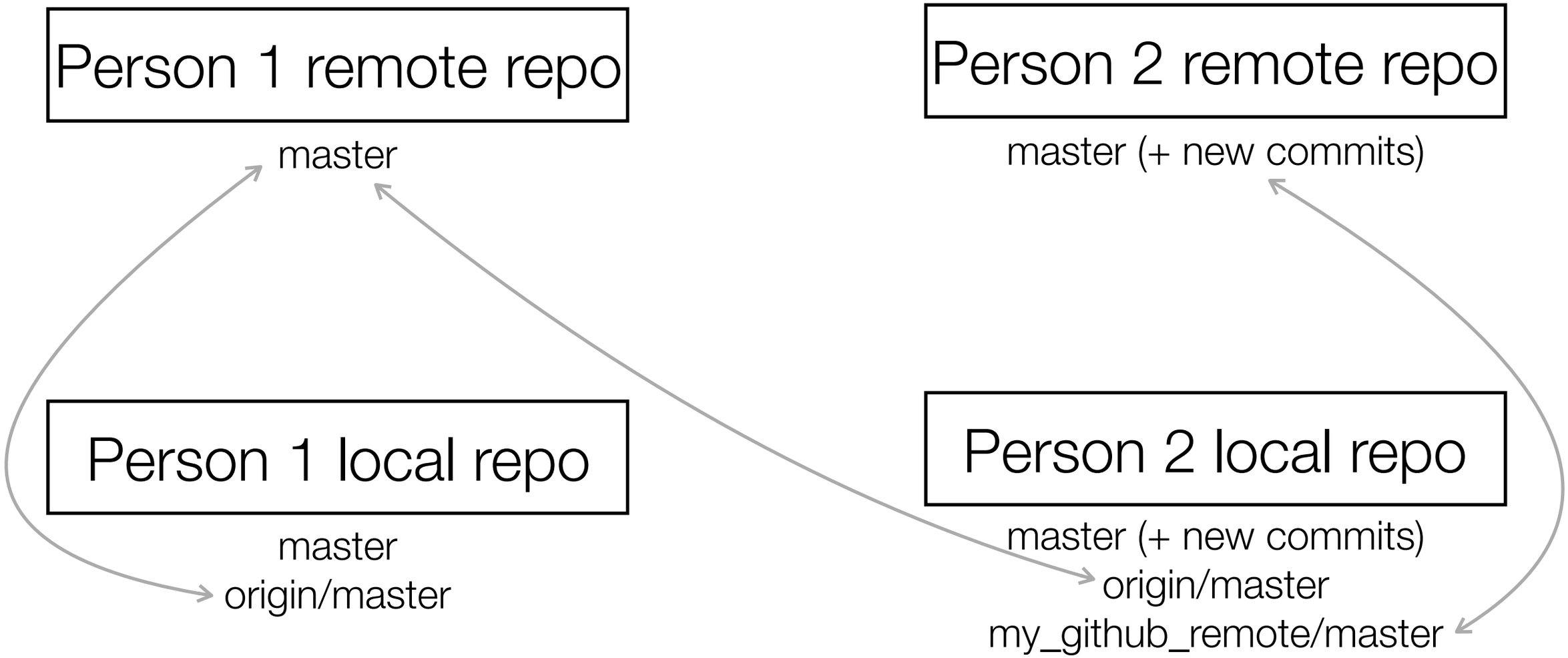
What it all looks like so far…



Instructions for “Person 2”

- Add or edit some files on your local master branch
- Commit your changes, and give it a friendly message. Your changes only exist on your local master branch
- Create a new GitHub repo (this will be a new remote)
- In your local repo, create a remote that points to the GitHub repo you just created. Call it `my_github_remote`
- Make sure you’ve committed everything you need to your local repo
- Push your local repo’s master branch to your GitHub remote:
  - `git push my_github_remote master`

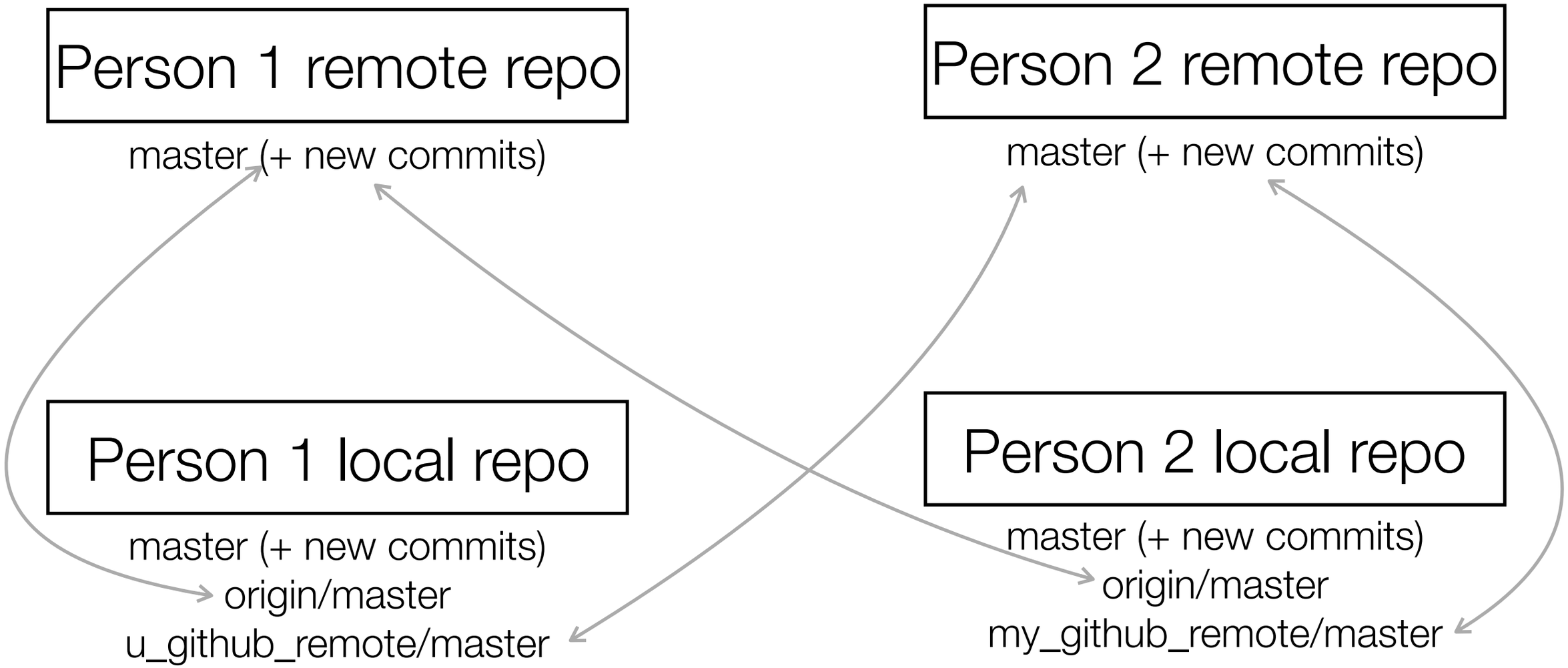
What it all looks like so far…



Instructions for “Person 1”

- Ask “Person 2” for their GitHub username: `u`
- Ask “Person 2” for the name of their remote repo: `v`
- Add a remote to your local repo that points to `https://github.com/u/v.git`
- Call it `u_github_remote`
- Fetch their commits with `git fetch u_github_remote`
- You can inspect their commits with `git log u_github_remote/master`
- You can incorporate their changes into your local repo if you are happy with them: `git merge u_github_remote/master`
- Now you have their commits on your local `master` branch. Push them to your GitHub remote: `git push origin master`

The final picture…



It's a lot to take in

- We've done a lot, and we only dealt with one branch (`master`)
- You can share commits on any branch, not just the `master` branch (remember, `master` is just a default name)
- Multiple remotes + multiple branches => `git` is complicated
- GitHub's web interface makes it a little easier to share commits
  - Github lets people modify their GitHub remotes directly from a web interface (i.e., you don't need to add a `remote` and inspect locally)
- Here's what a multi-branch multi-remote situation might look like…

The final picture…

Person 1 remote repo

master
develop

Person 2 remote repo

master
develop
feature456

Person 1 local repo

master
develop
bug123
origin/master
origin/develop
u_github_remote/master
u_github_remote/develop
u_github_remote/feature456

Person 2 local repo

master
develop
feature456
origin/master
origin/develop
my_github_remote/master
my_github_remote/develop
my_github_remote/feature456

Wrapping up

- You have learned:
  - How to handle merge conflicts
  - How to share commits with a collaborator (multiple remotes)
  - How to handle compartmentalising work (multiple branches)
- We haven’t used any of GitHub’s web features
  - GitHub makes it a little easier to share commits, review a colleague’s commits, and discuss changes in a web interface
- We haven’t talked about best practices for managing branches

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

What to commit?

- Text files only
- Do not commit pdfs, compiled binaries/objects, logs, temporary files, or anything not *needed* by collaborators
- Commit message is just as important as the change you commit. Make your commit messages descriptive and follow this format:
  - First line summarises the change
  - Second line is blank
  - Third line and onwards write as much as you need to describe why you are making the change you are making. And make it convincing.

Some friendly advice

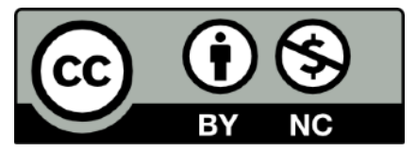
- Always know what branch you are on
    - I have it in my bash prompt
- Minimum Guidelines — Actually using version control is the first step
- Ideal Usage
    - Put EVERYTHING under version control
    - Consider putting parts of your home directory under VC (dot files)
    - Use a consistent project structure and naming convention
    - Commit often and in logical chunks
    - Do all file operations in the VCS
    - Set up change notifications if working with multiple people

Here are some resources

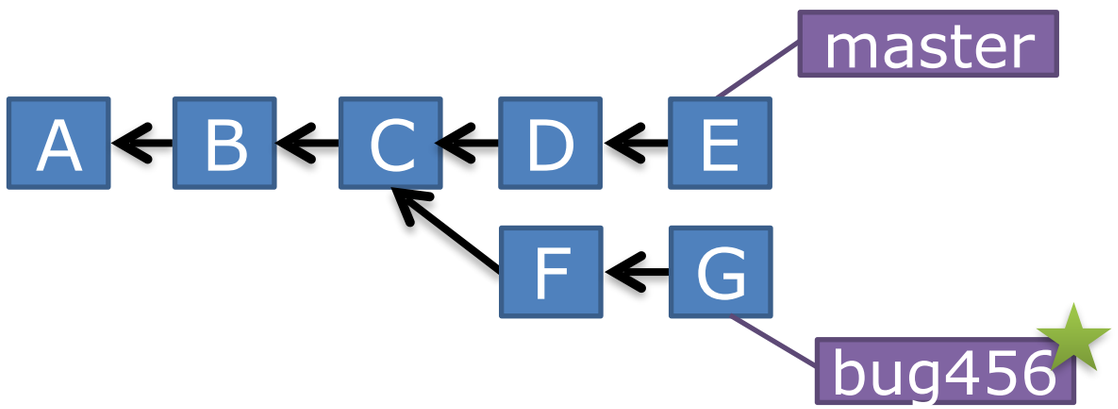
- https://git-scm.com/documentation
- https://git-scm.com/docs/gittutorial
- https://git-scm.com/downloads
- http://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide#323764
- https://github.com
- https://bitbucket.org

Any questions?

License

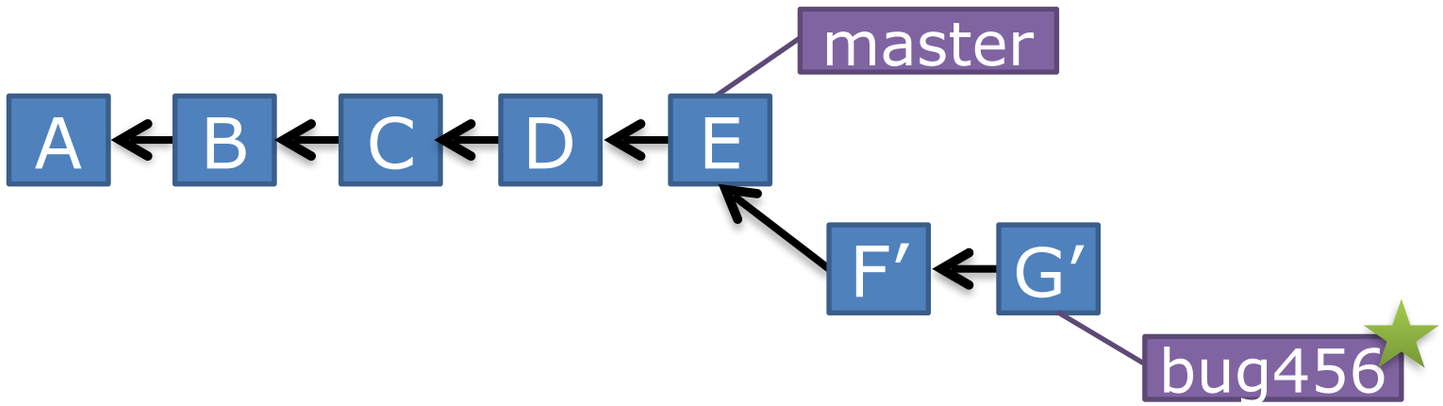


Branches Illustrated



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

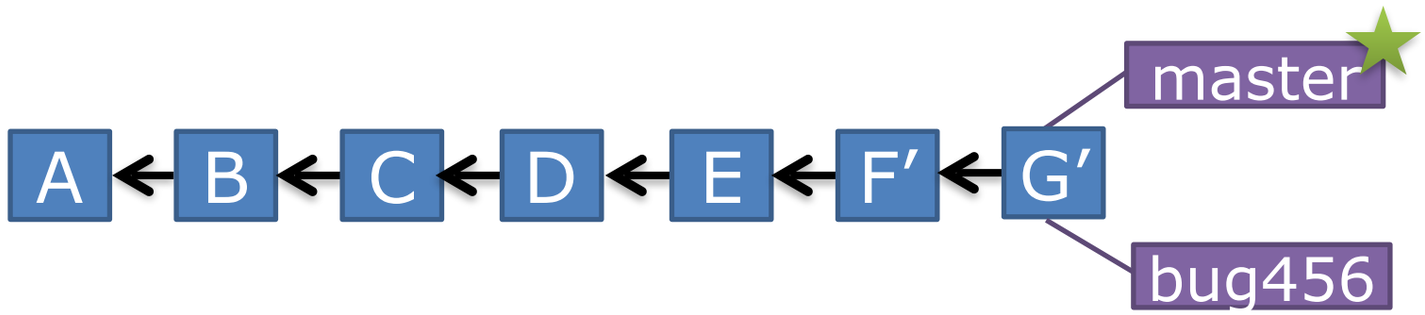
Branches Illustrated




```
> 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

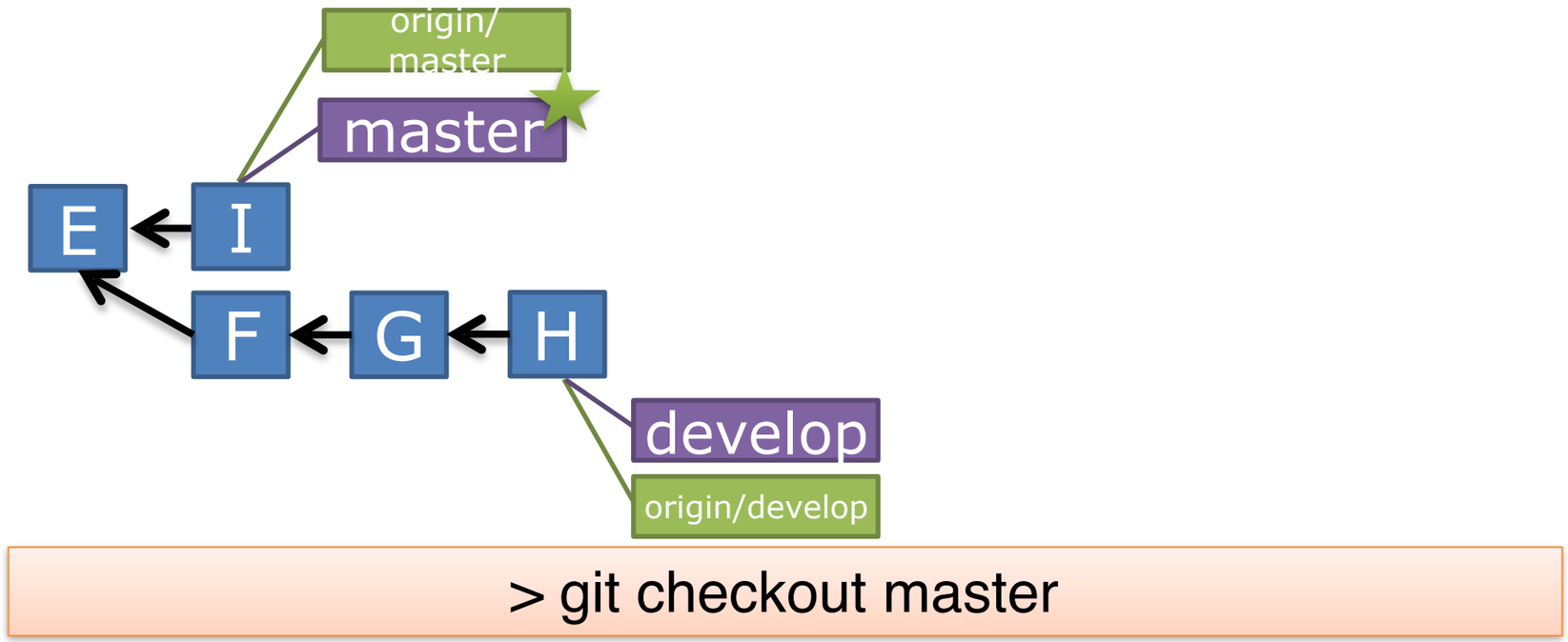
Branches Illustrated




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

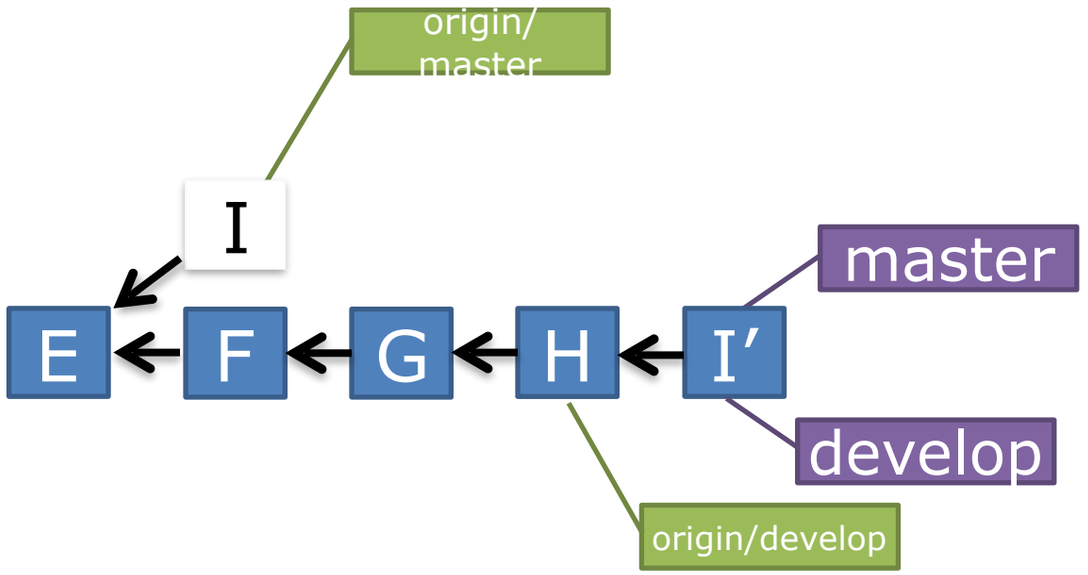
- Now when we merge them, we get a nice linear flow
- Also, the actual change set ordering in the repo mirrors what actually happened. F’ and G’ come after E rather than in parallel to it.

Rebase Flow



- Move onto master

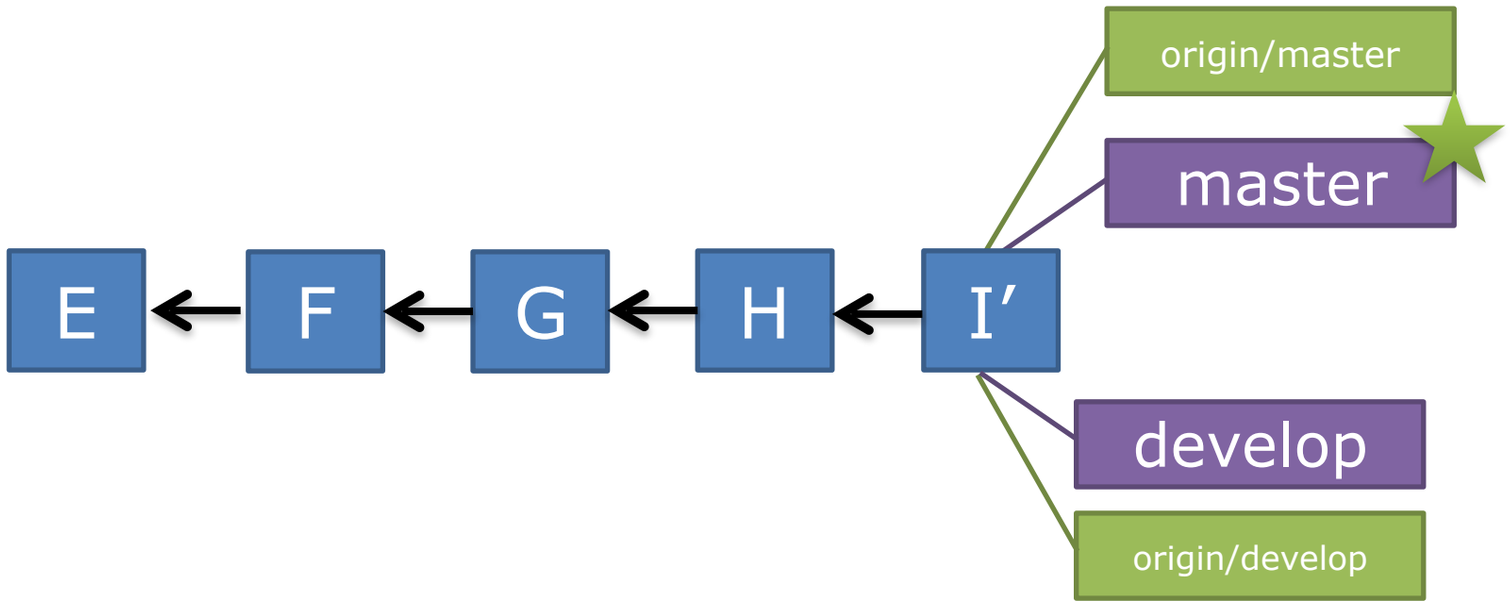
Rebase Flow



> git rebase develop

- Rebase develop onto master

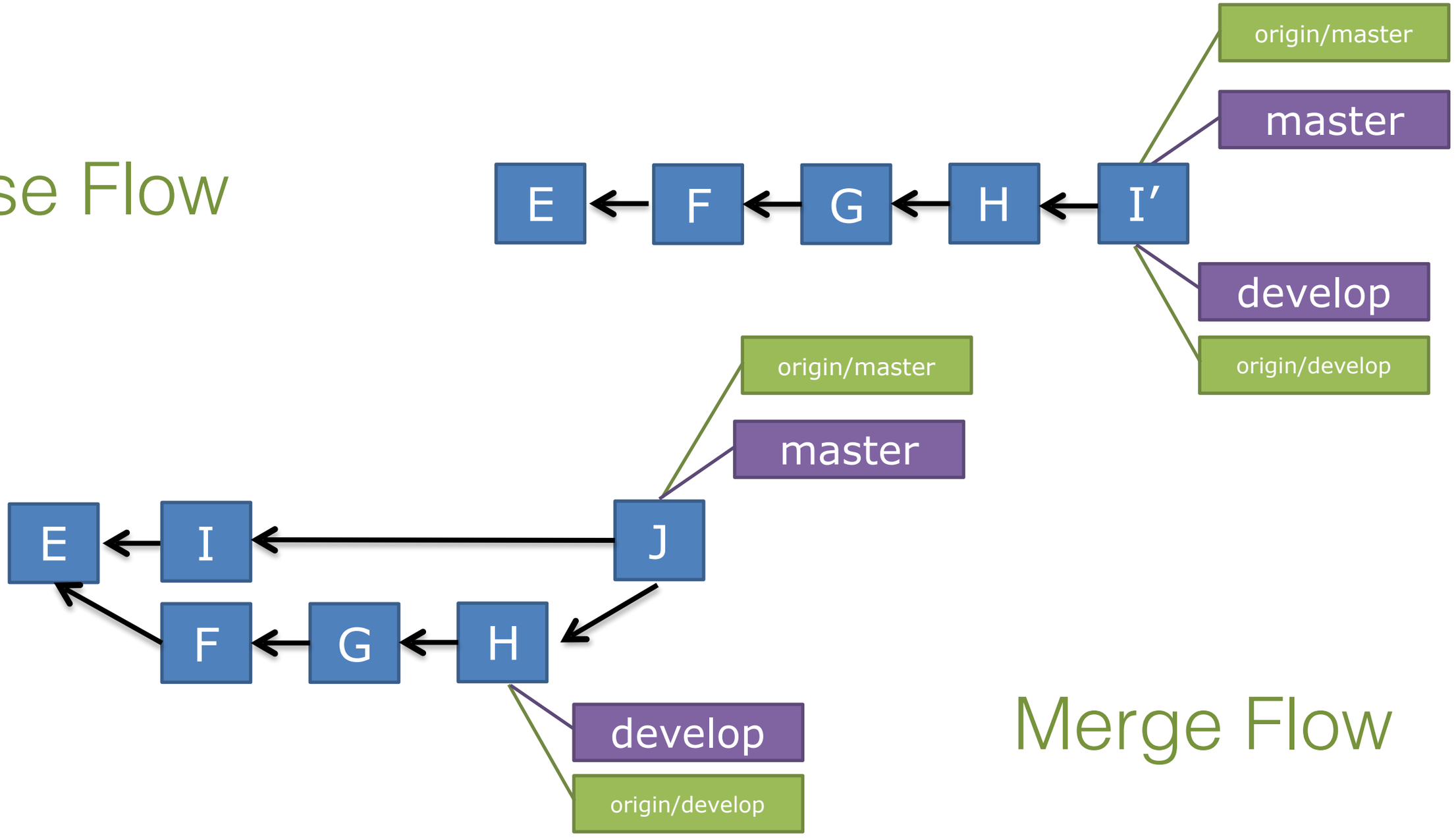
Rebase Flow




```
> git push -f origin
```

- Get origin up to date

Rebase Flow

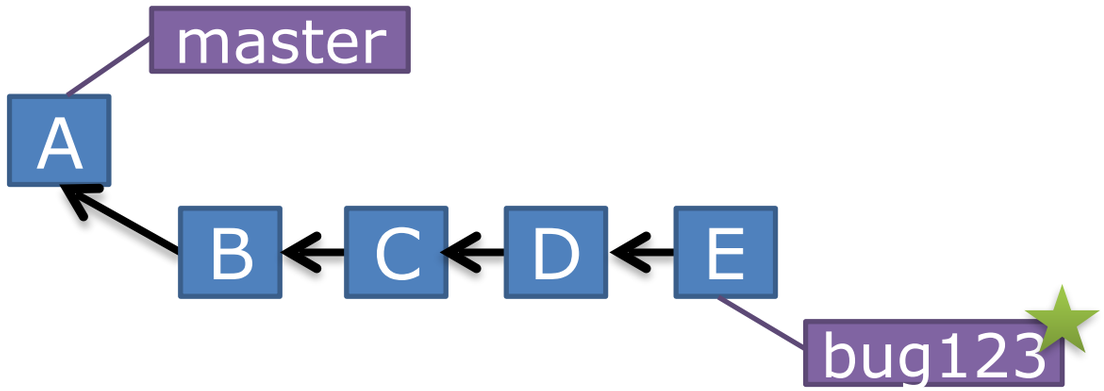


Merge Flow

- 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

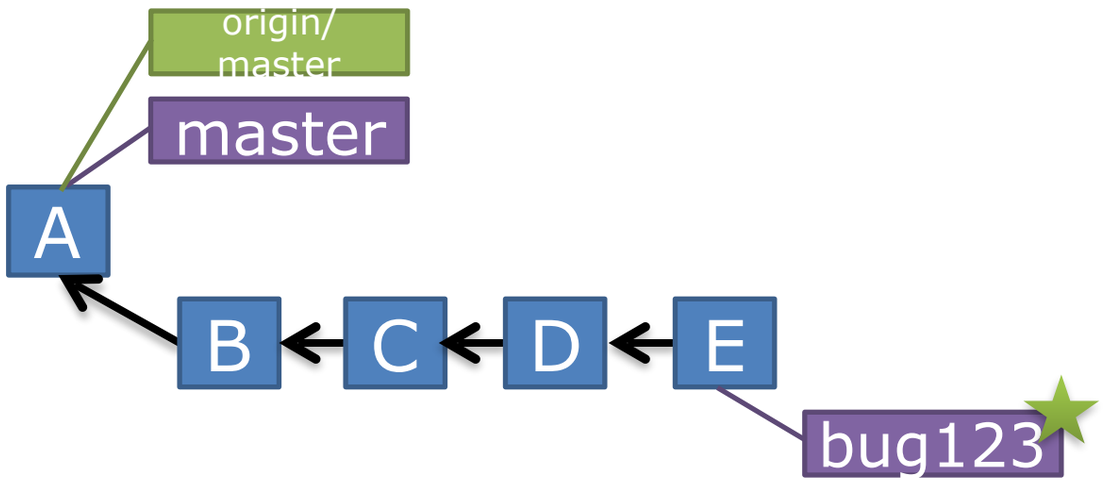
Branches Illustrated

- Suppose we are here:



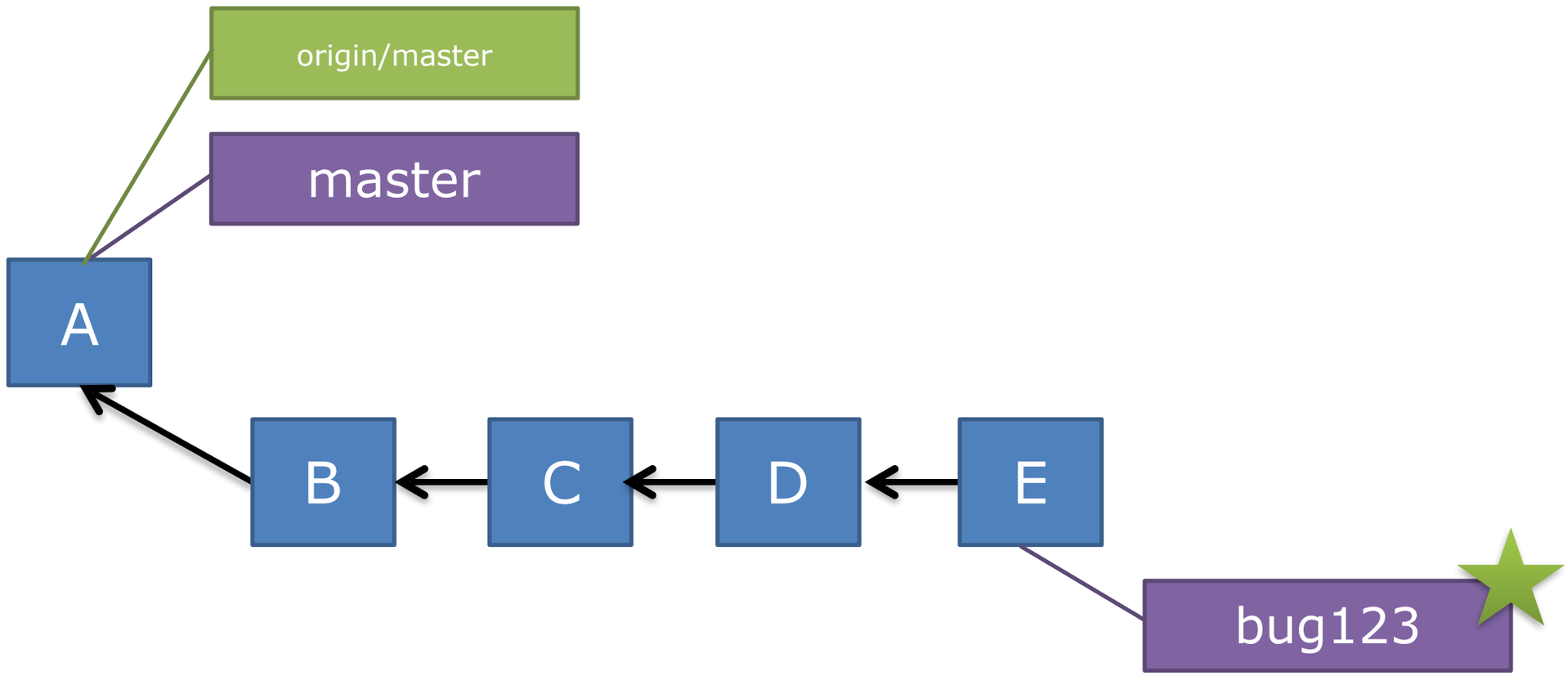
- We cloned master on (A) and have been fixing bug123 in our story branch

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

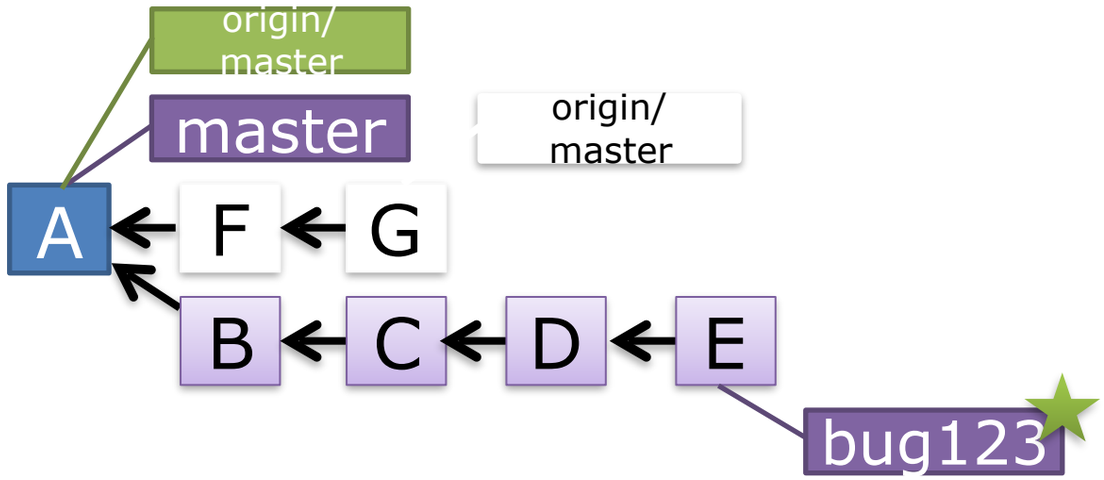
Branches Illustrated



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

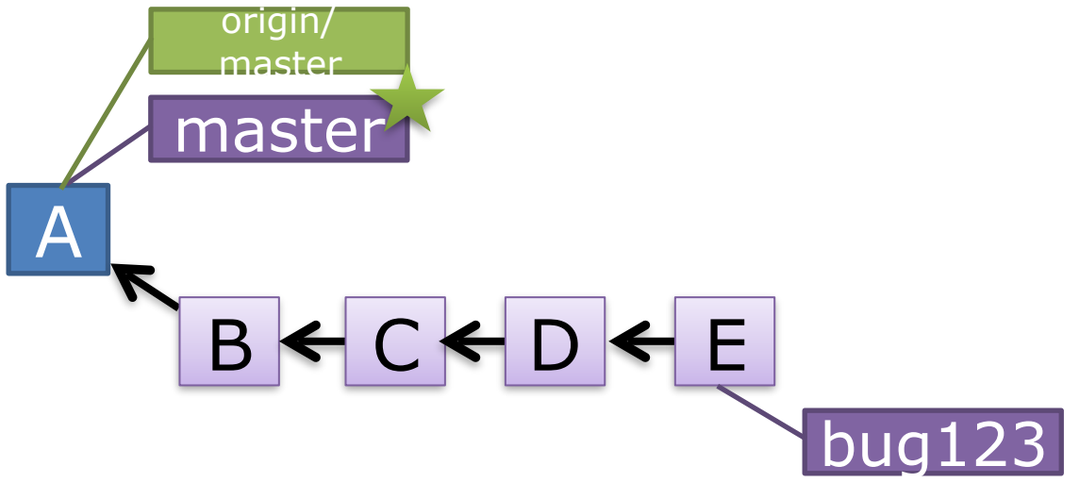
- The changes on the bug123 branch are only known to the local repo. The remote server does not have these changes (or the bug123 branch for that matter)

Branches Illustrated



- In reality, there are two versions 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

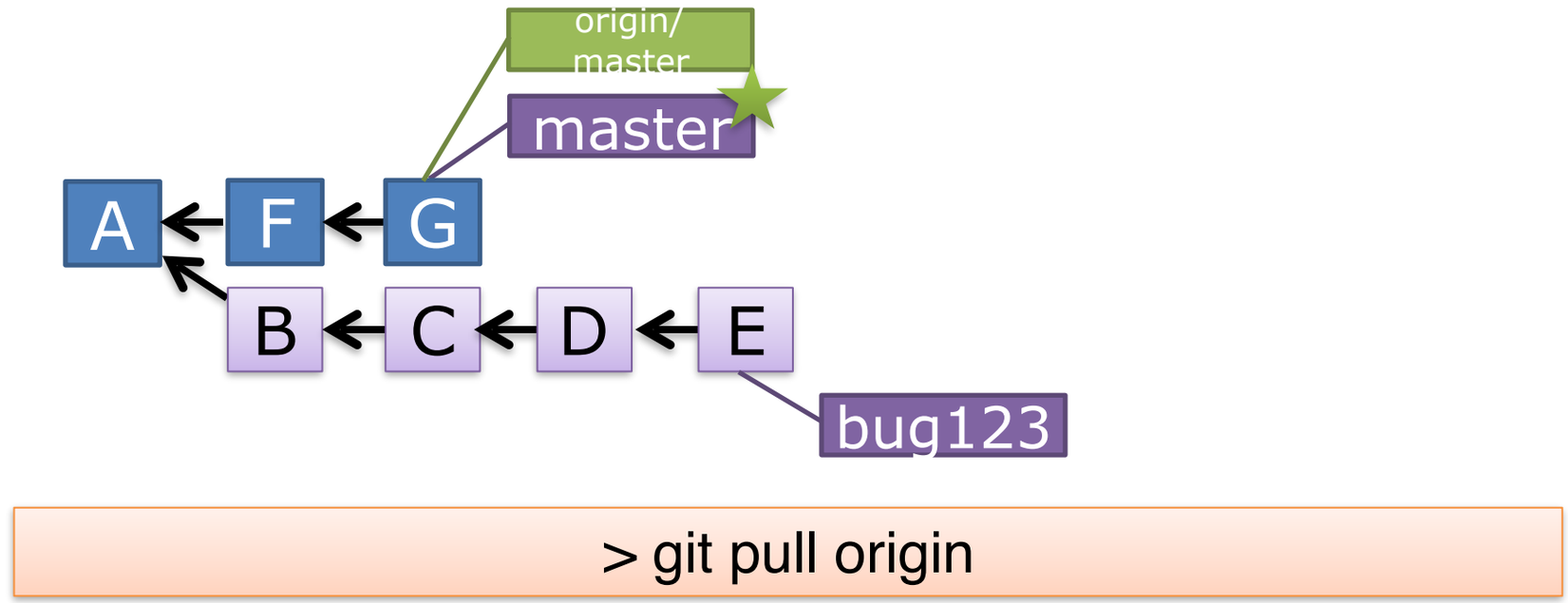



```
> git checkout master
```

- So if this is what we know locally, we can update our master to catch up

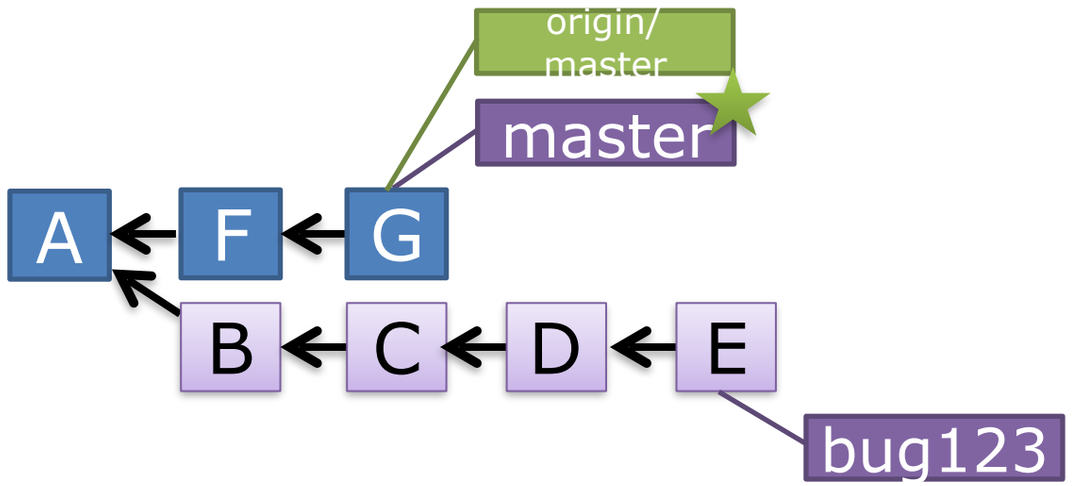
- First we checkout master which move our current (*) to there. Note that we are actually on OUR master not the upstream one. This is always true. But the tracking branch is also pointing to (A) at this point.

Branches Illustrated



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

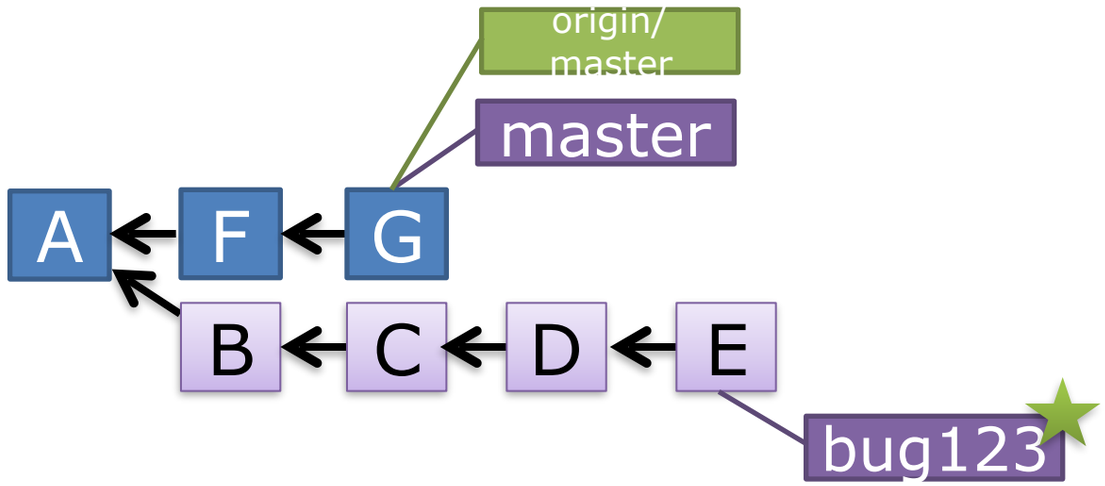
Pull = Fetch + Merge



> git pull origin

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

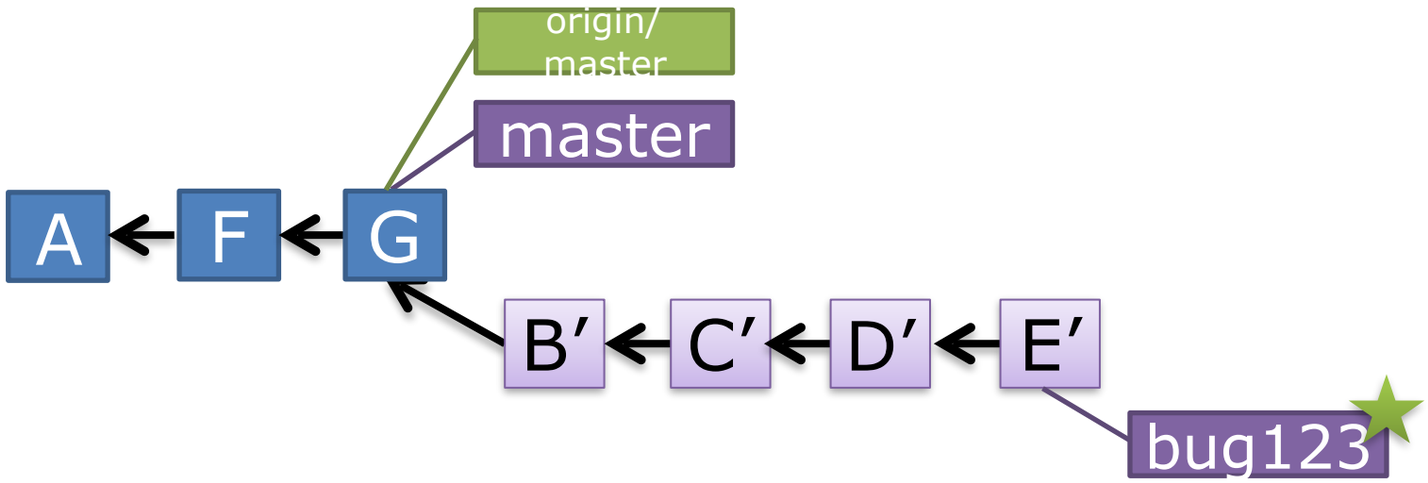
Branches Illustrated




```
> 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

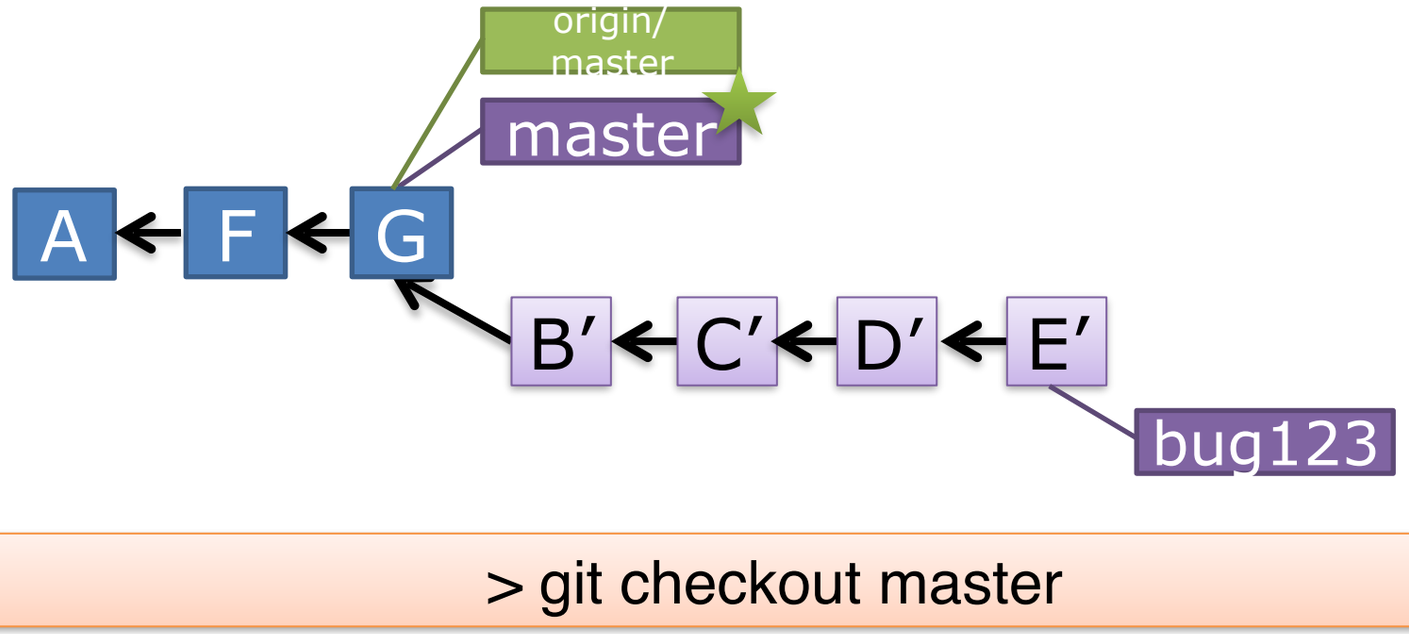
Branches Illustrated




```
> git rebase master
```

- So, we rebase our bug123 branch onto master

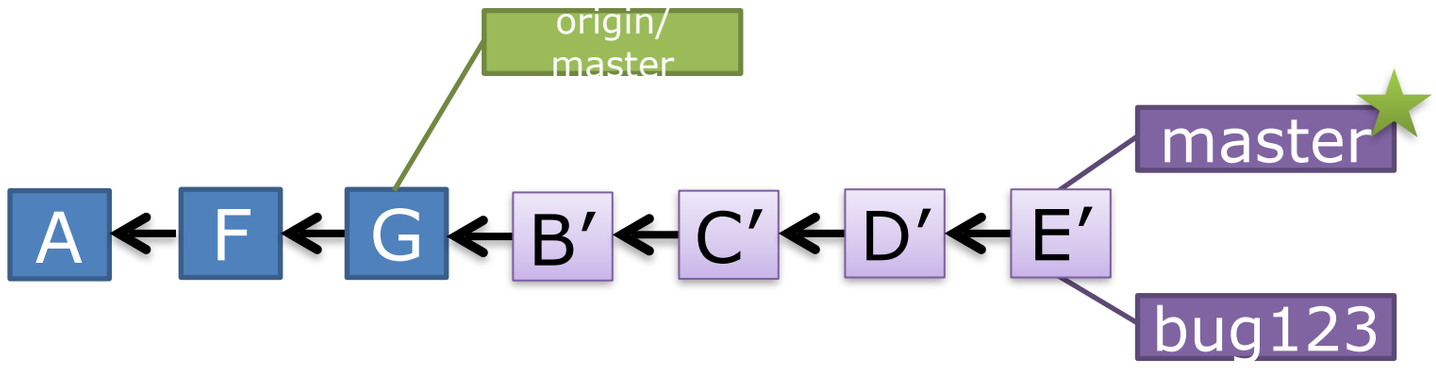
Branches Illustrated



> git checkout master

- We checkout master, once again

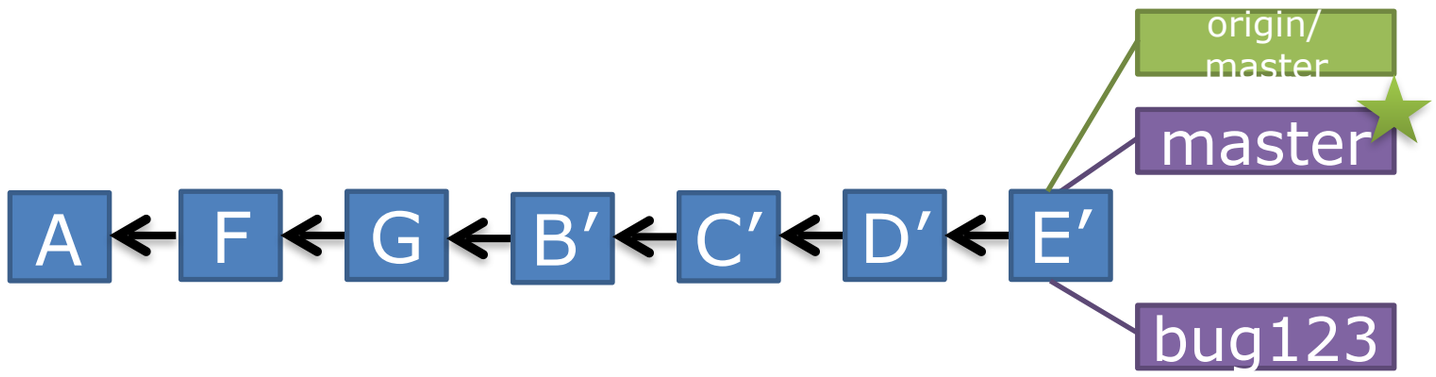
Branches Illustrated




```
> git merge bug123
```

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

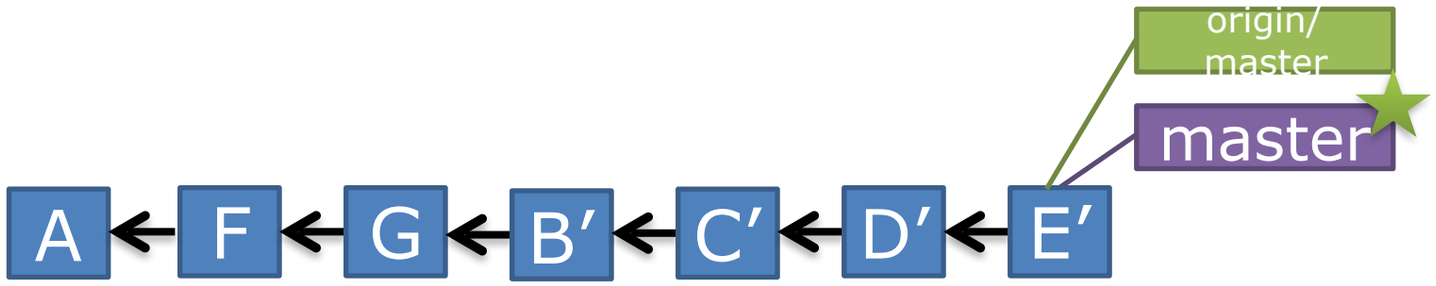
Branches Illustrated




```
> 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

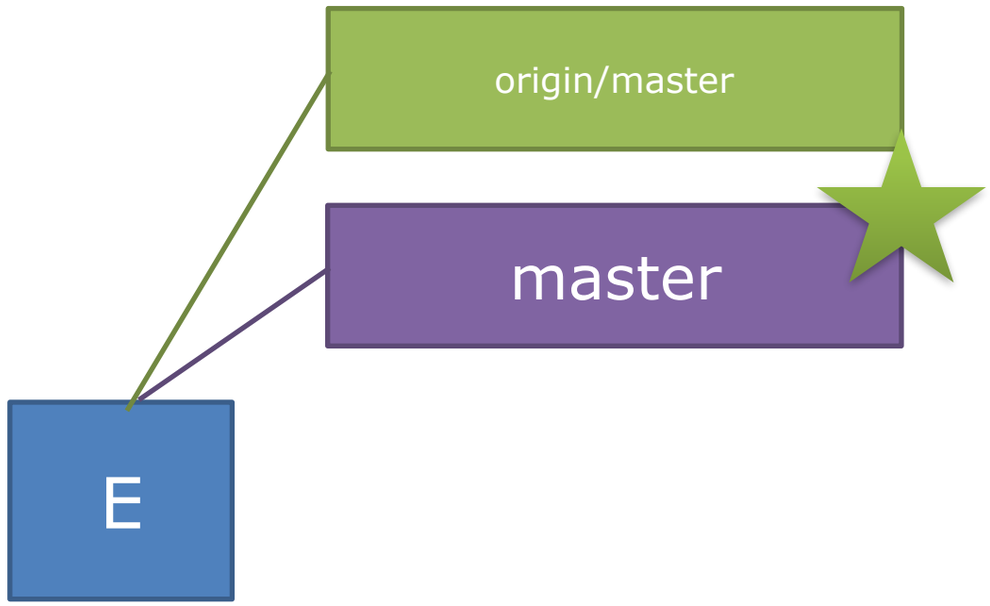
- Local branches are short lived
- Staying off master keeps merges simple
- Enables working on several changes at once

- Create
- Commit
- Merge
- Delete

Short vs. Long-Lived Branches

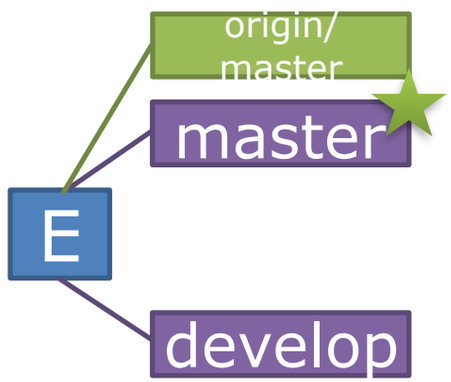
- Great for multi-version work
- Follow same rules as Master / Story branches
- Integrate frequently
- Pushed to Remotes

Branches Illustrated



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

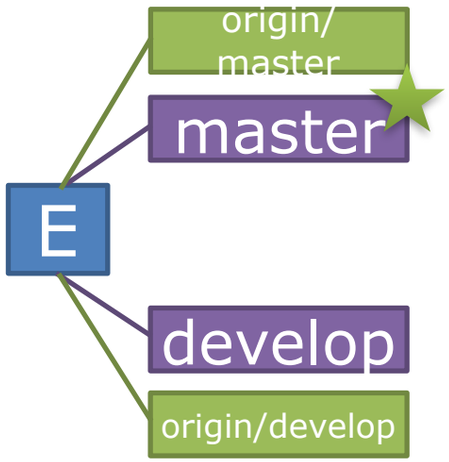
Branches Illustrated




```
> git branch develop
```

- We, will want to create a develop branch

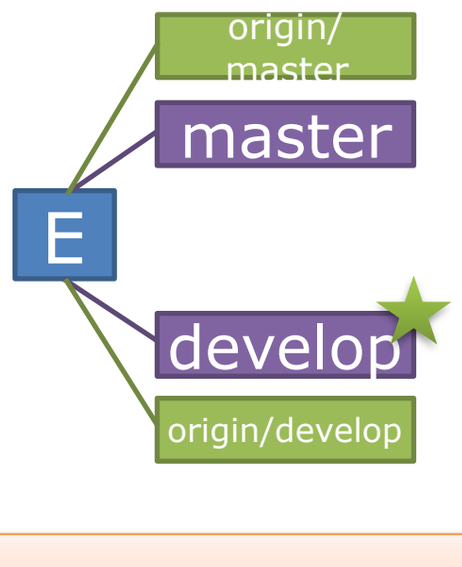
Branches Illustrated




```
> git push origin develop
```

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

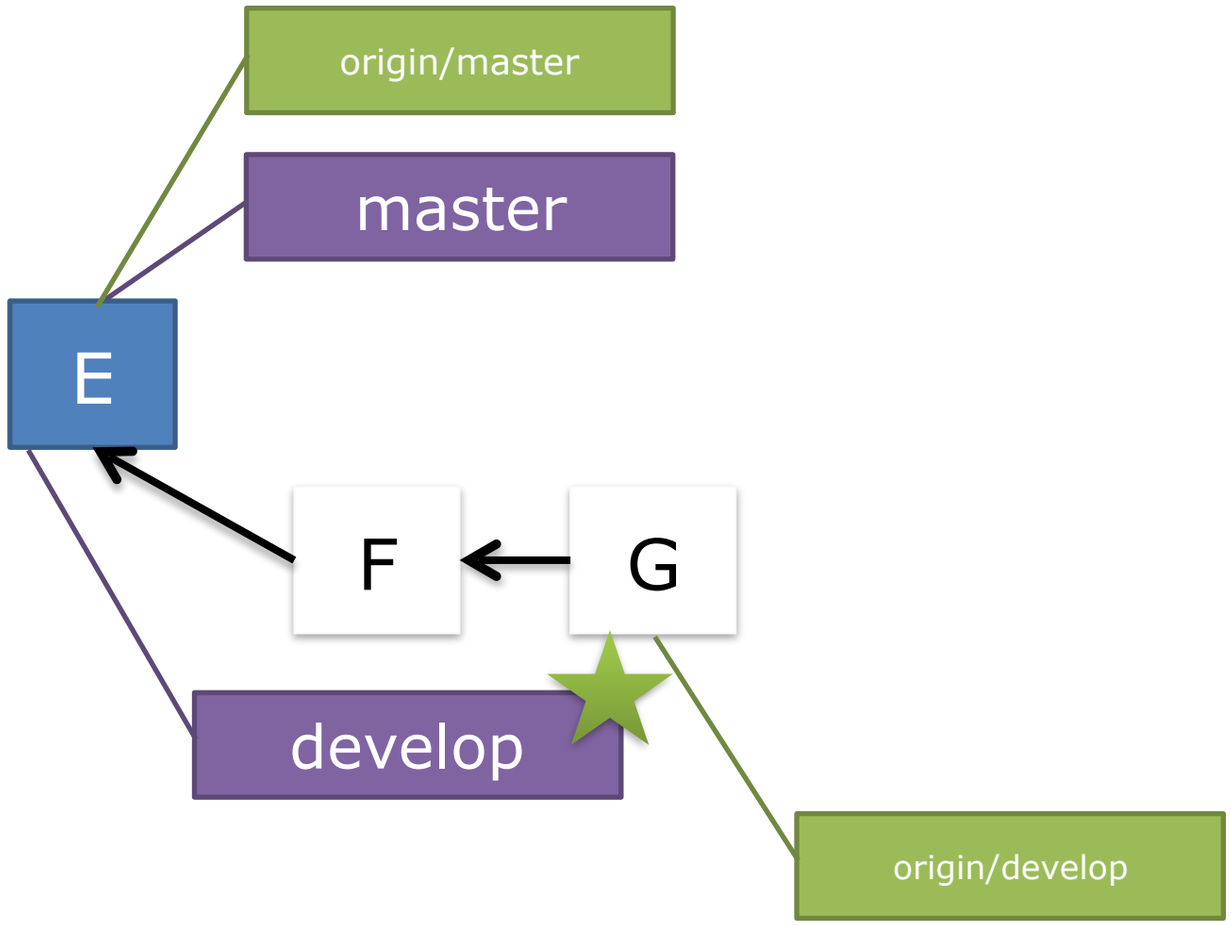
Branches Illustrated




```
> git checkout develop
```

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

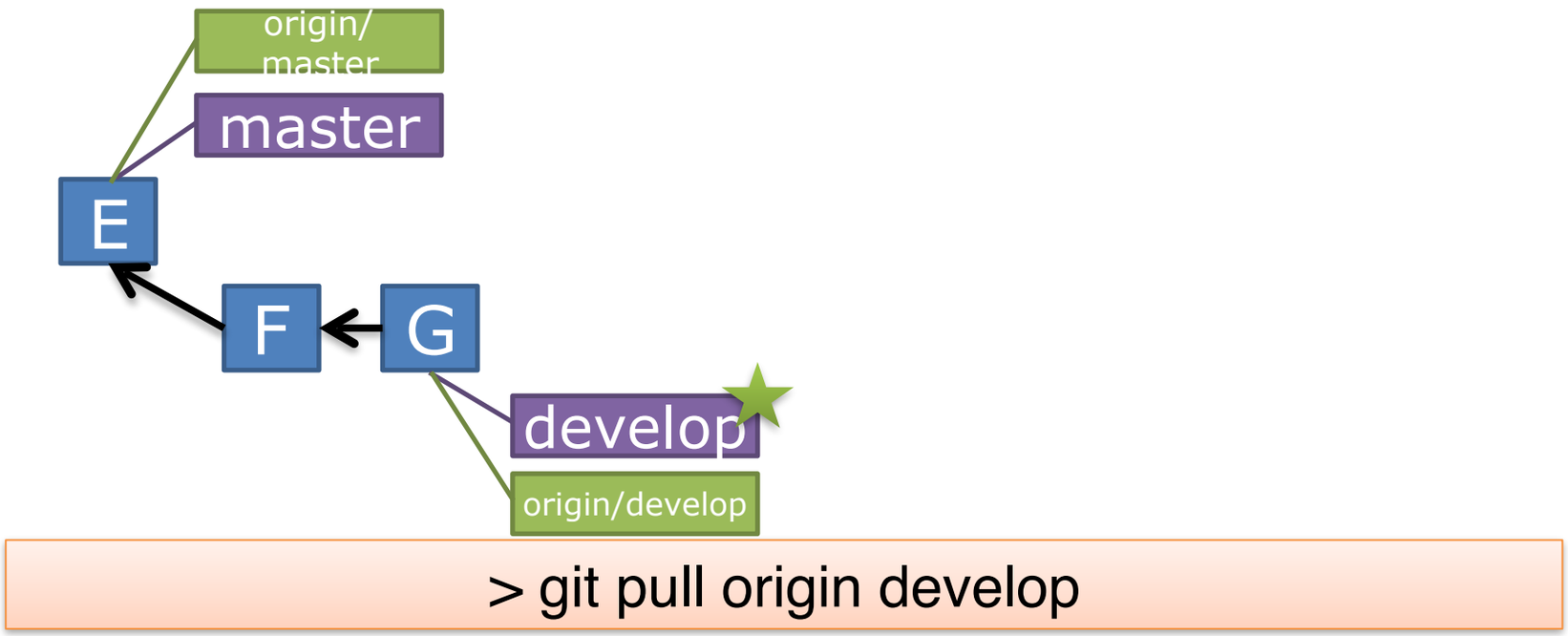
Branches Illustrated



- 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

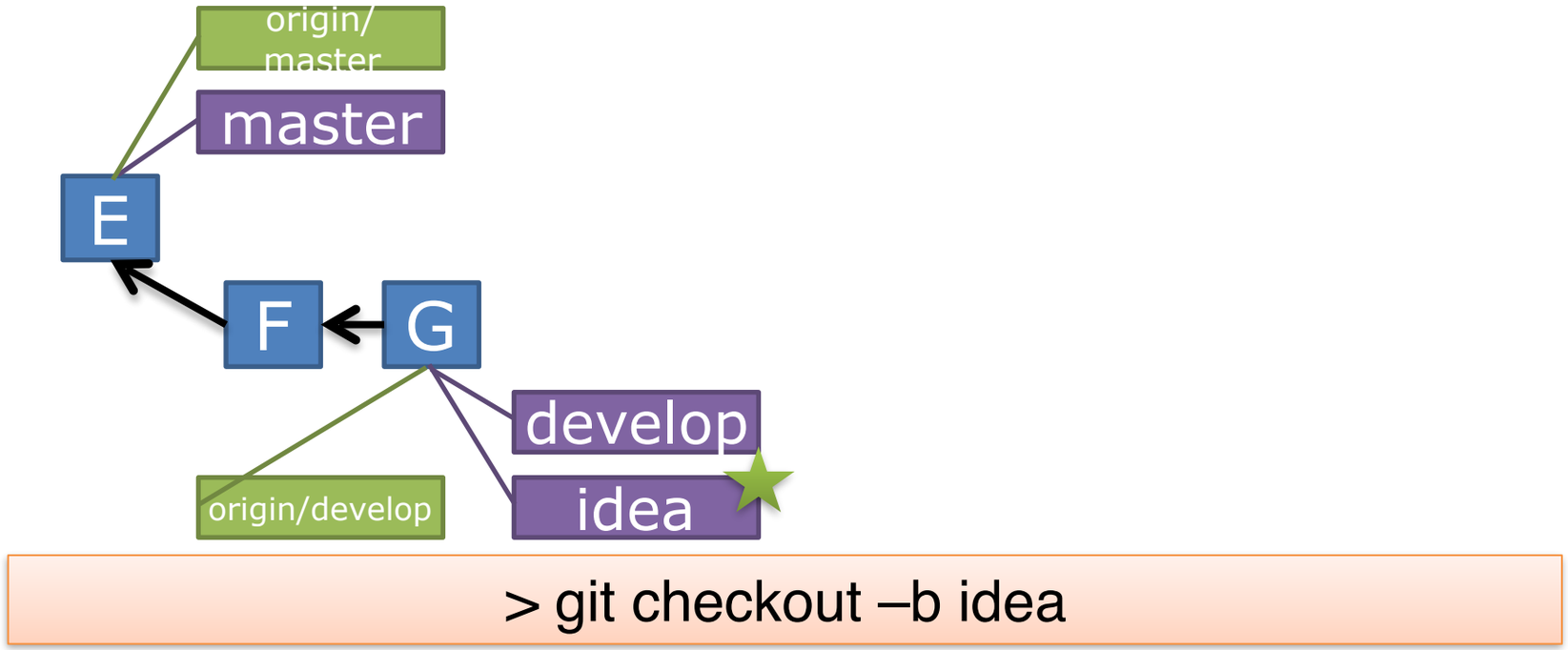
Branches Illustrated



- So, before we start our work for the day, we pull down the latest changes to origin/develop

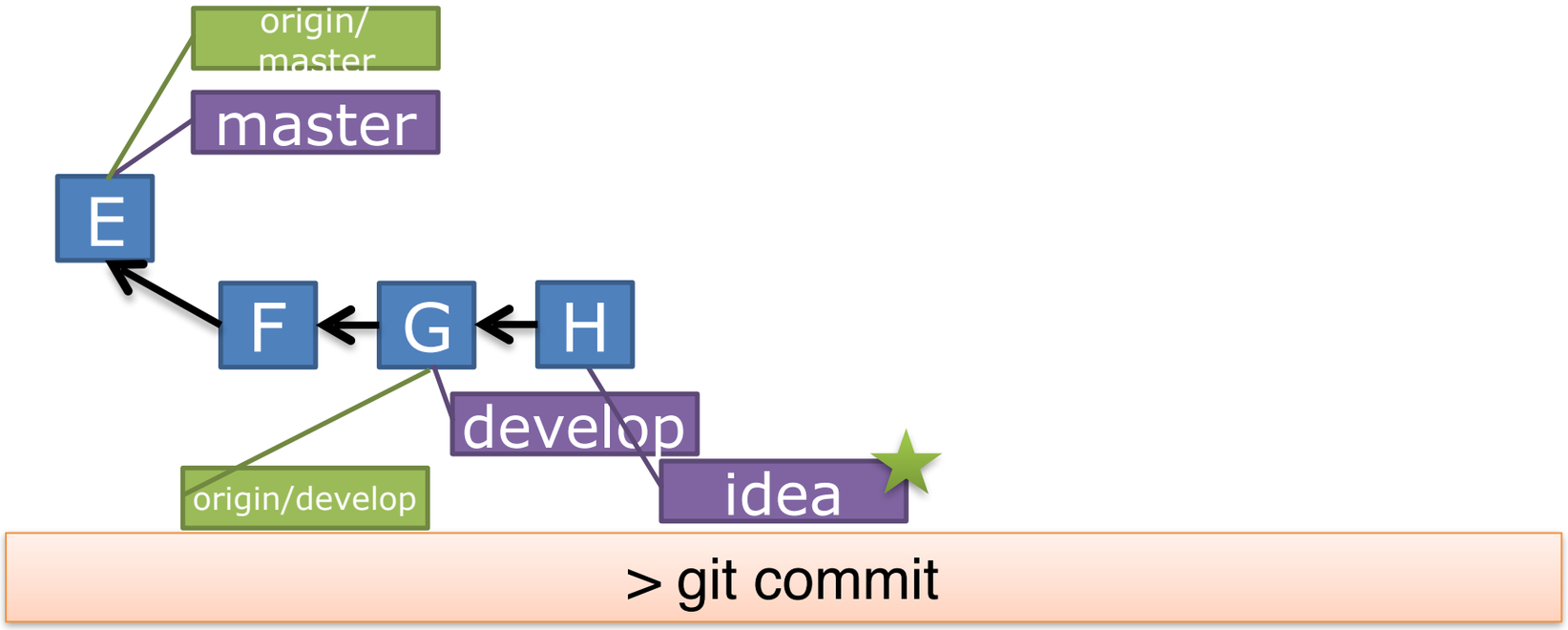
- Remember, pull = fetch + merge

Branches Illustrated



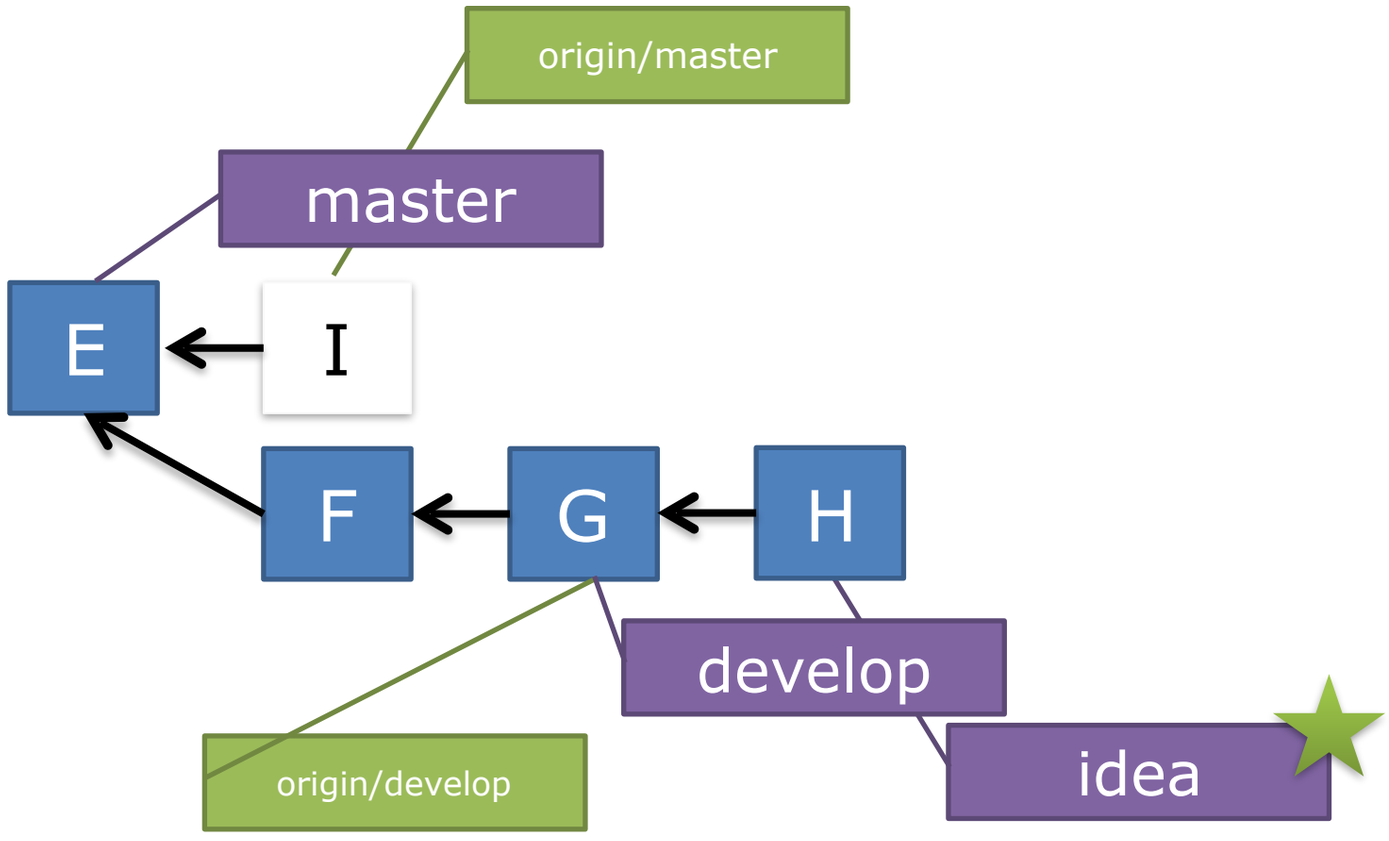
- I have a great idea. So, I create a working branch “idea” off of develop

Branches Illustrated



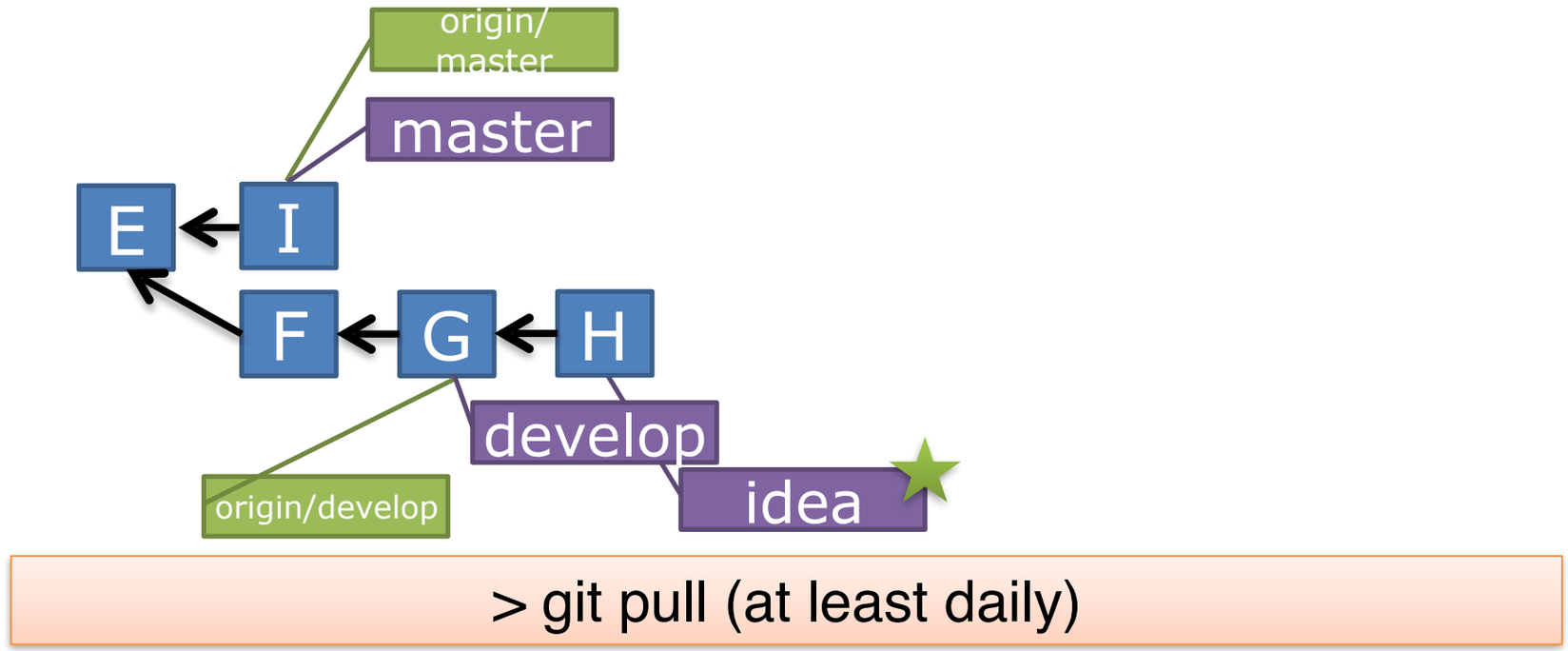
- I did some work in “idea” and moved my local copy along a commit

Branches Illustrated



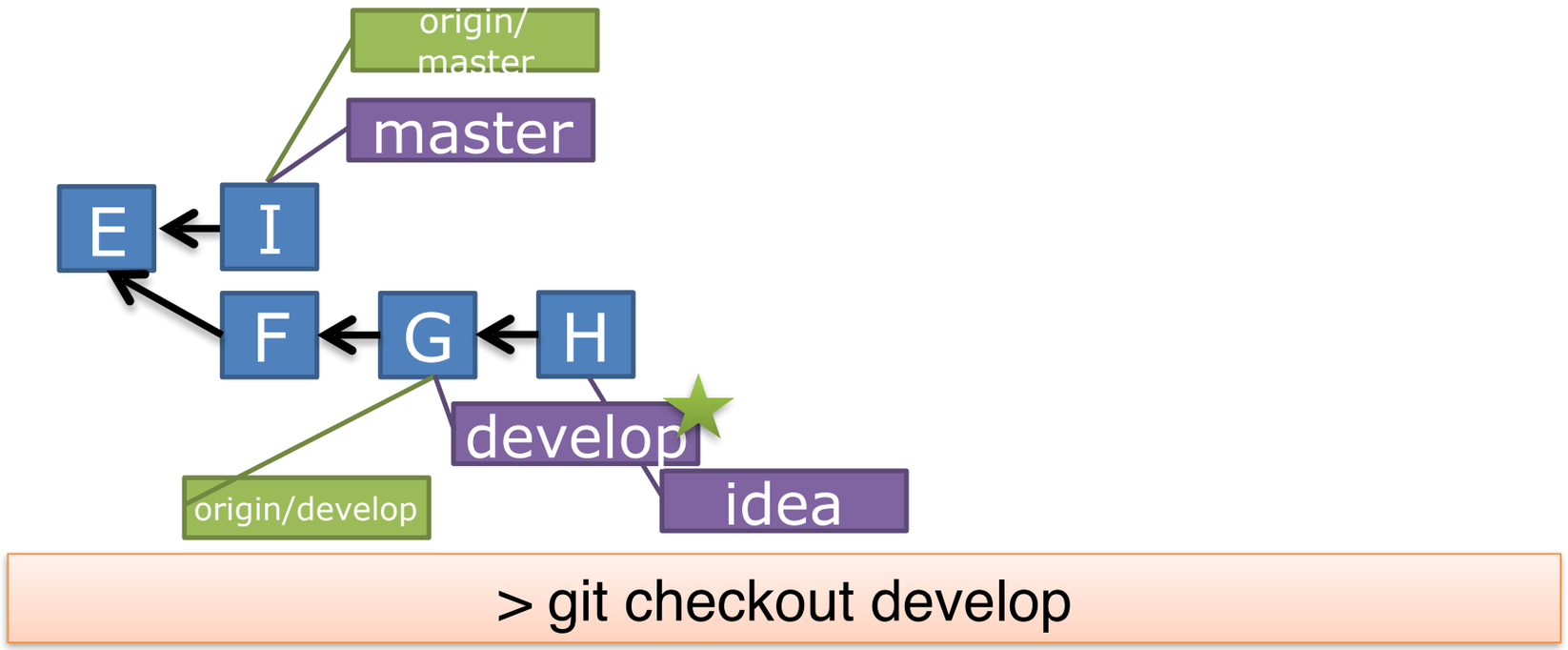
- In the meantime, one of my group mates did a hot fix to the production “master” branch

Branches Illustrated



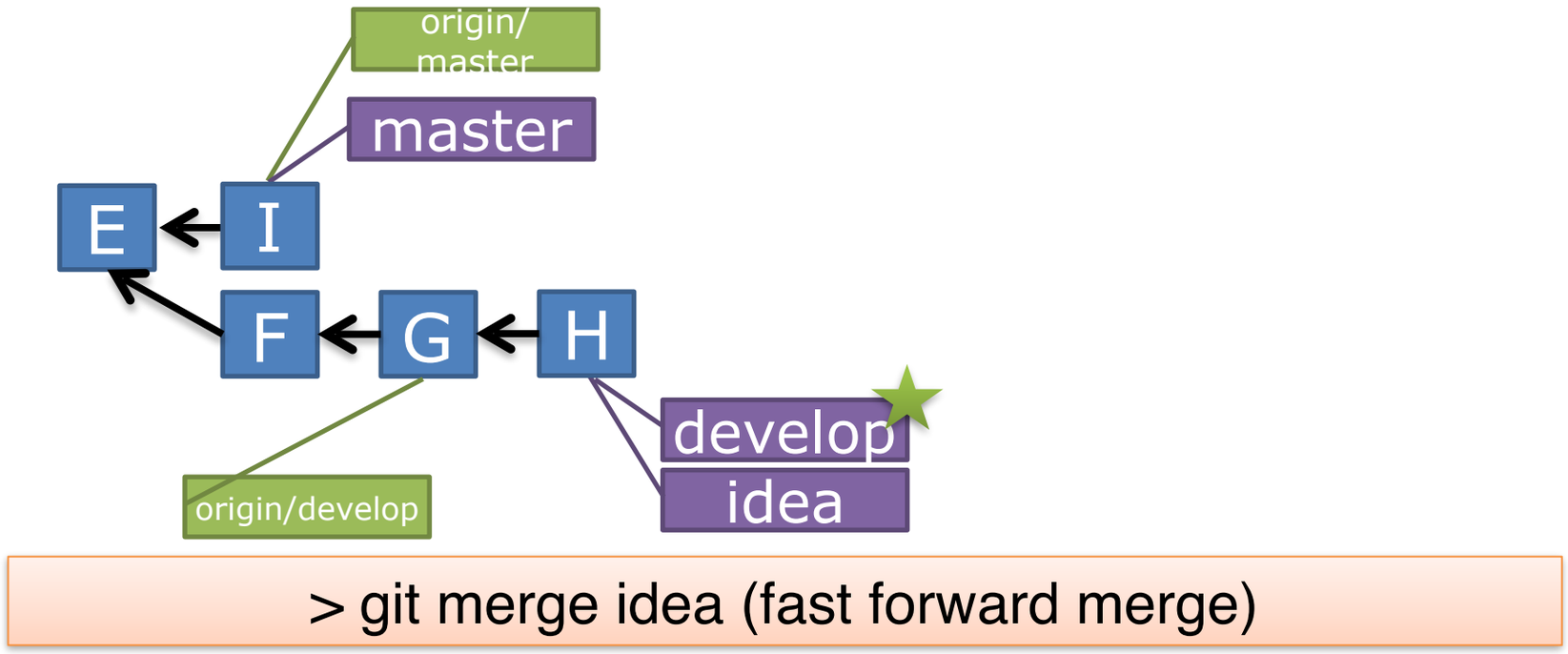
- Since I like to keep up to date, I pull down the changes while I'm working (perhaps I got a notification via email that a hot fix went in)

Branches Illustrated



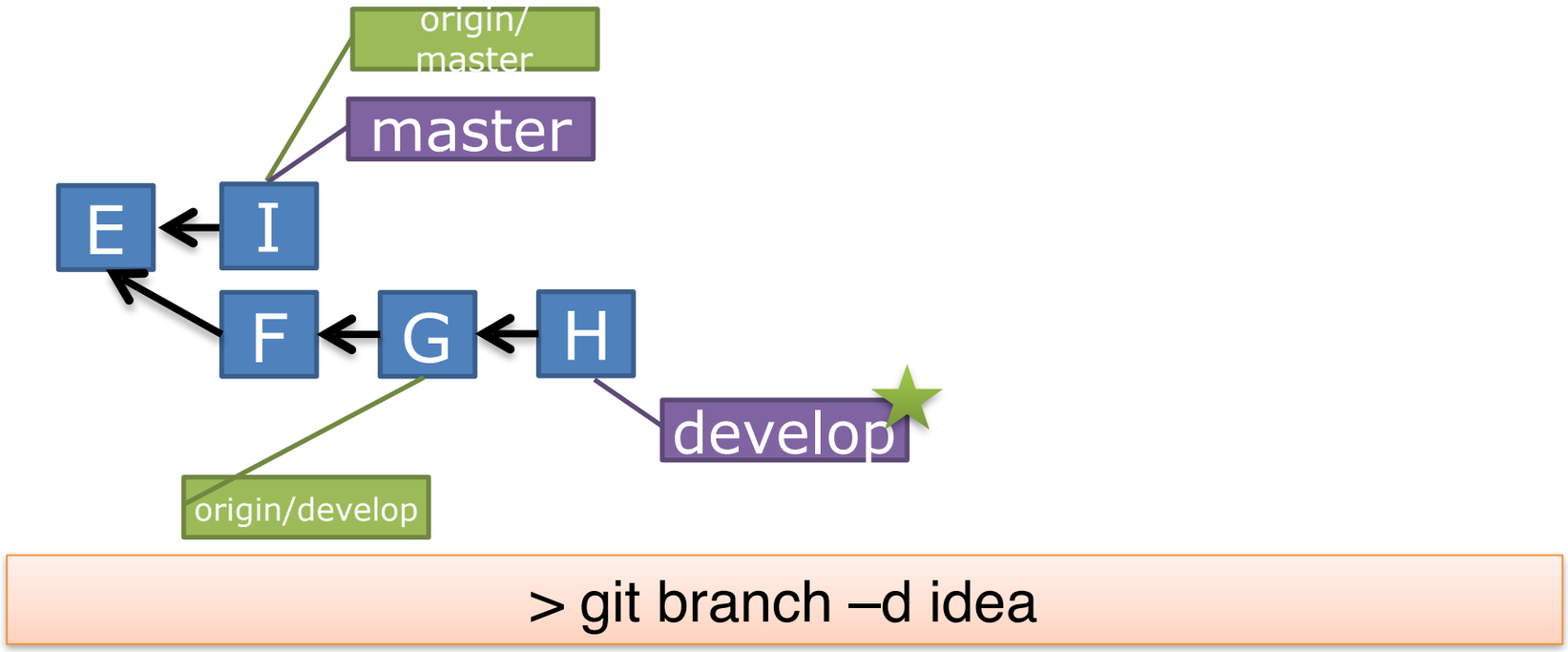
- Now, I'm going to merge "idea" back into develop, so I switch back my local working copy to the develop branch

Branches Illustrated



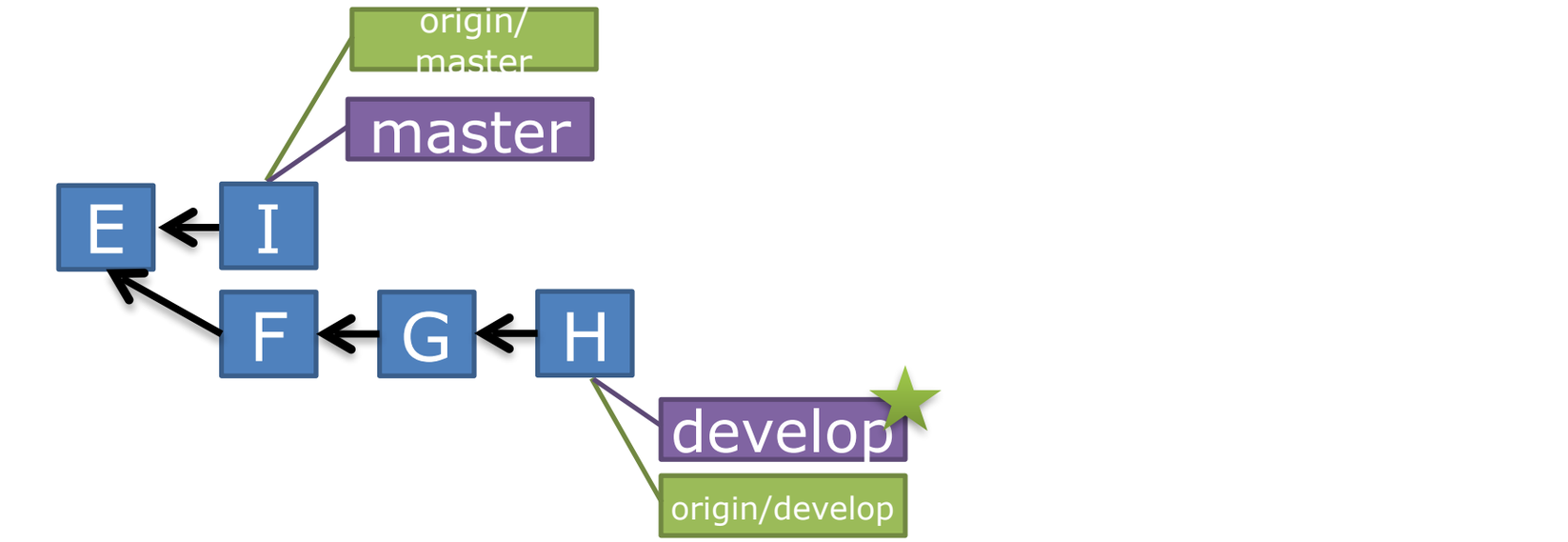
- Now, I merged my idea changes into the local develop branch. Since there are no other changes to develop, this is called a “fast forward” merge

Branches Illustrated



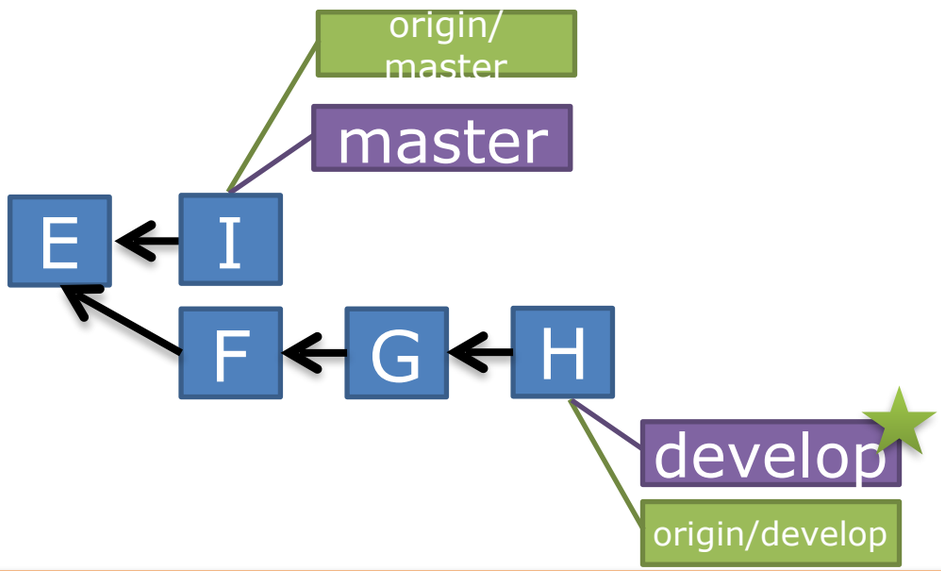
- I clean up my idea branch (delete it)

Branches Illustrated



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

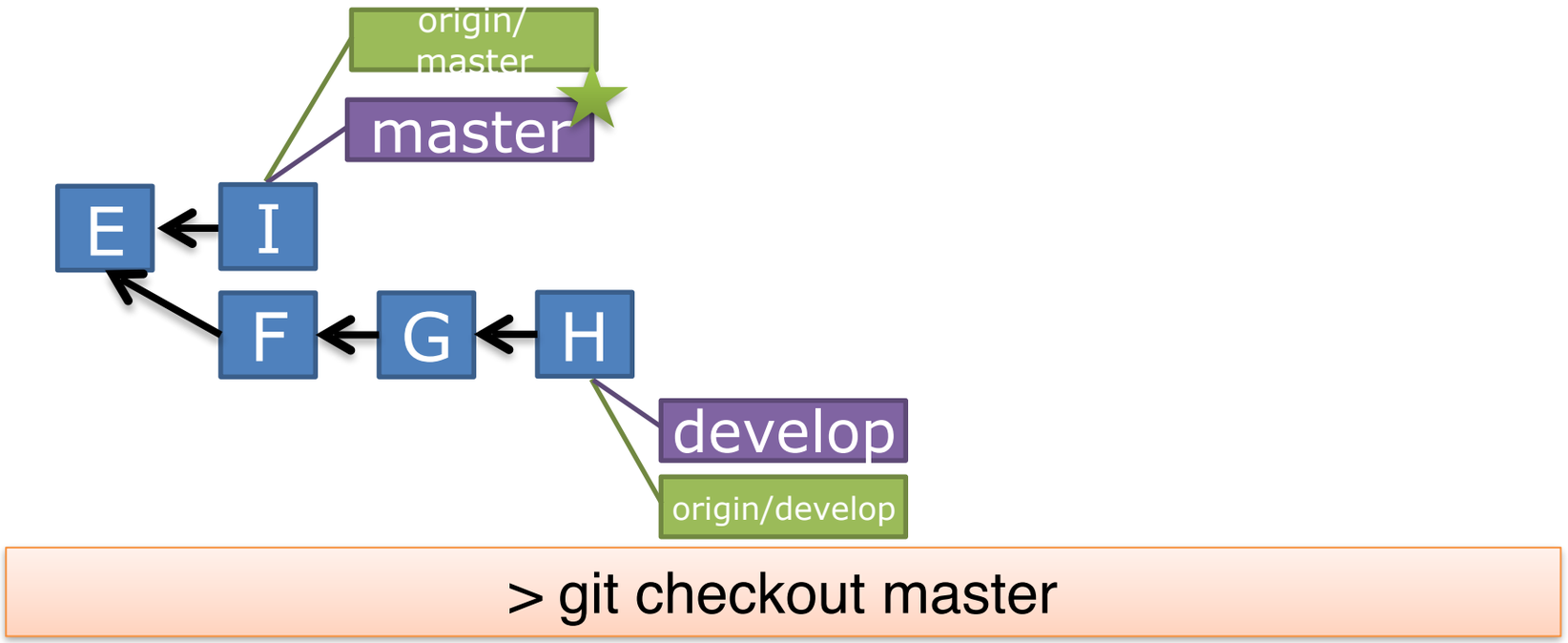
Merge vs. Rebase Workflows




```
> git push origin develop
```

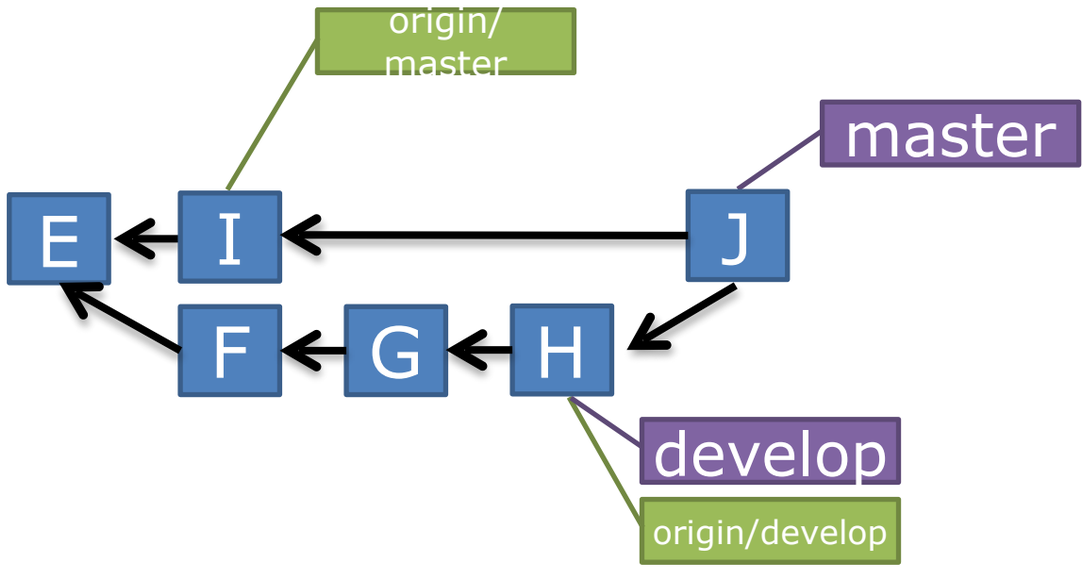
- I’m now ready to move the develop branch to production (master)
  - I can either use a:
    - merge workflow
    - or rebase workflow

Merge Flow



- Move local working copy to master

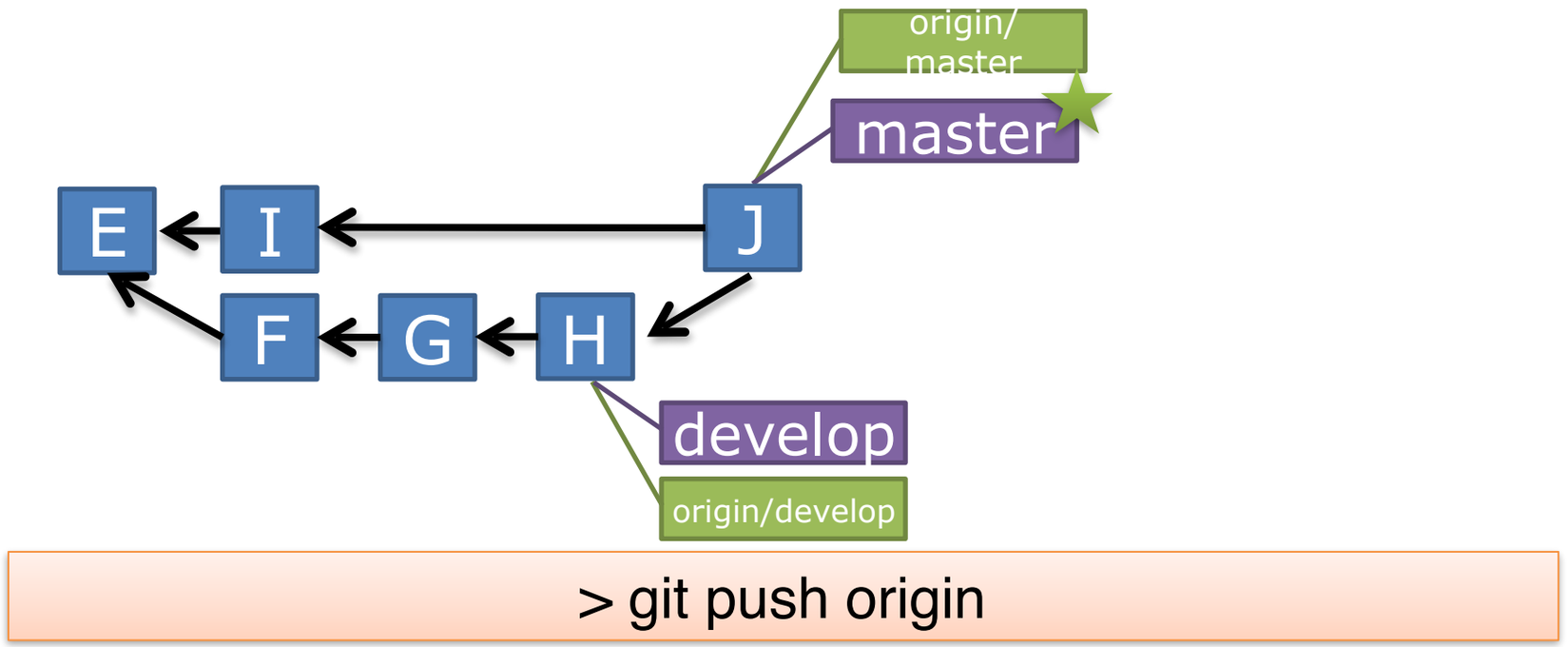
Merge Flow




```
> git merge develop
```

- Merge develop into master

Merge Flow



- Push my local master changes to origin/master