Git

A DISTRIBUTED VERSION CONTROL SYSTEM

Version control systems

- Version control (or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc.
    - Almost all “real” projects use some kind of version control
    - Essential for team projects, but also very useful for individual projects
- Some well-known version control systems are CVS, Subversion, Mercurial, and Git
    - CVS and Subversion use a “central” repository; users “check out” files, work on them, and “check them in”
    - Mercurial and Git treat all repositories as equal
- Distributed systems like Mercurial and Git are newer and are gradually replacing centralized systems like CVS and Subversion

Why version control?

- For working by yourself:
  - Gives you a “time machine” for going back to earlier versions
  - Gives you great support for different versions (standalone, web app, etc.) of the same basic project
- For working with others:
  - Greatly simplifies concurrent work, merging changes
- For getting an internship or job:
  - Any company with a clue uses some kind of version control
  - Companies without a clue are bad places to work

Why Git?

- Git has many advantages over earlier systems such as CVS and Subversion
  - More efficient, better workflow, etc.
  - See the literature for an extensive list of reasons
  - Of course, there are always those who disagree
- Best competitor: Mercurial
  - I like Mercurial better
  - Same concepts, slightly simpler to use
  - In my (very limited) experience, the Eclipse plugin is easier to install and use
  - Much less popular than Git

What is Git?

Git is a

Distributed

Version Control System

OR

Git is a

Directory

Content Management System

Git is a

Tree

history storage system

Git is a

Stupid

content tracker

How ever you think about it…

How ever you think about it…

Git is <u>SUPER</u> cool

Distributed

Everyone has the complete history

Distributed

Everyone has the complete history

Everything is done offline

…except push/pull

Distributed

Everyone has the complete history

Everything is done offline

No central authority

…except by convention

Distributed

Everyone has the complete history

Everything is done offline

No central authority

Changes can be shared without a server

Download and install Git

- There are online materials that are better than any that I could provide
- Here's the standard one: http://git-scm.com/downloads
- Note: Git is primarily a command-line tool
- Though , prefer GUIs over command-line tools, GIT GUIs are more trouble than they are worth (YMMV)

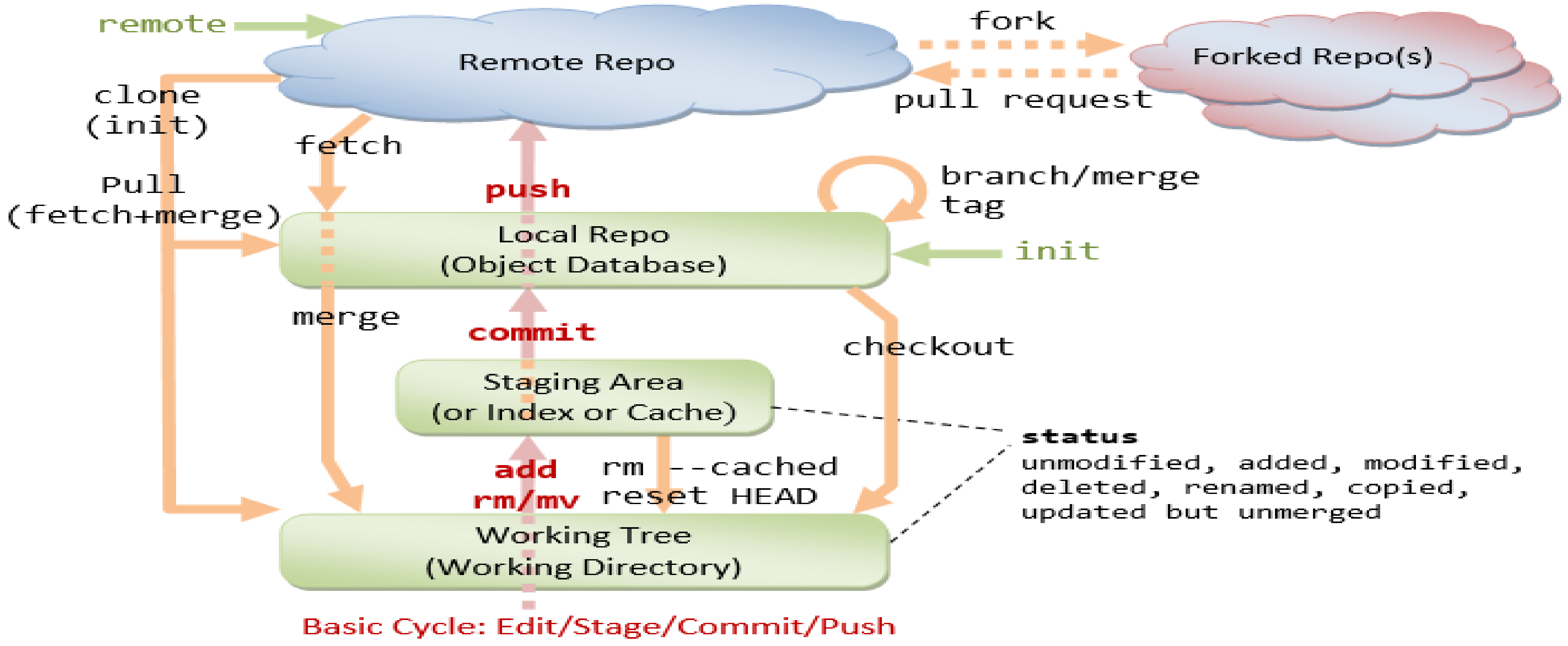
Introduce yourself to Git

- Enter these lines (with appropriate changes):
  - `git config --global user.name "John Smith"`
  - `git config --global user.email jsmith@seas.upenn.edu`
- You only need to do this once

  Git config --list

  To check with the configuration
- If you want to use a different name/email address for a particular project, you can change it for just that project
  - `cd` to the project directory
  - Use the above commands, but leave out the `--`

Git Work Flow



Create and fill a repository

1. `cd` to the project directory you want to use
2. Type in `git init`
   - This creates the repository (a directory named `.git`)
   - You seldom (if ever) need to look inside this directory
3. Type in `git add .`
   - The period at the end is part of this command!
     - Period means "this directory"
   - This adds all your current files to the repository
4. Type in `git commit -m "Initial commit"`
   - You can use a different commit message, if you like

Clone a repository from elsewhere

- `git clone *URL*`
- `git clone *URL mypath*`
  - These make an exact copy of the repository at the given URL
- `git clone git://github.com/*rest_of_path/file.git*`
  - Github is the most popular (free) public repository
- **All repositories are equal**
  - But you can treat some particular repository (such as one on Github) as the “master” directory
- **Typically, each team member works in his/her own repository, and “merges” with other repositories as appropriate**

The repository

- Your top-level working directory contains everything about your project
  - The working directory probably contains many subdirectories—source code, binaries, documentation, data files, etc.
  - One of these subdirectories, named `.git`, is your repository
- At any time, you can take a “snapshot” of everything (or selected things) in your project directory, and put it in your repository
  - This “snapshot” is called a commit object
  - The commit object contains (1) a set of files, (2) references to the “parents” of the commit object, and (3) a unique “SHA1” name
  - Commit objects do *not* require huge amounts of memory
- You can work as much as you like in your working directory, but the repository isn’t updated until you `commit` something

init and the .git repository

- When you said `git init` in your project directory, or when you cloned an existing project, you created a repository
  - The repository is a subdirectory named `.git` containing various files
  - The dot indicates a “hidden” directory
  - You do *not* work directly with the contents of that directory; various git commands do that for you
  - You *do* need a basic understanding of what is in the repository

Making commits

- You do your work in your project directory, as usual
- If you create new files and/or folders, they are *not tracked* by Git unless you ask it to do so
  - **`git add newFile1 newFolder1 newFolder2 newFile2`**
- Committing makes a "snapshot" of everything being tracked into your repository
  - A message telling what you have done is required
  - **`git commit –m "Uncrevulated the conundrum bar"`**
  - **`git commit`**
    - This version opens an editor for you the enter the message
    - To finish, save and quit the editor
- Format of the commit message
  - One line containing the complete summary
  - If more than one line, the second line must be blank

Commits and graphs

- A commit is when you tell git that a change (or addition) you have made is ready to be included in the project
- When you commit your change to git, it creates a commit object
  - A commit object represents the complete state of the project, including all the files in the project
  - The *very first* commit object has no "parents"
  - Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object
    - Hence, most commit objects have a single parent
  - You can also merge two commit objects to form a new one
    - The new commit object has two parents
- Hence, commit objects form a **directed graph**
  - Git is all about using and manipulating this graph

Working with your own repository

- A **head** is a reference to a commit object
- The "current head" is called **HEAD** (all caps)
- Usually, you will take HEAD (the current commit object), make some changes to it, and commit the changes, creating a new current commit object
  - This results in a linear graph: A → B → C → …→ HEAD
- You can also take any previous commit object, make changes to it, and commit those changes
  - This creates a branch in the graph of commit objects
- You can merge any previous commit objects
  - This joins branches in the commit graph

Commit messages

- In git, “Commits are cheap.” Do them often.
- When you commit, you must provide a one-line message stating what you have done
  - Terrible message: “Fixed a bunch of things”
  - Better message: “Corrected the calculation of median scores”
- Commit messages can be very helpful, to yourself as well as to your team members
- You can’t say much in one line, so commit often

Choose an editor

- When you “commit,” git will require you to type in a commit message
- For longer commit messages, you will use an editor
- The default editor is probably `vim`
- To change the default editor:
  - `git config --global core.editor /path/to/editor`

- You may also want to turn on colors:
  - `git config --global color.ui`

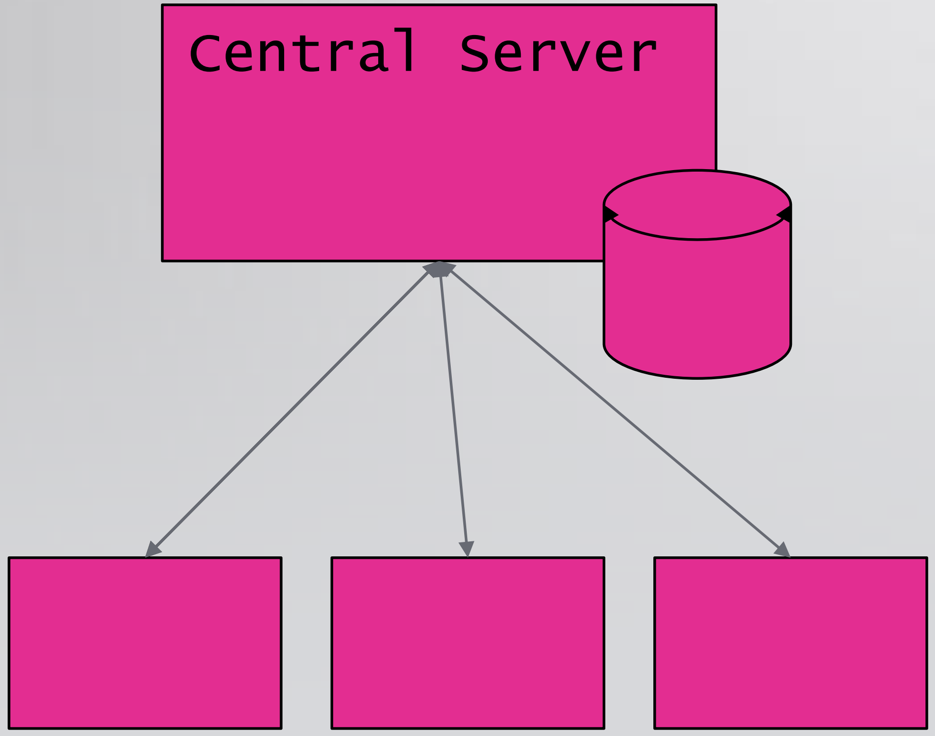
Working with others

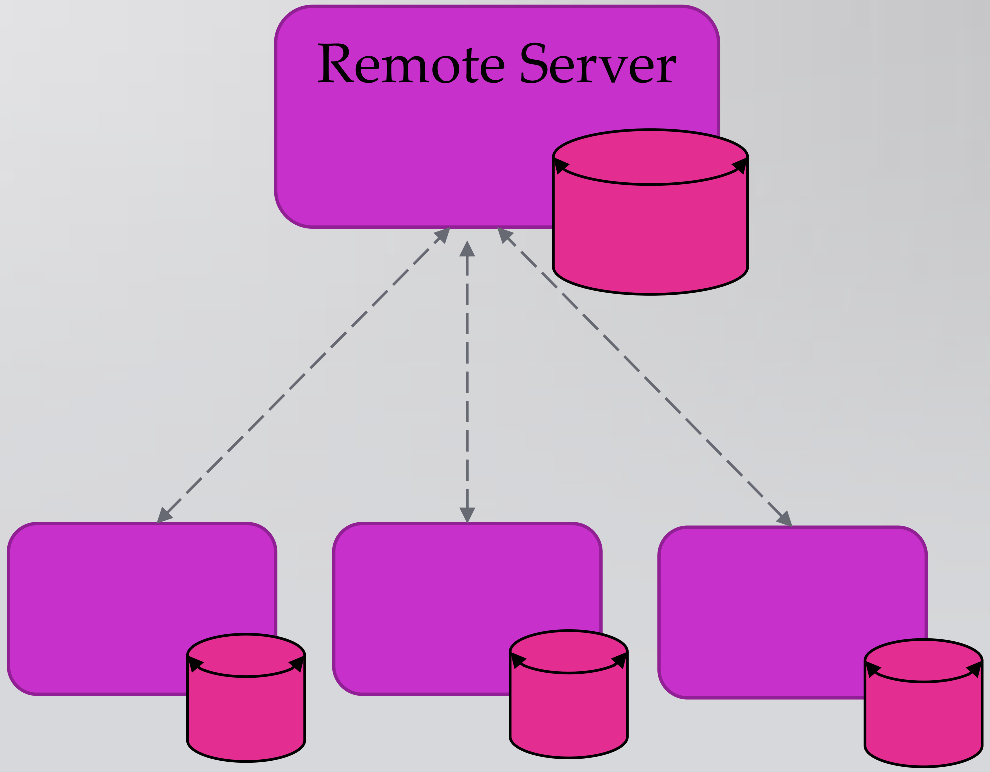
- All repositories are equal, but it is convenient to have one central repository in the cloud
- Here's what you normally do:
  - Download the current HEAD from the central repository
  - Make your changes
  - Commit your changes to your local repository
  - Check to make sure someone else on your team hasn't updated the central repository since you got it
  - Upload your changes to the central repository
- If the central repository *has* changed since you got it:
  - It is *your* responsibility to **merge your two versions**
    - This is a strong incentive to commit and upload often!
  - Git can often do this for you, if there aren't incompatible changes

Typical workflow

- `git pull` *`remote_repository`*
  - Get changes from a remote repository and merge them into your own repository
- `git status`
  - See what Git thinks is going on
  - Use this frequently!
- Work on your files (remember to `add` any new ones)
- `git commit -m` *“What I did”*
- `git push`

Centralized VC vs. Distributed VC





Branching

Branching

Forget what you know from Central VC (…TFS, SVN, Perforce...)

Branching

Forget what you know from Central VC

Git branch is “Sticky Note” on a graph node

Branching

Forget what you know from Central VC

Git branch is “Sticky Note” on a graph node

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

Branching

Forget what you know from Central VC

Git branch is “Sticky Note” on the graph

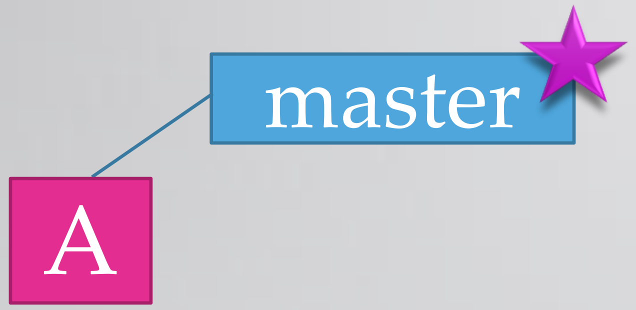
All branch work takes place within the same folder within your file system.

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

Initialization

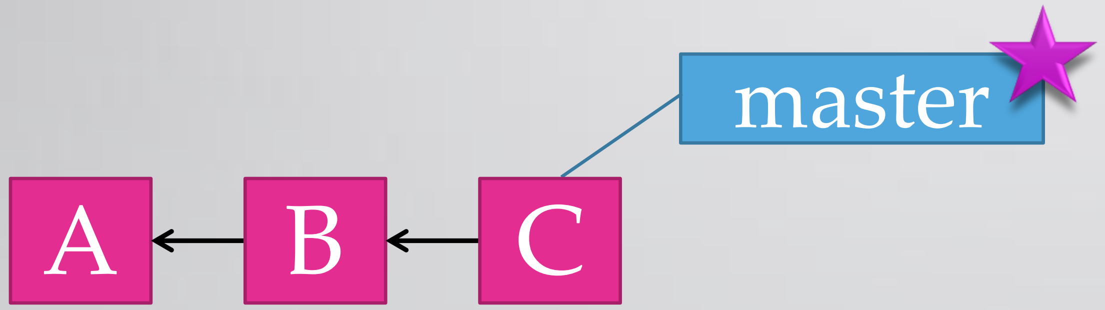
```
C:\> mkdir CoolProject
C:\> cd CoolProject
C:\CoolProject > git init
Initialized empty Git repository in
C:/CoolProject/.git
C:\CoolProject > notepad README.txt
C:\CoolProject > git add .
C:\CoolProject > git commit -m 'my first
commit'
[master (root-commit) 7106a52] my first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

Branches Illustrated



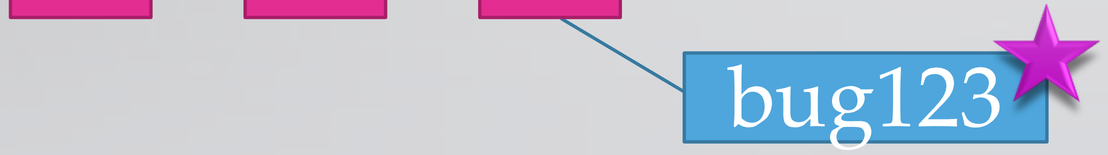
```
> git commit –m ‘my first commit’
```

Branches Illustrated



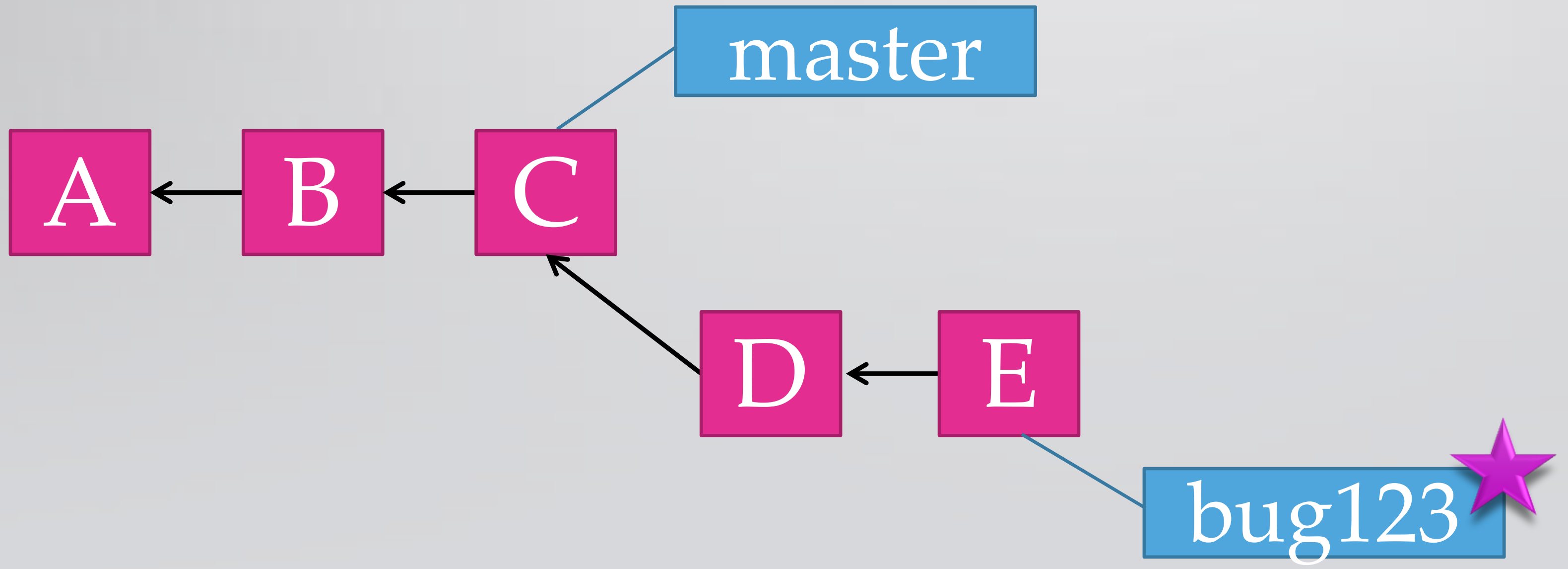
```
> git commit (x2)
```

Branches Illustrated



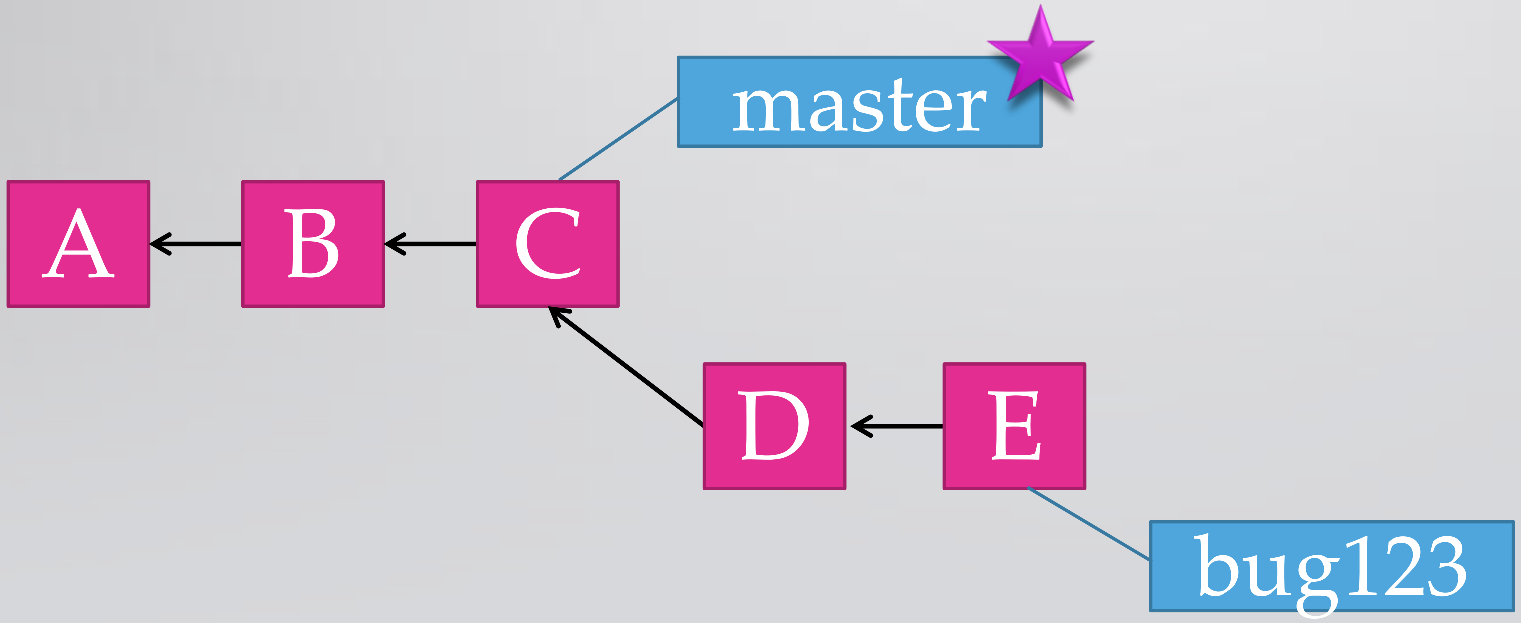
```
> git checkout -b bug123
```

Branches Illustrated



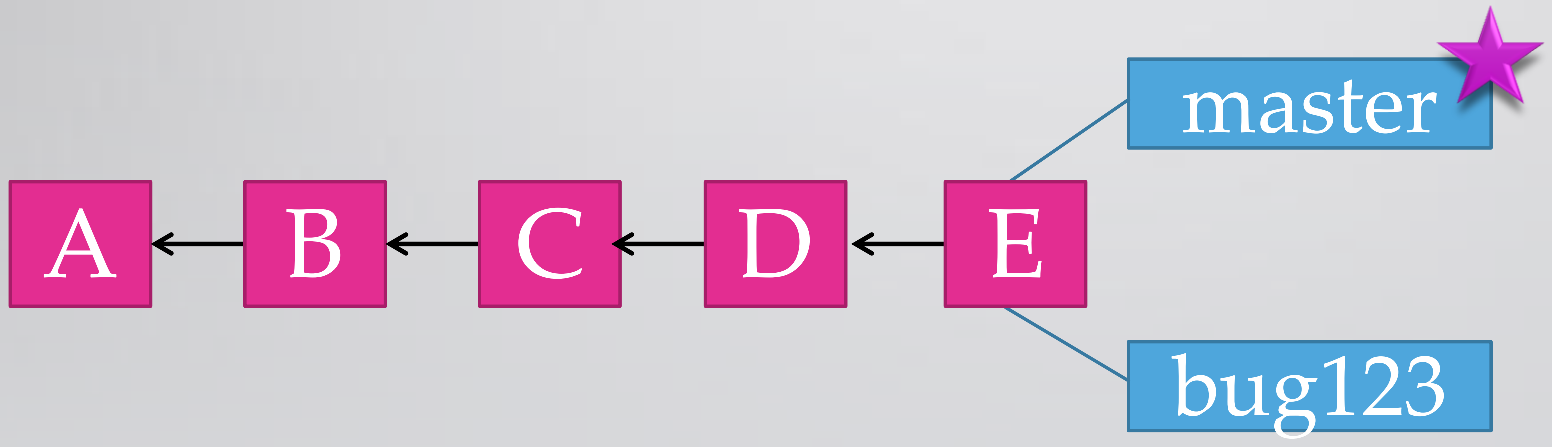
```
> git commit (x2)
```

Branches Illustrated



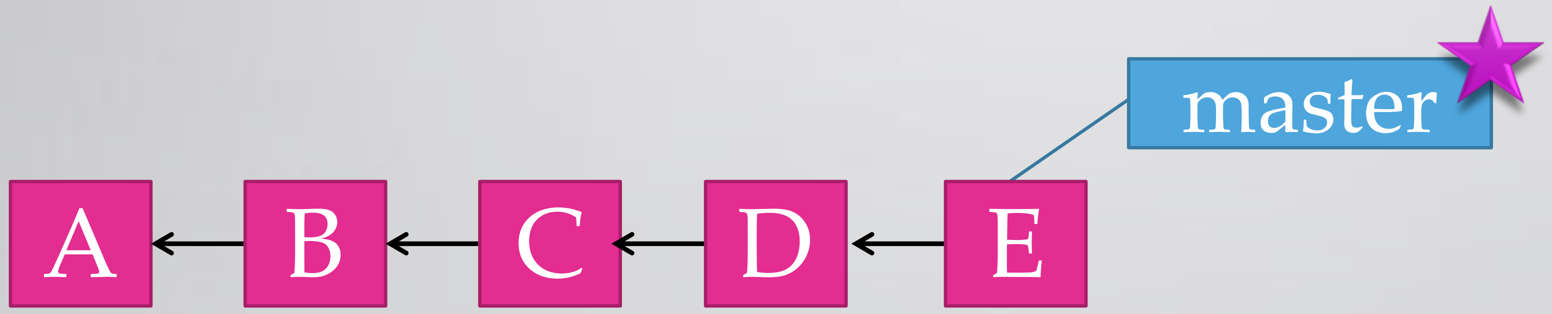
```
> git checkout master
```

Branches Illustrated



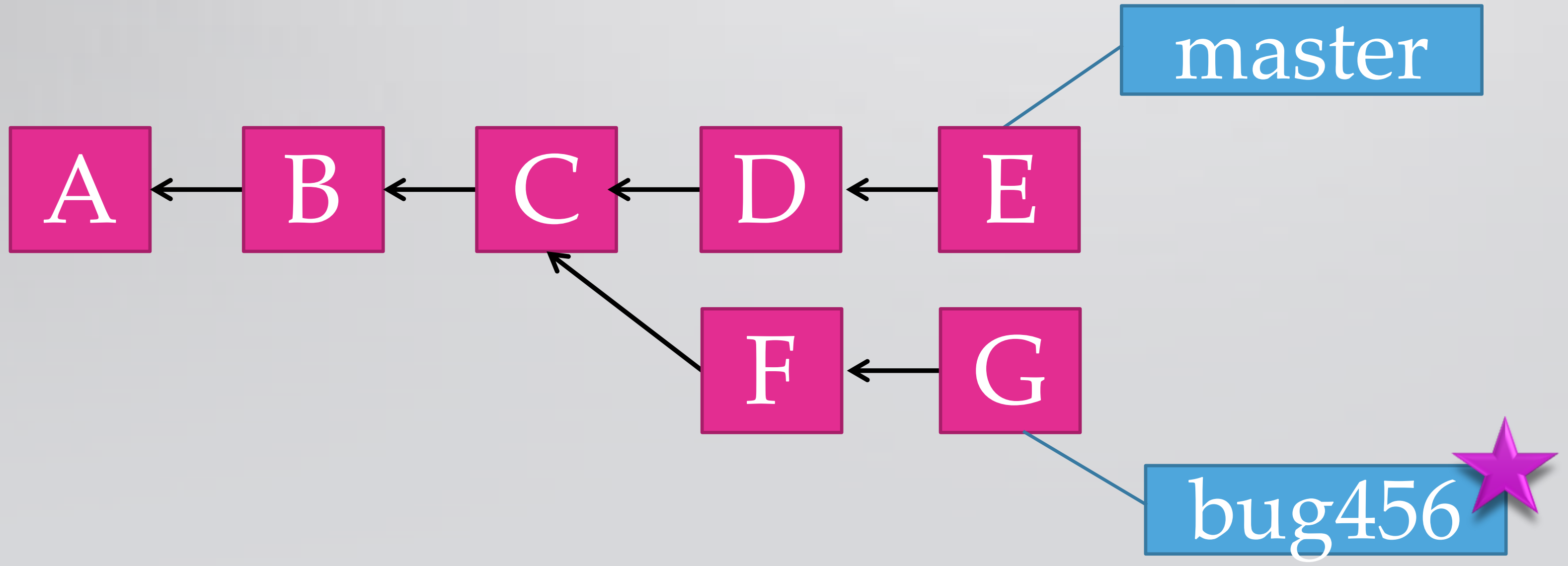
```
> git merge bug123
```

Branches Illustrated

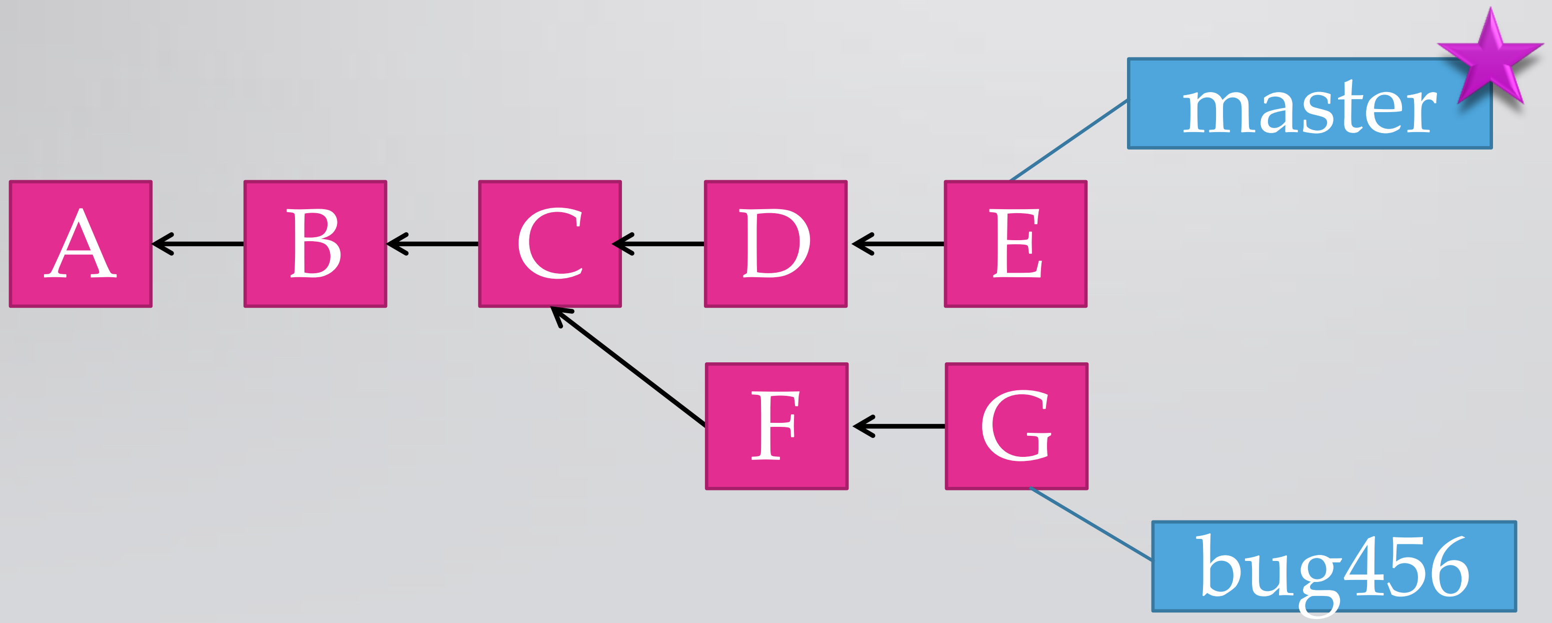


```
> git branch -d bug123
```

Branches Illustrated

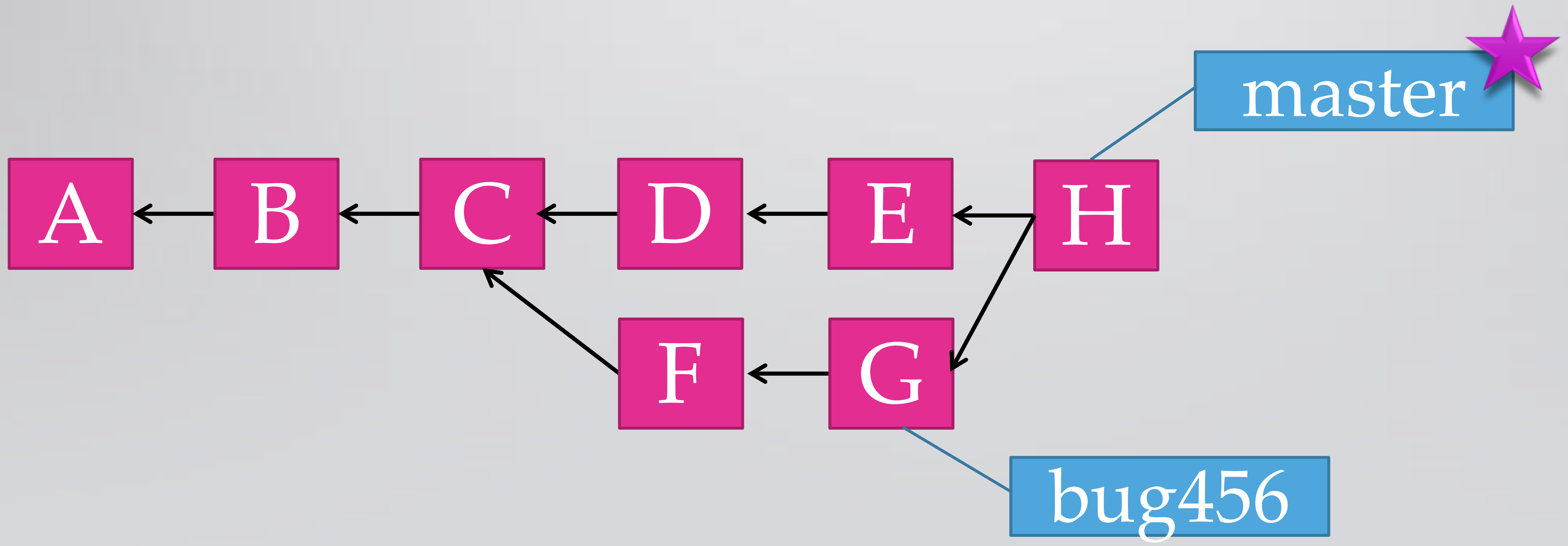


Branches Illustrated



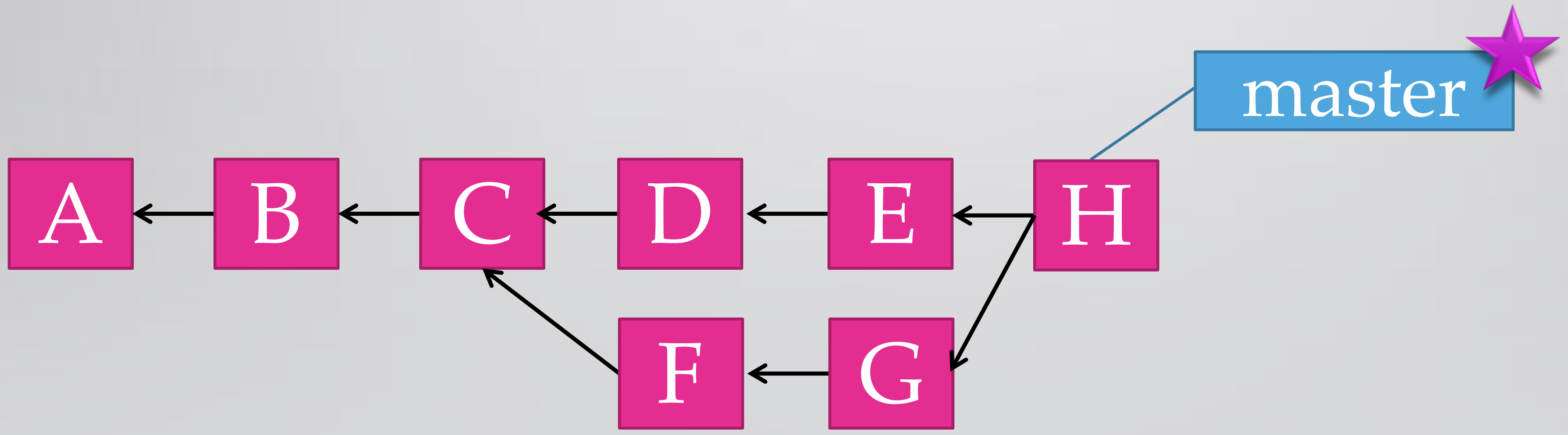
```
> git checkout master
```

Branches Illustrated



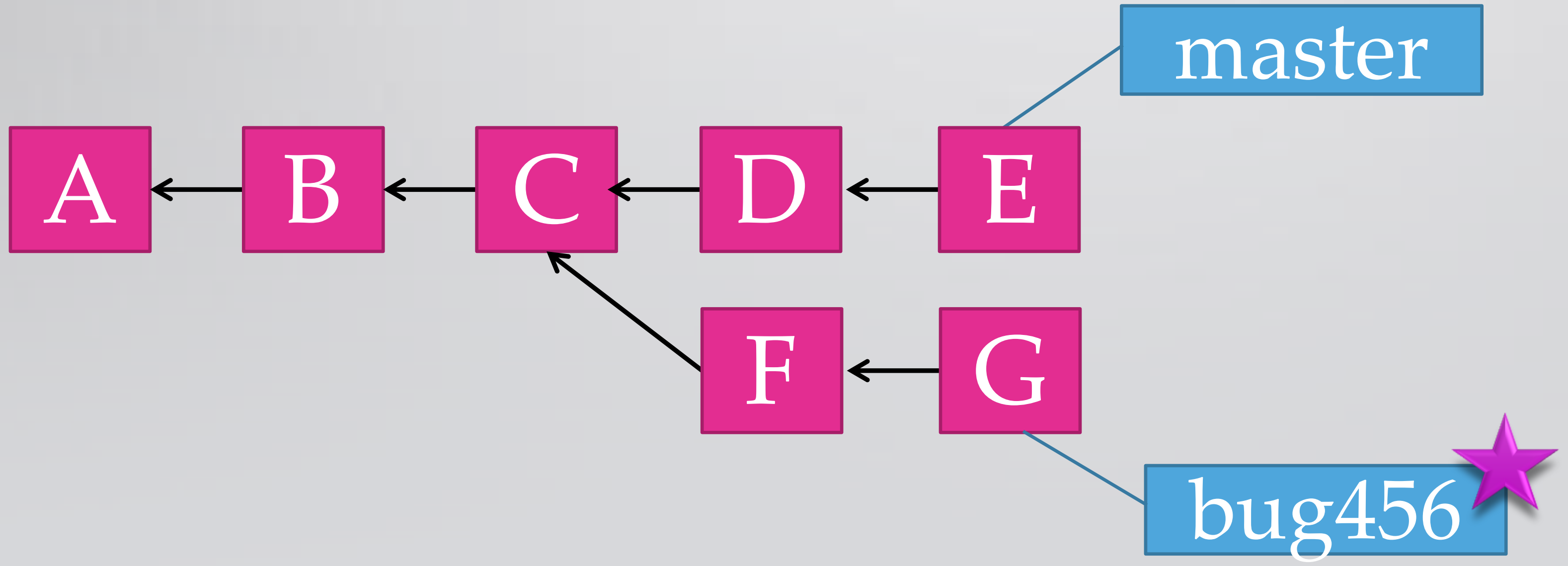
```
> git merge bug456
```

Branches Illustrated

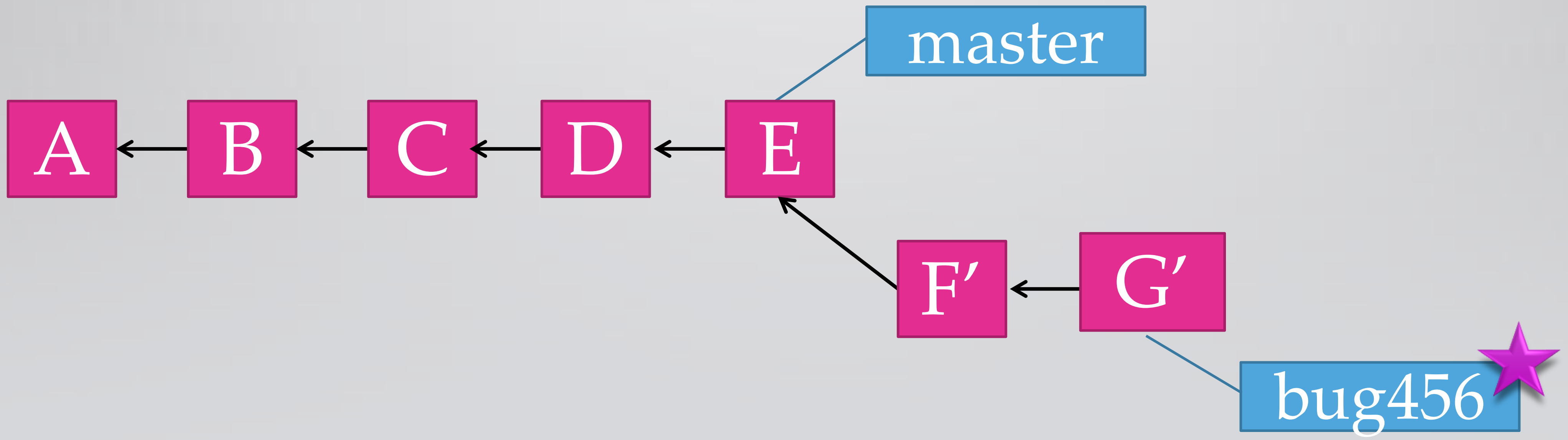


```
> git branch -d bug456
```

Branches Illustrated

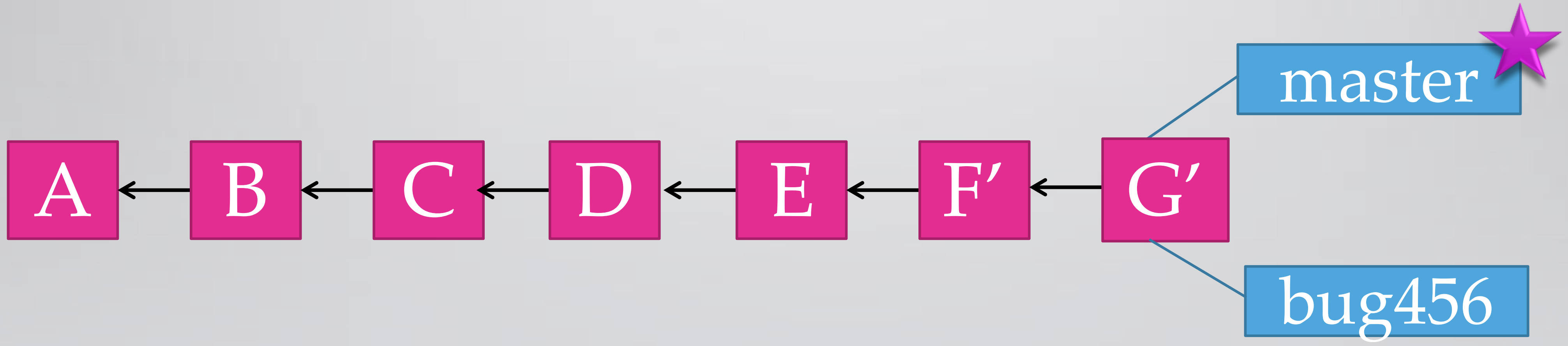


Branches Illustrated



```
> git rebase master
```

Branches Illustrated



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

Branching Review

Branching Review

Quick and Easy to create ‘Feature’ Branches

Branching Review

Quick and Easy to create ‘Feature’ Branches

Local branches are very powerful

Branching Review

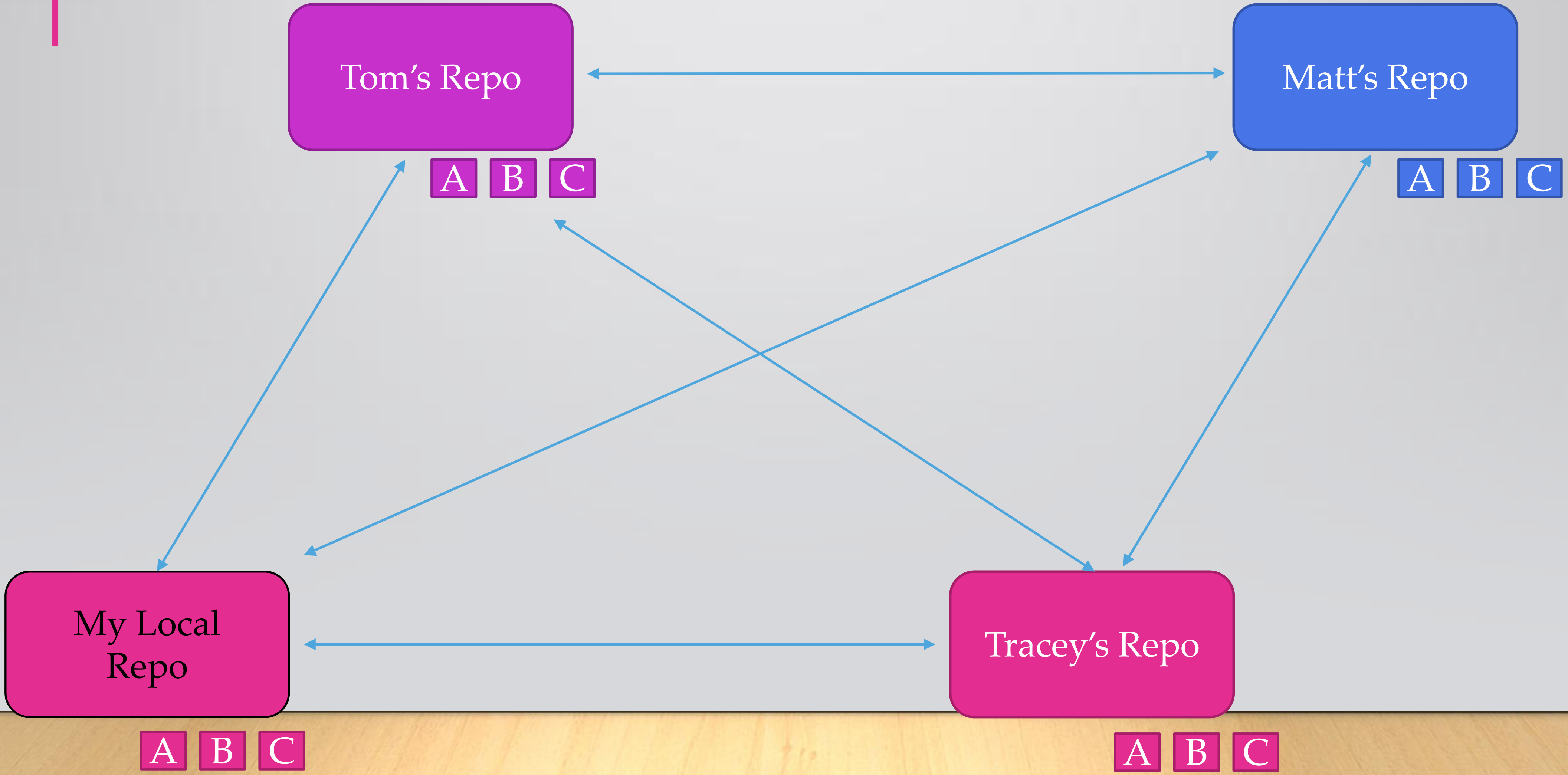
Quick and Easy to create ‘Feature’ Branches

Local branches are very powerful

Rebase is not scary

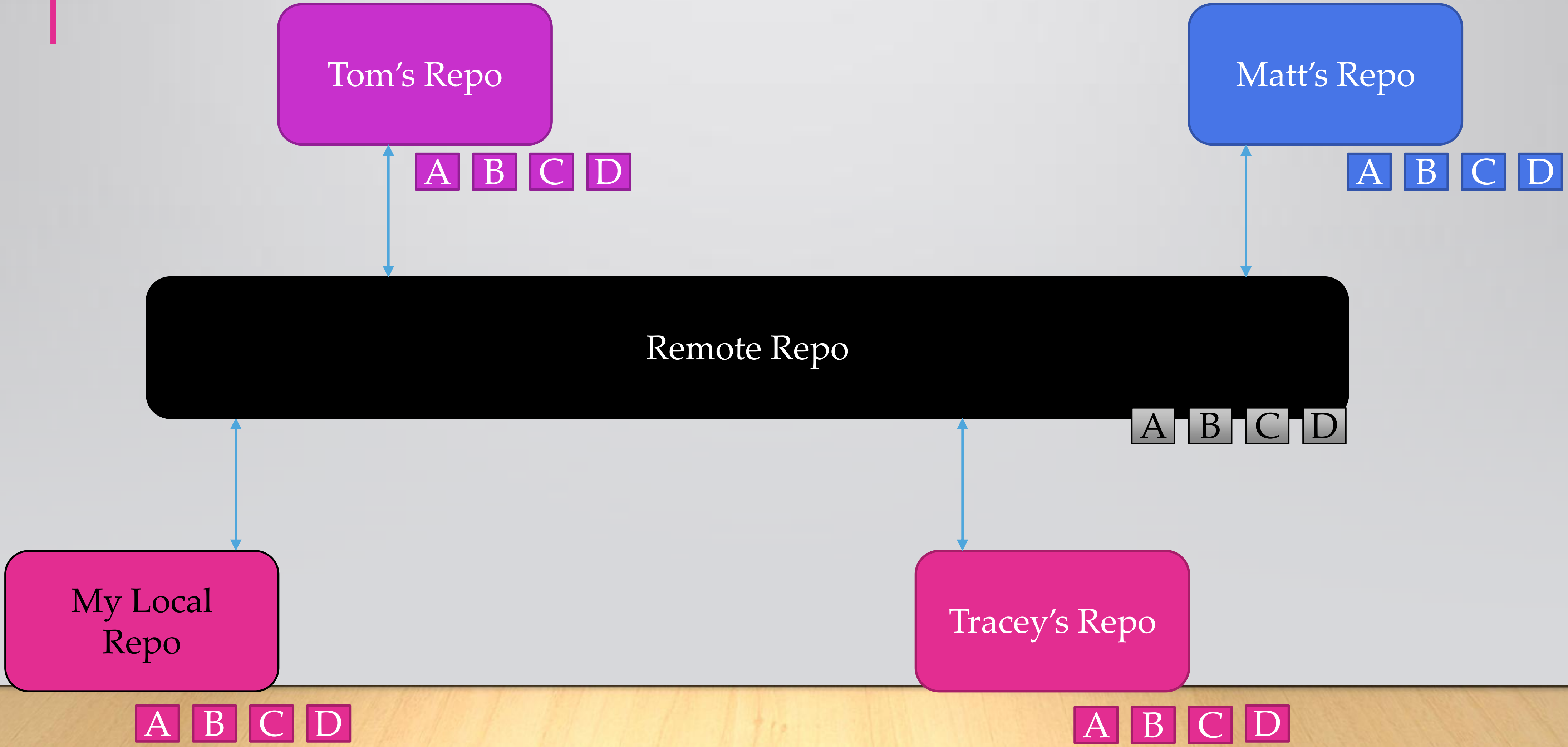
Software is a Team Sport

Sharing commits



Adding a Remote

Sharing commits



Setting up a Remote

Setting up a Remote

Adding a remote to an existing local repo

```
C:\CoolProject > git remote add origin https://git01.codeplex.com/coolproject
C:\CoolProject > git remote -v
origin  https://git01.codeplex.com/coolproject (fetch)
origin  https://git01.codeplex.com/coolproject (push)
```

Setting up a Remote

Clone will auto setup the remote

```
C:\> git clone https://git01.codeplex.com/coolproject
Cloning into 'coolproject'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
C:\> cd .\coolproject
C:\CoolProject> git remote -v
origin  https://git01.codeplex.com/coolproject (fetch)
origin  https://git01.codeplex.com/coolproject (push)
```

Setting up a Remote

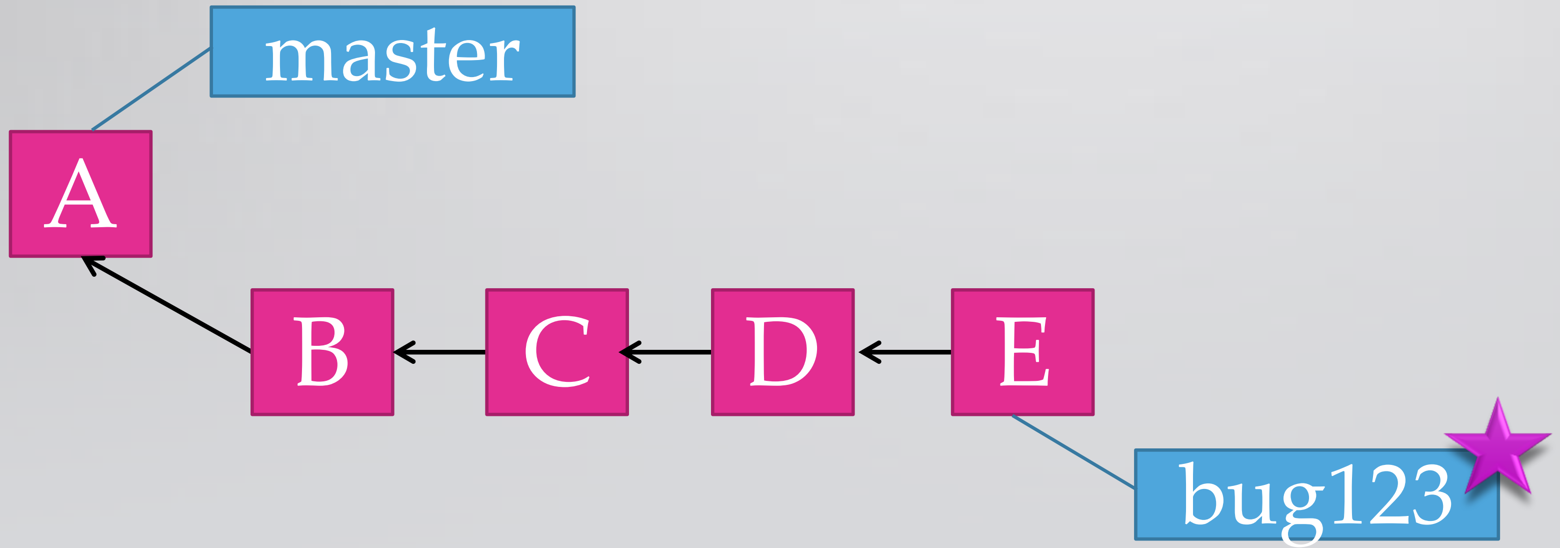
Name remotes what you want

Setting up a Remote

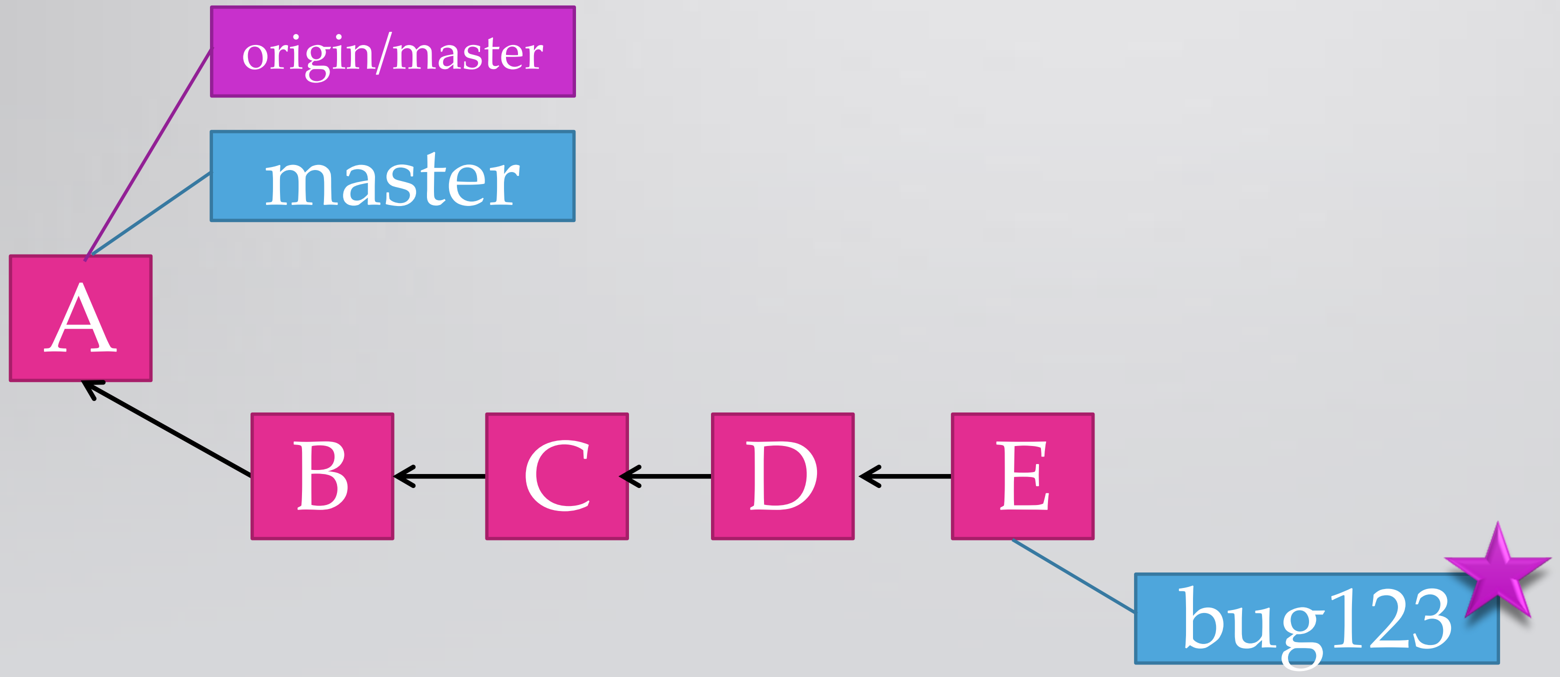
Name remotes what you want

Origin is only a convention

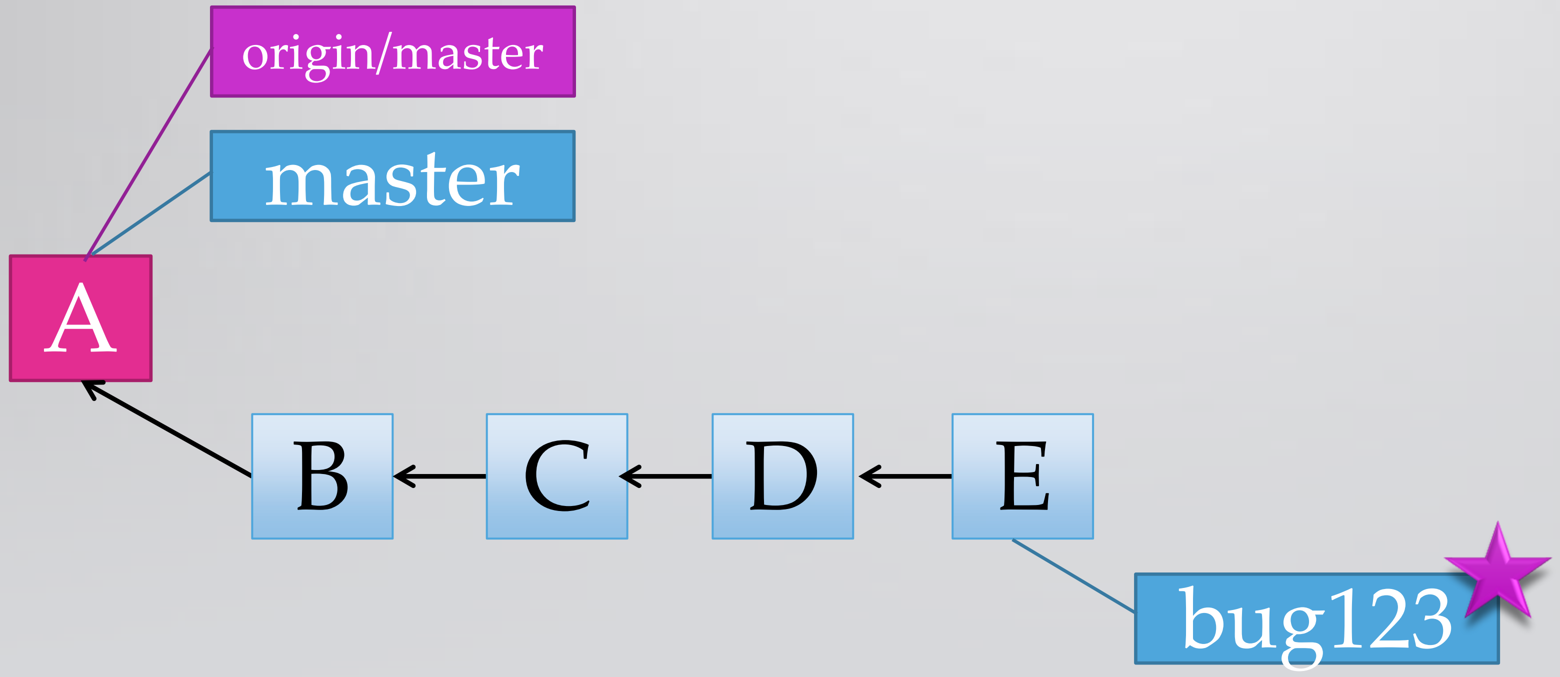
Branches Illustrated



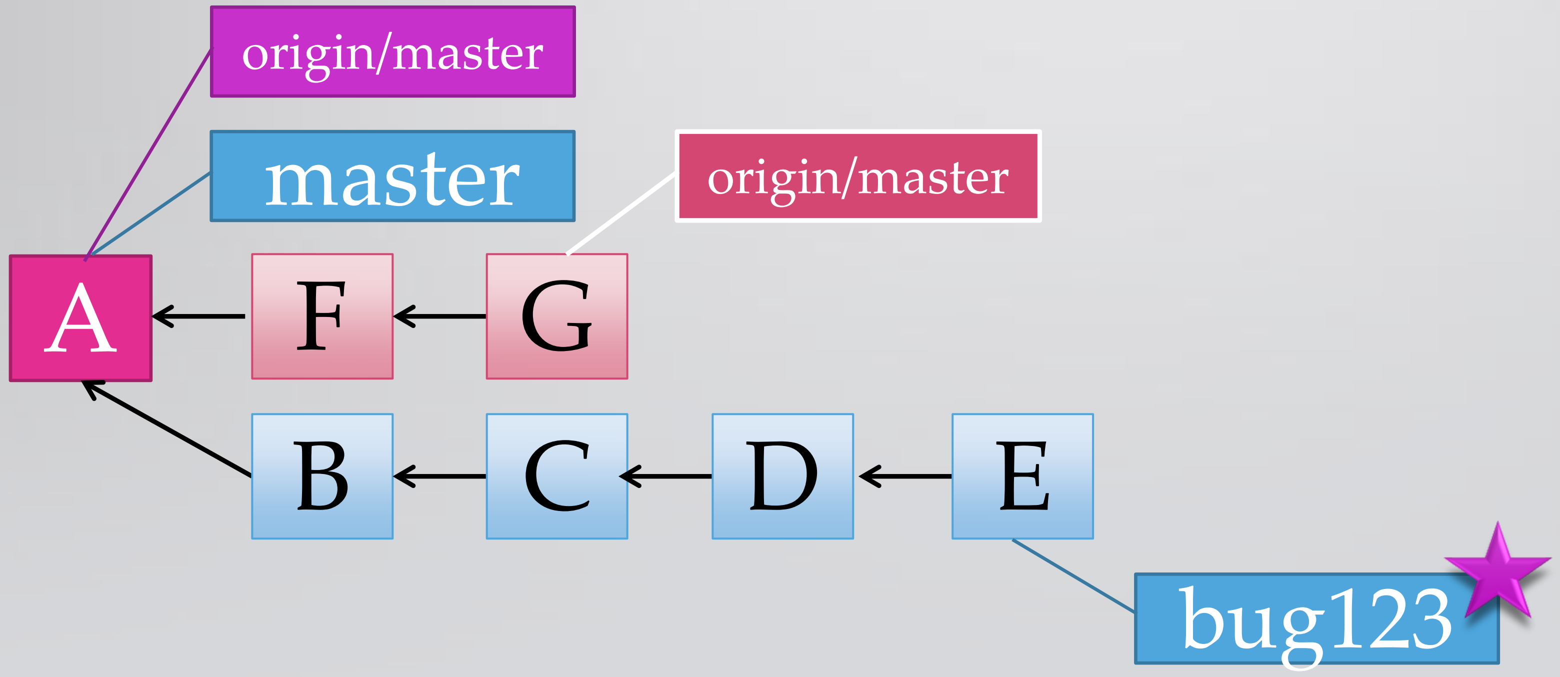
Branches Illustrated



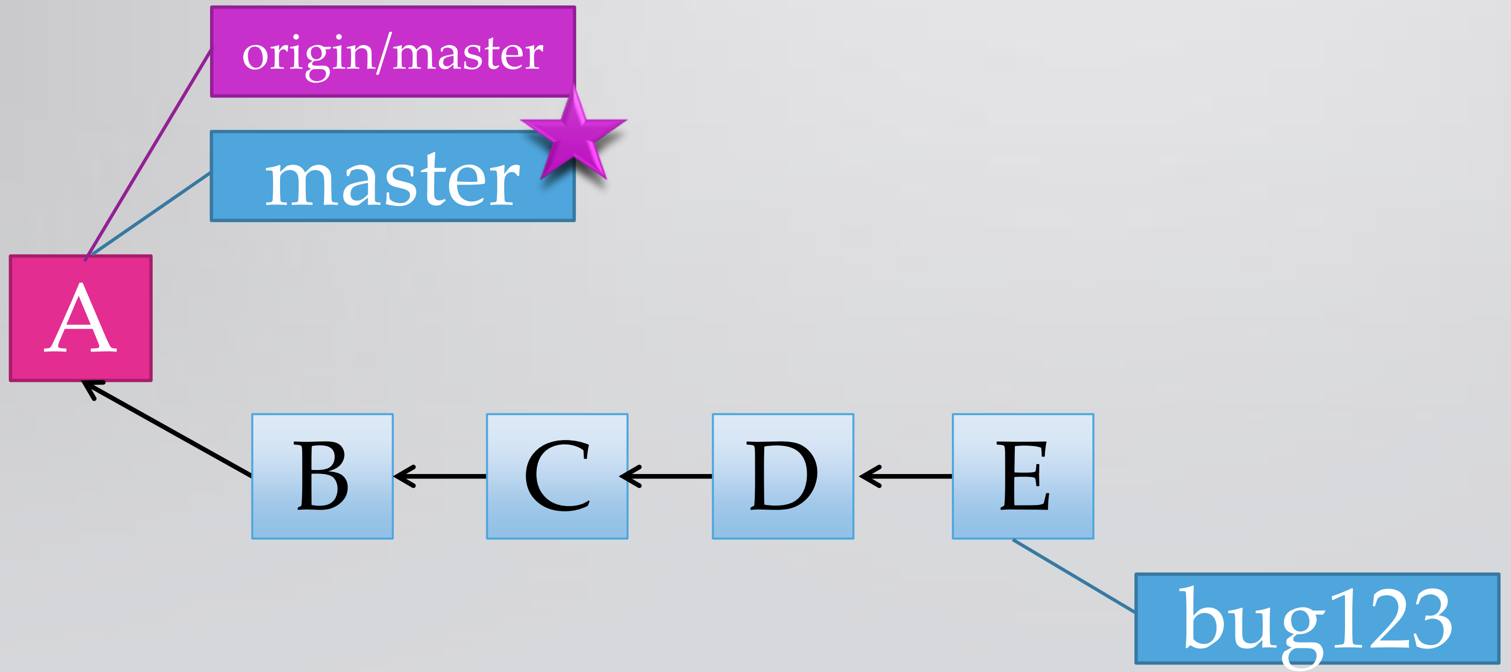
Branches Illustrated



Branches Illustrated

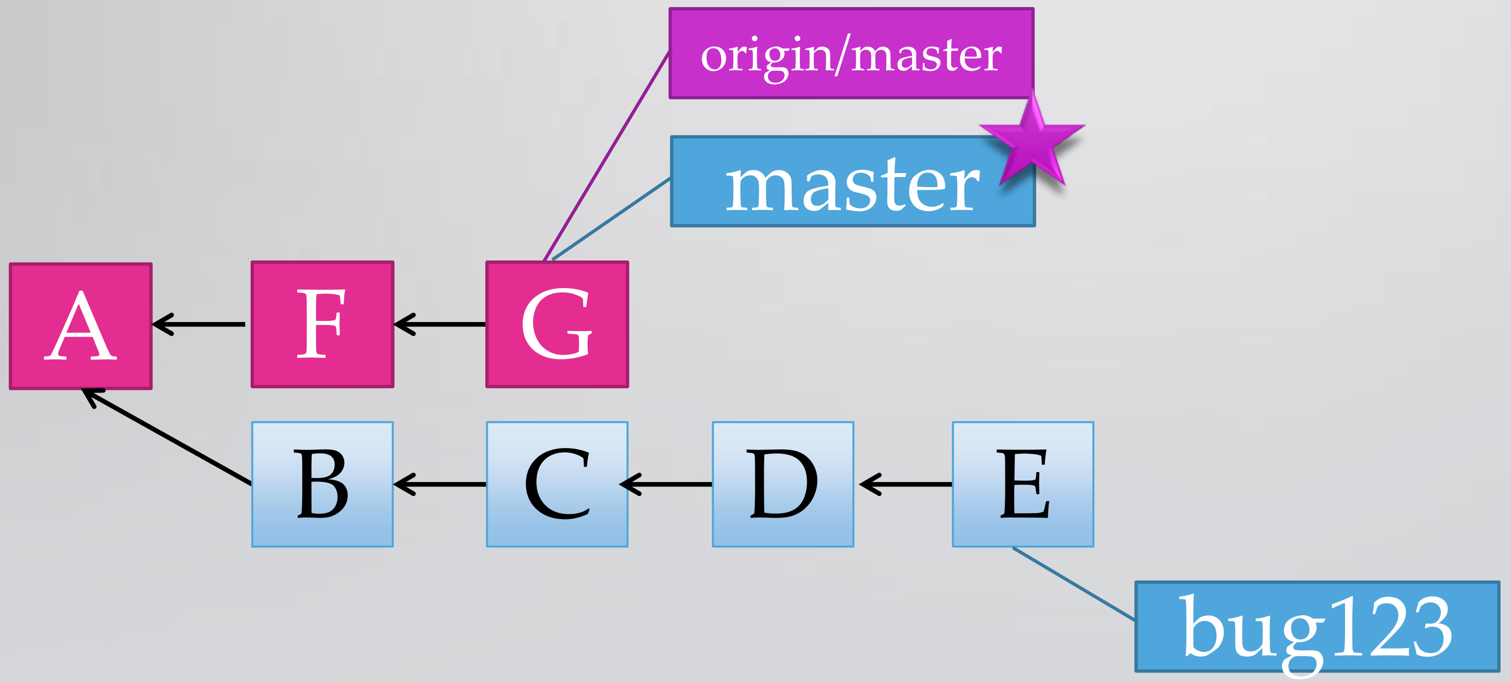


Branches Illustrated



```
> git checkout master
```

Branches Illustrated



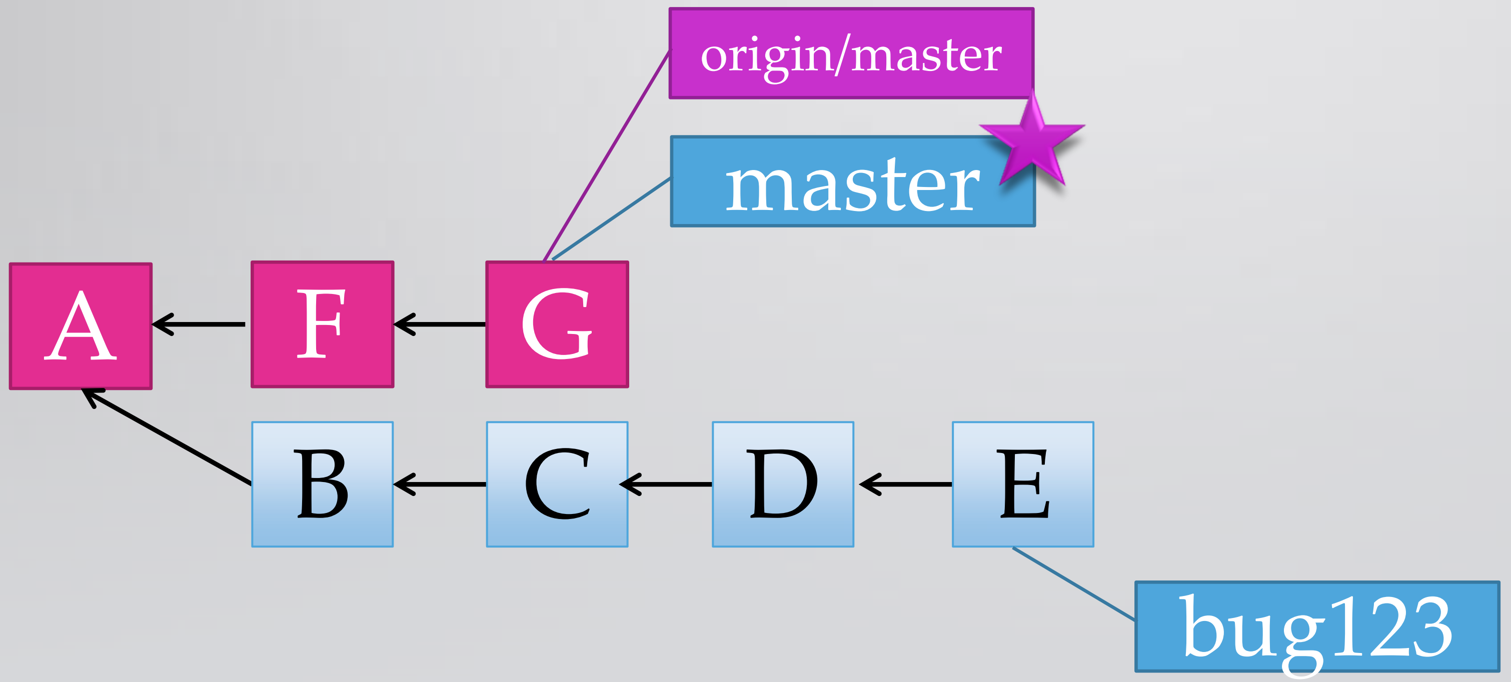
```
> git pull origin
```

Pull = Fetch + Merge

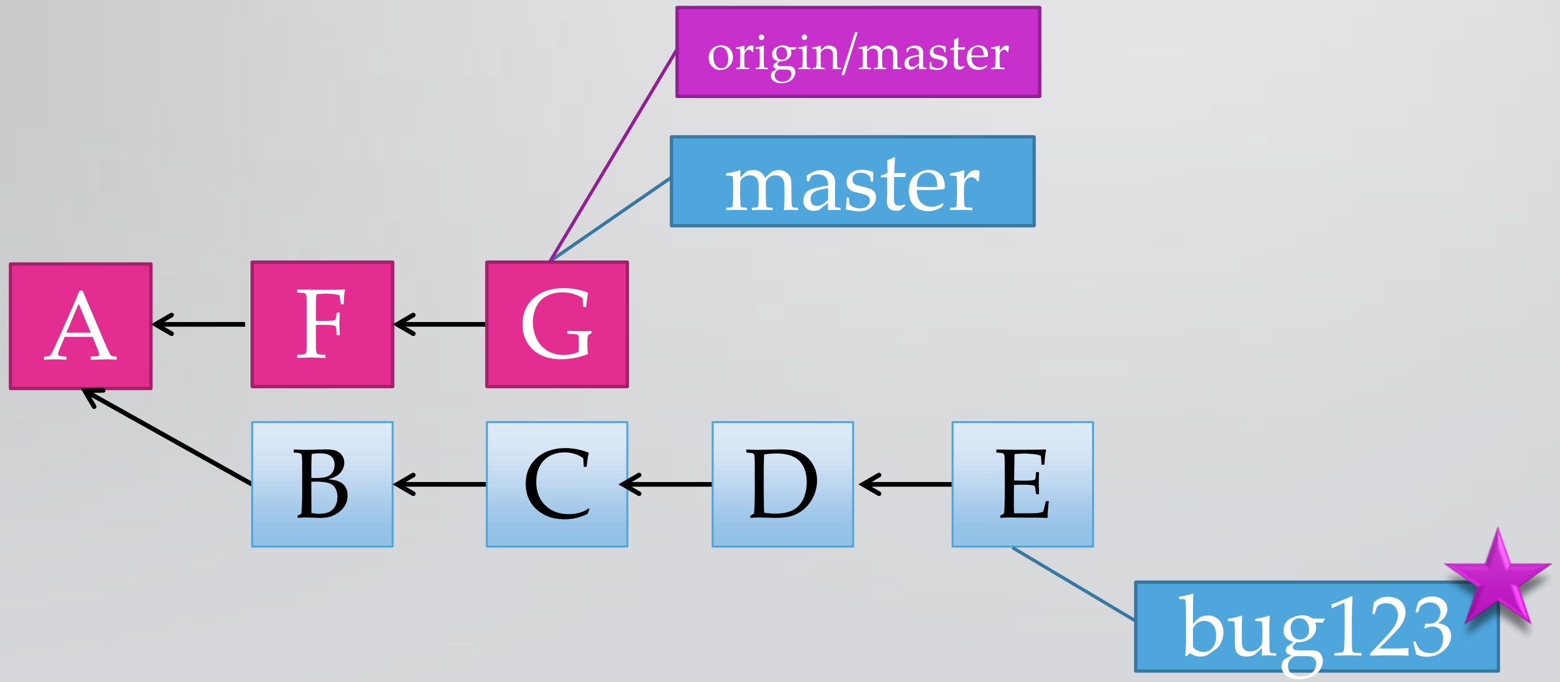
Fetch - updates your local copy of the remote branch

Pull essentially does a fetch and then runs the merge in one step.

Branches Illustrated

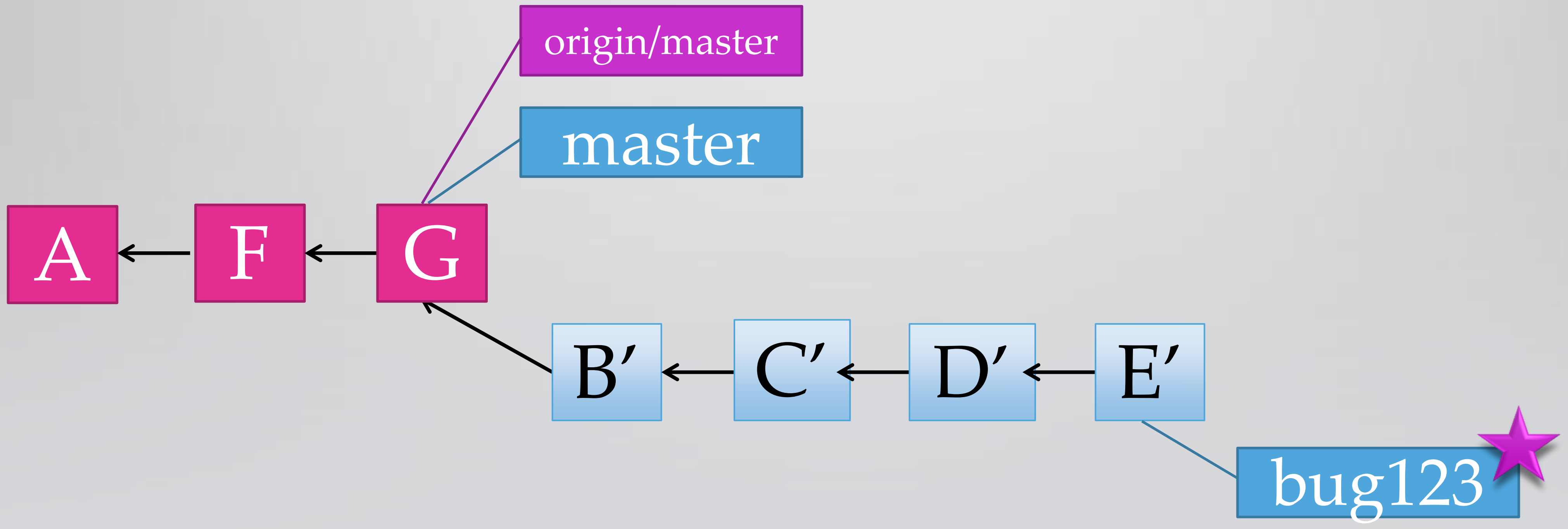


Branches Illustrated



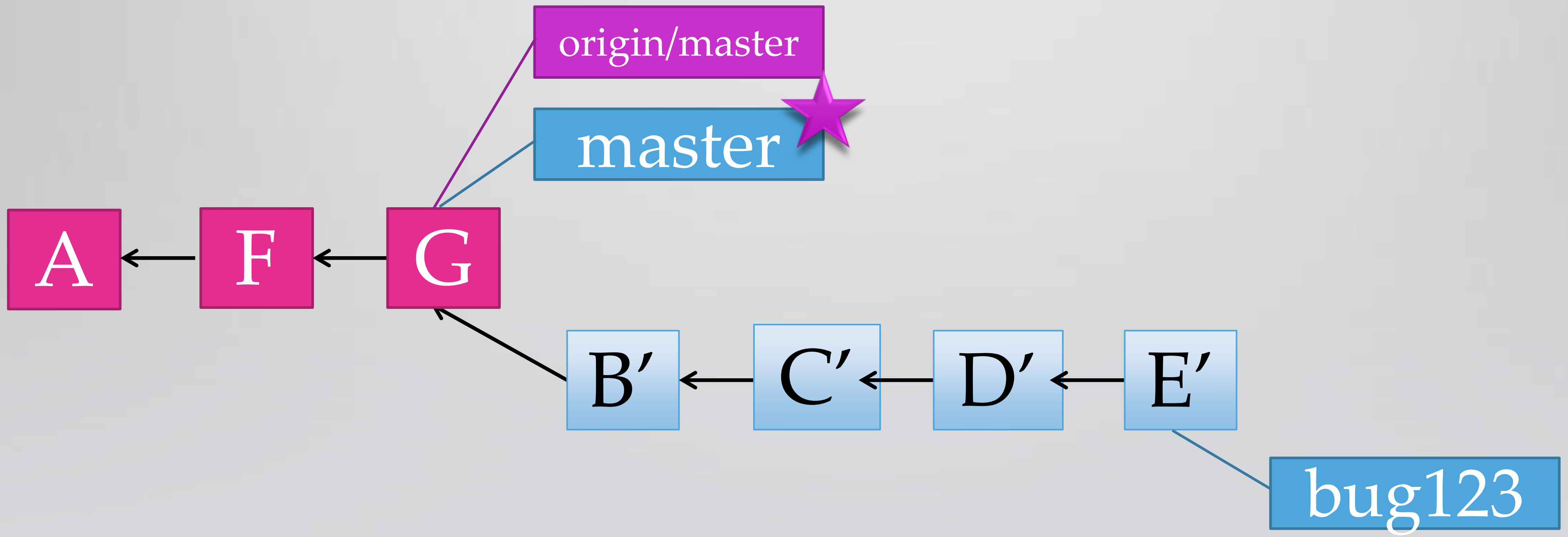
```
> git checkout bug123
```

Branches Illustrated



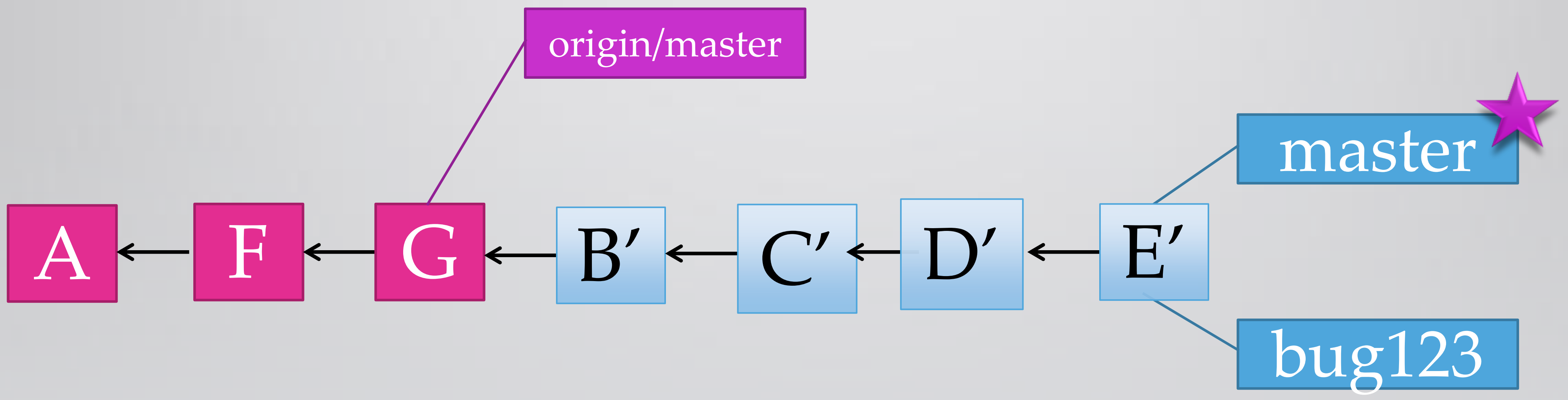
```
> git rebase master
```

Branches Illustrated



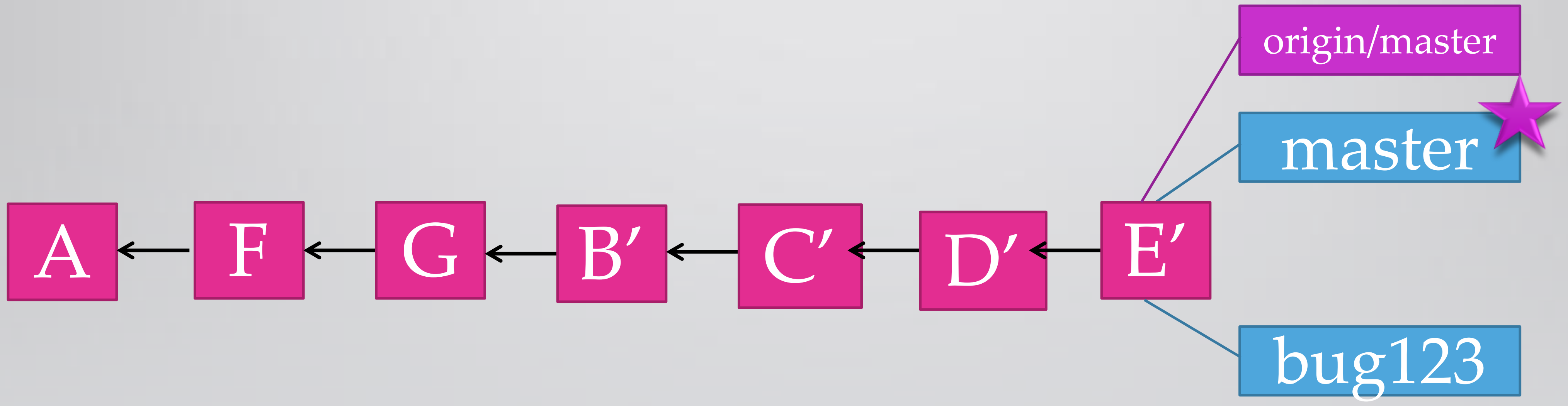
```
> git checkout master
```

Branches Illustrated



```
> git merge bug123
```

Branches Illustrated



```
> git push origin
```

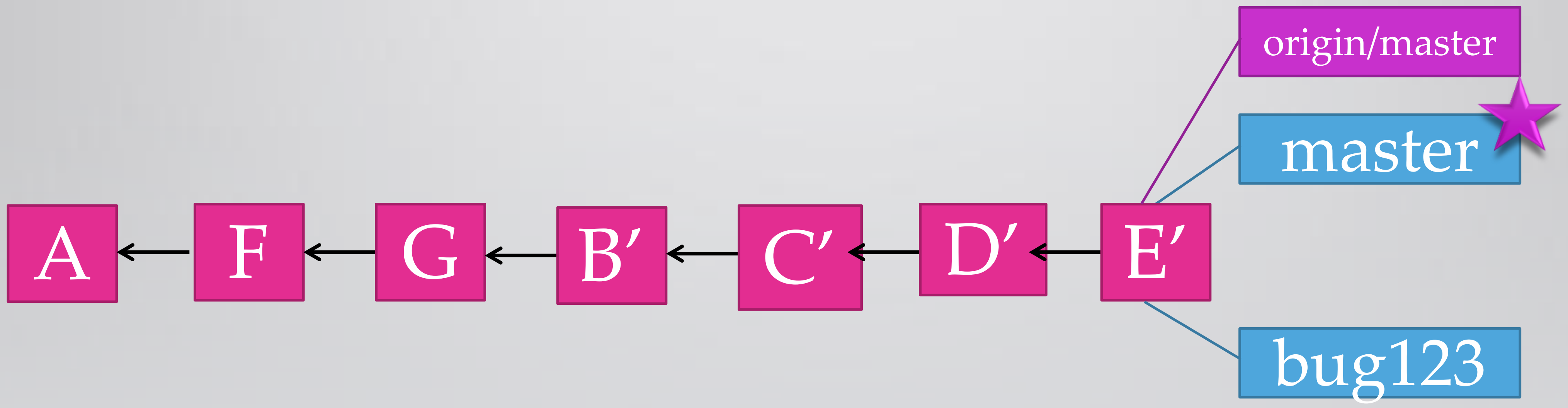
Push

Pushes your changes upstream

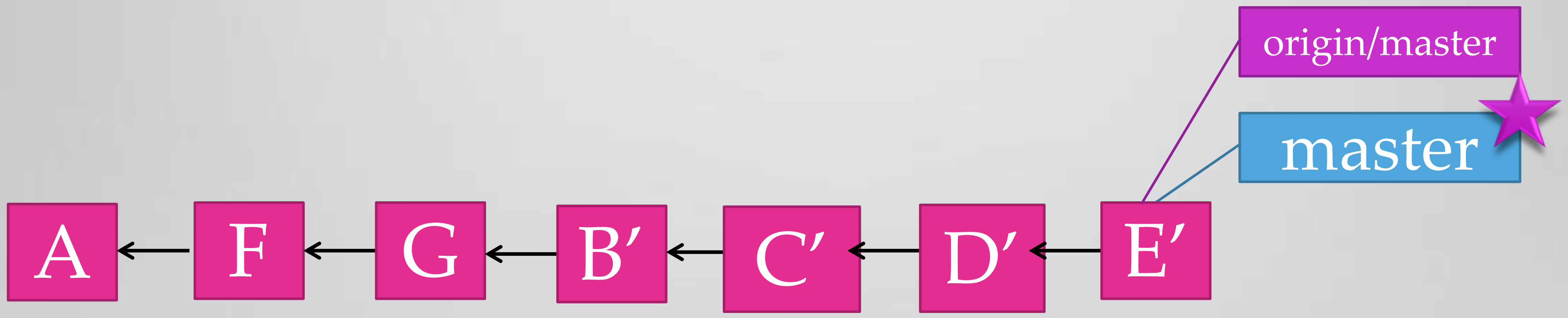
Git will reject pushes if newer changes exist on remote.

- Good practice: Pull then Push

Branches Illustrated



Branches Illustrated



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

Adding a Remote Review

Adding a remote makes it easy to share

Pulling from the remote often helps keep you up to date

Short vs. Long-Lived Branches

Local branches are short lived

<u>Short</u> vs. Long-Lived Branches

Local branches are short lived

Staying off master keeps merges simple

Short vs. Long-Lived Branches

Local branches are short lived

Staying off master keeps merges simple

Enables working on several changes at once

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

Short vs. <u>Long-Lived</u> Branches

Great for multi-version work

Follow same rules as Master

Short vs. Long-Lived Branches

Great for multi-version work
Follow same rules as Master…Story branches

Short vs. Long-Lived Branches

Great for multi-version work

Follow same rules as Master…Story branches

Integrate frequently

Short vs. Long-Lived Branches

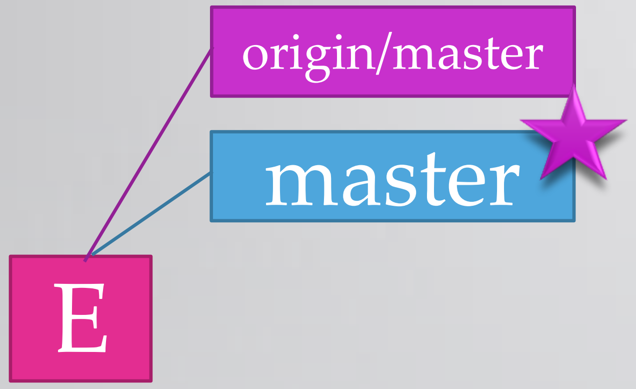
Great for multi-version work

Follow same rules as Master…Story branches

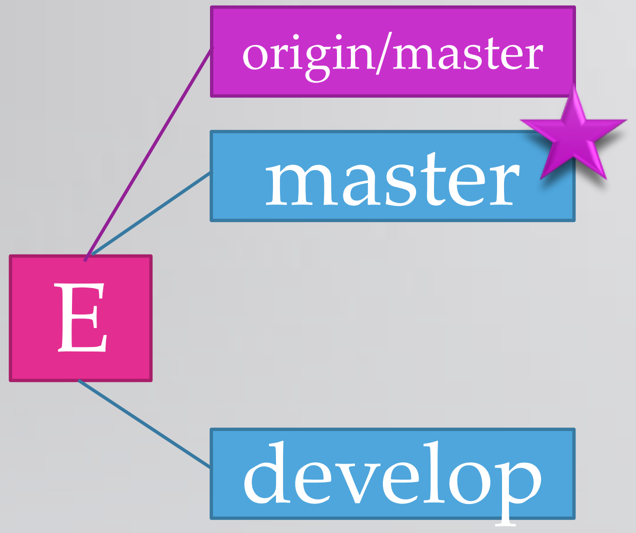
Integrate frequently

Pushed to Remotes

Branches Illustrated

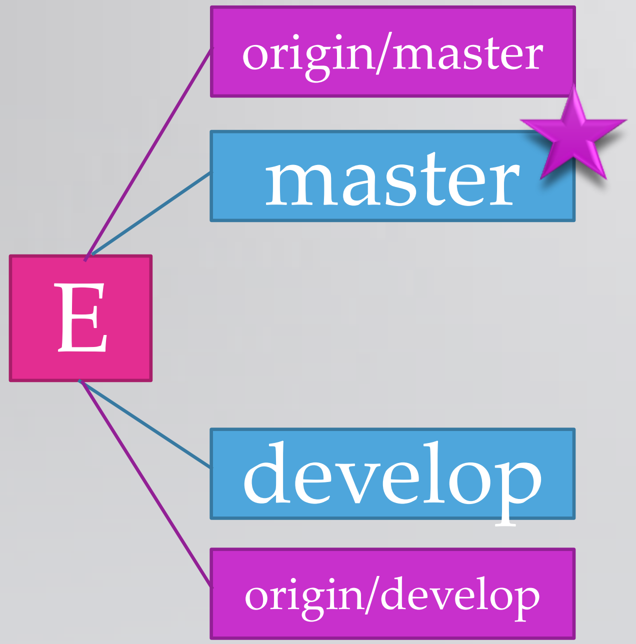


Branches Illustrated



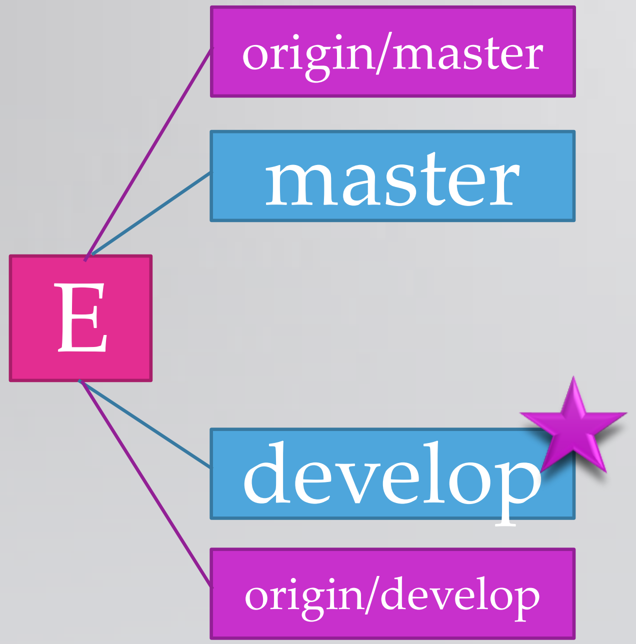
```
> git branch develop
```

Branches Illustrated



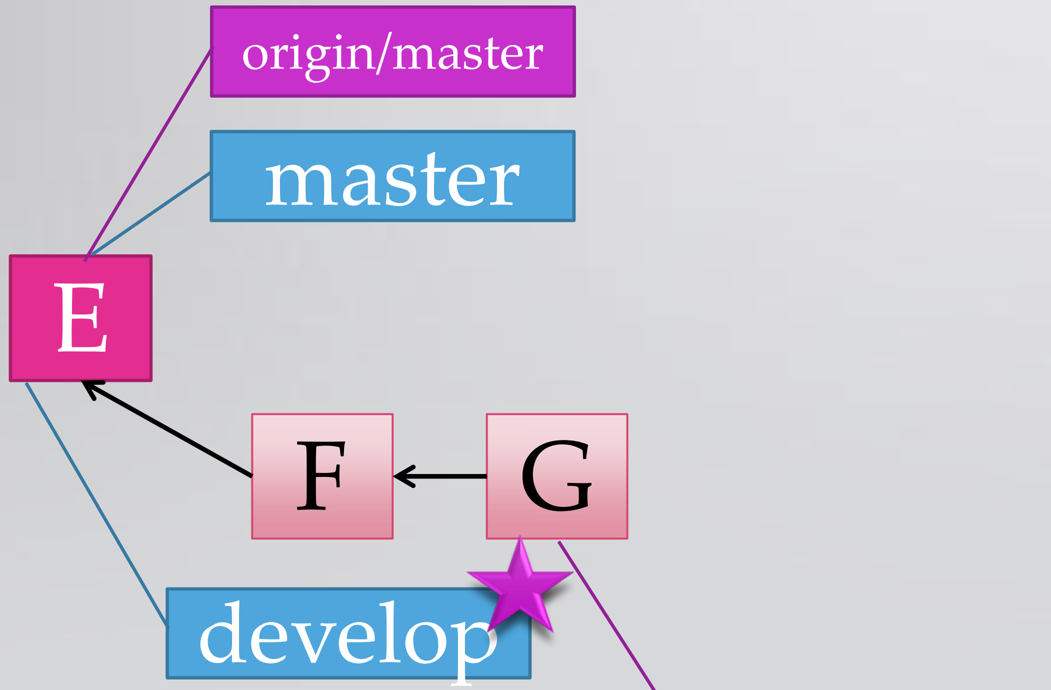
```
> git push origin develop
```

Branches Illustrated

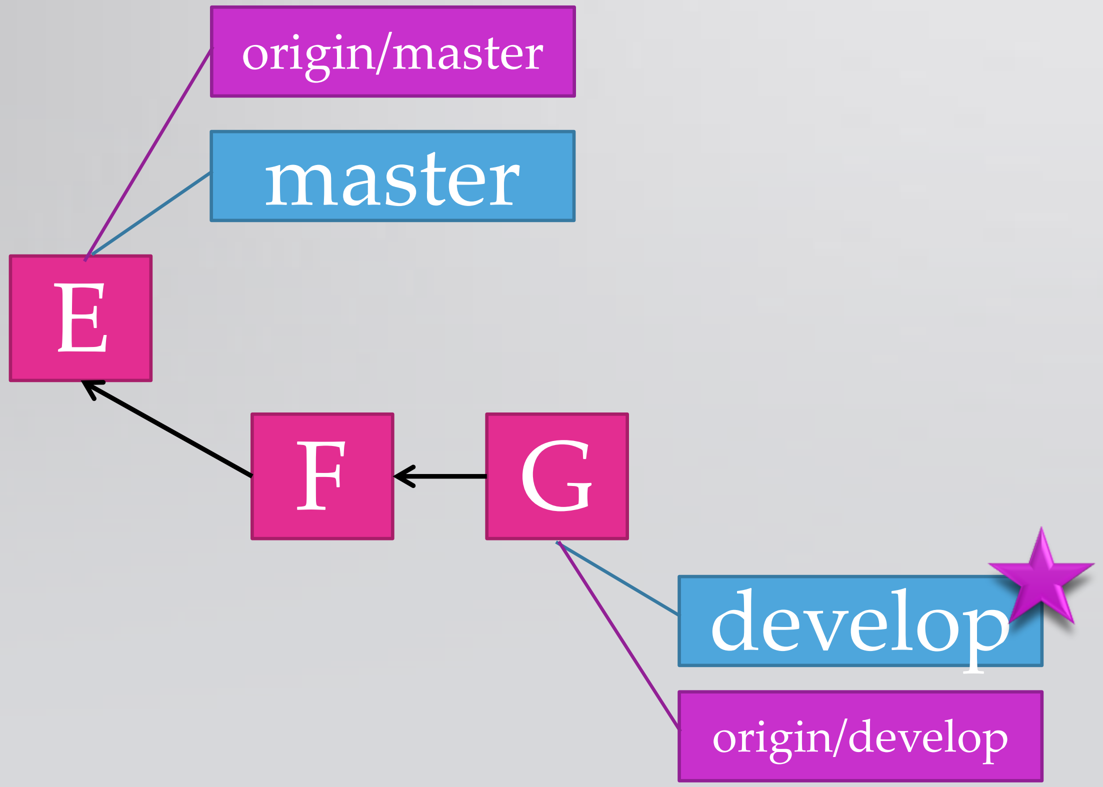


```
> git checkout develop
```

Branches Illustrated

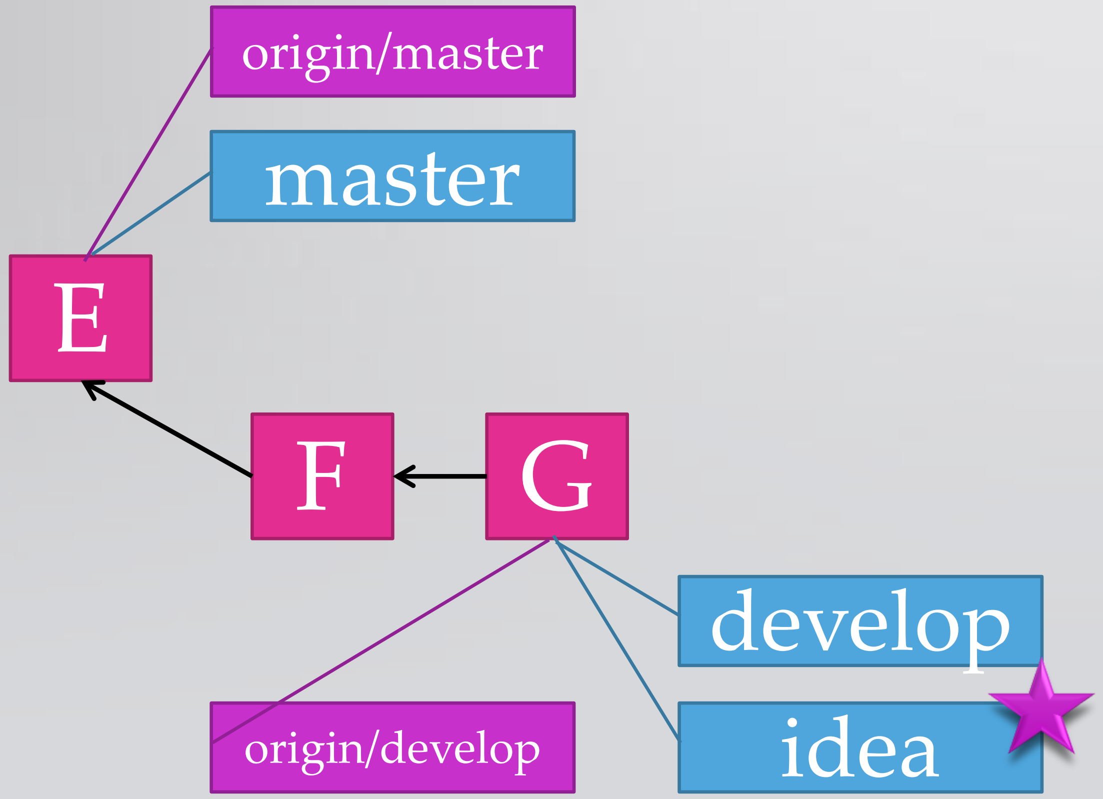


Branches Illustrated



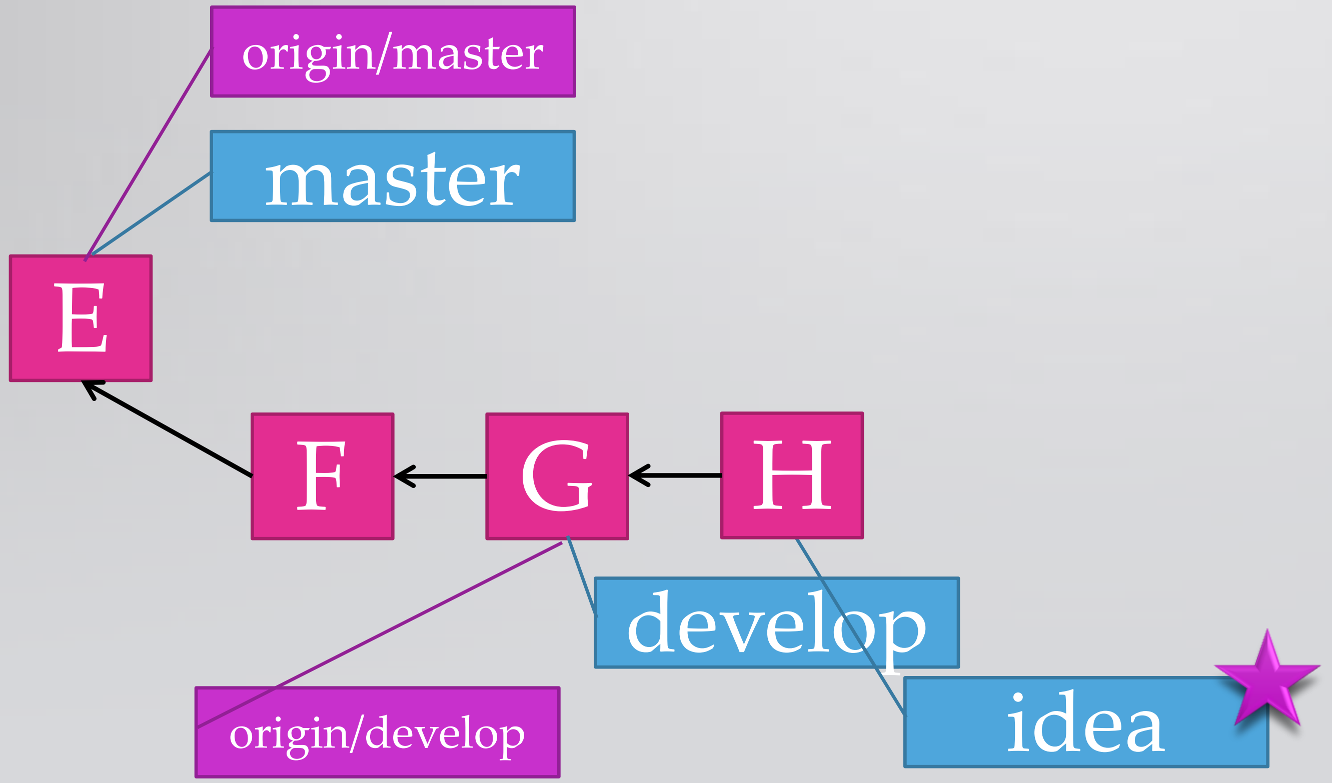
```
> git pull origin develop
```

Branches Illustrated



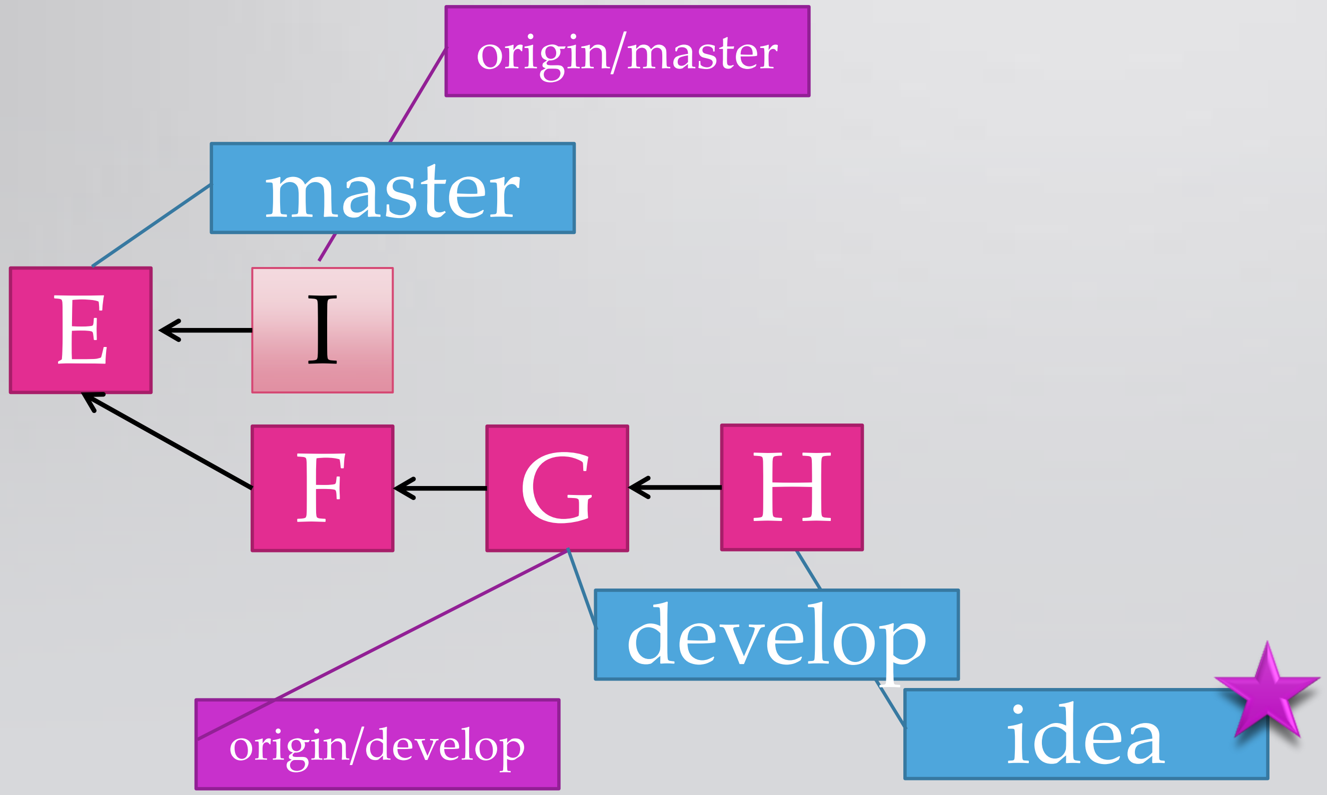
```
> git checkout -b idea
```

Branches Illustrated

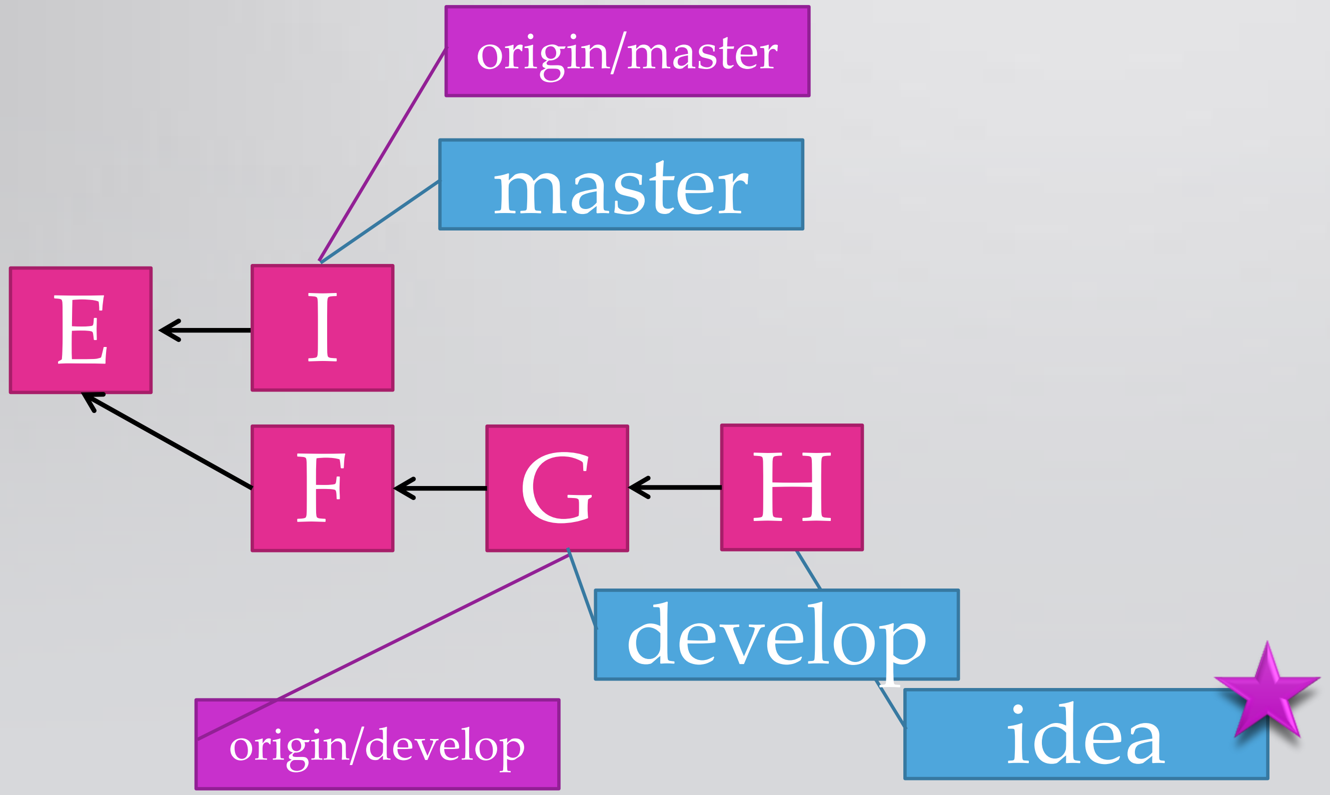


```
> git commit
```

Branches Illustrated

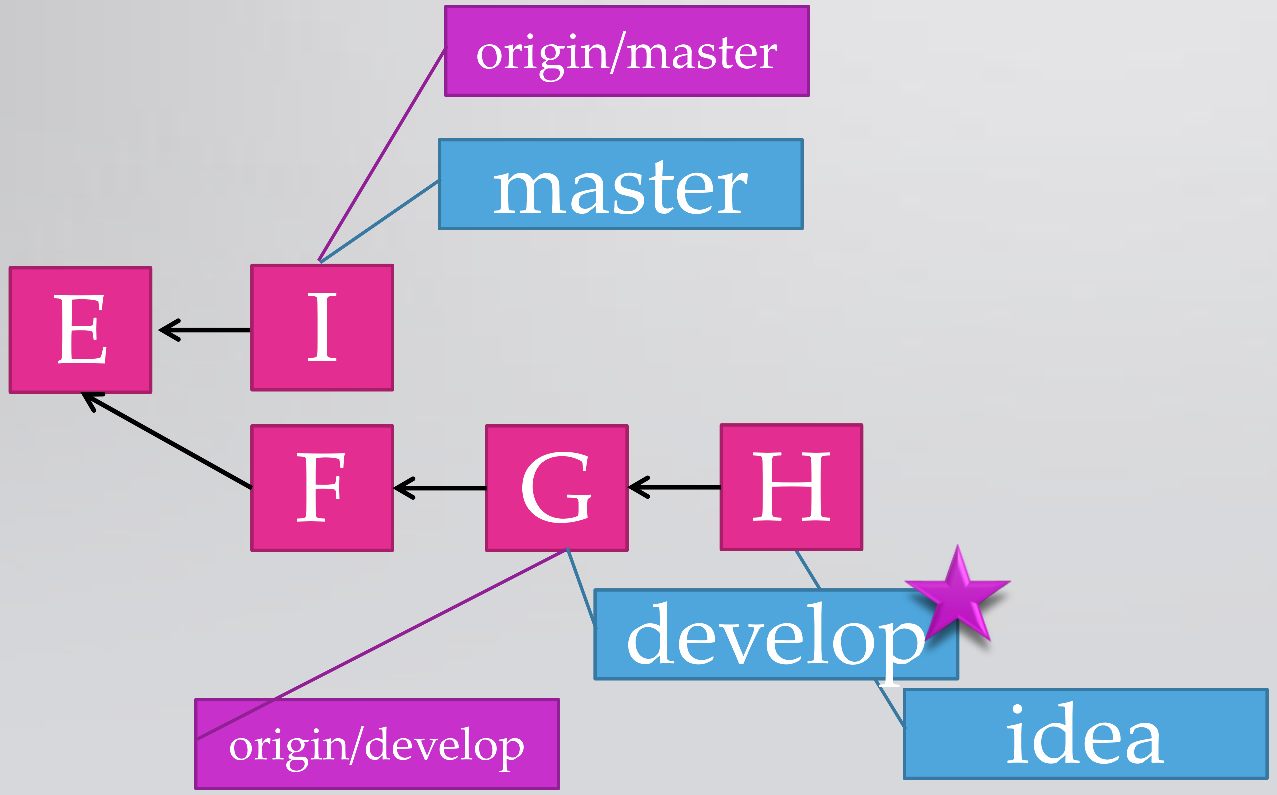


Branches Illustrated



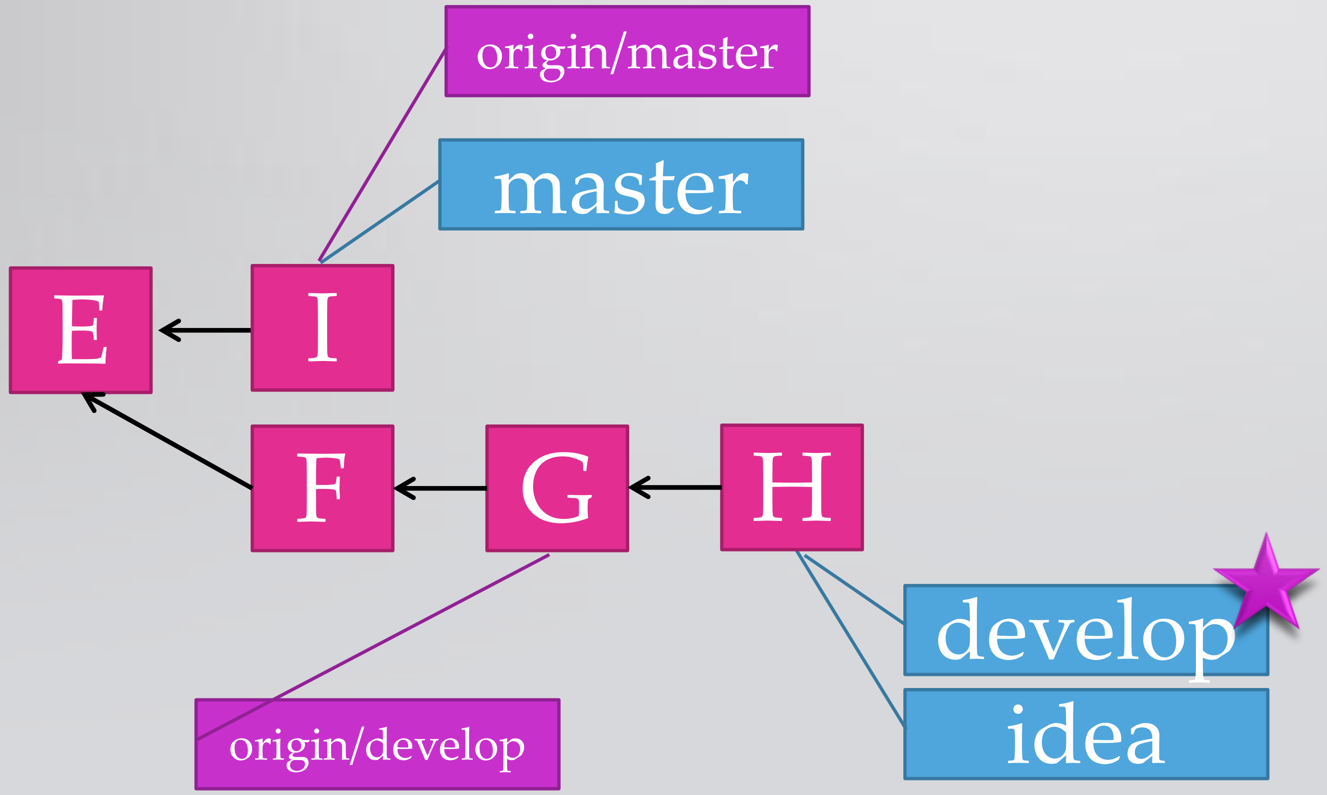
```
> git pull (at least daily)
```

Branches Illustrated



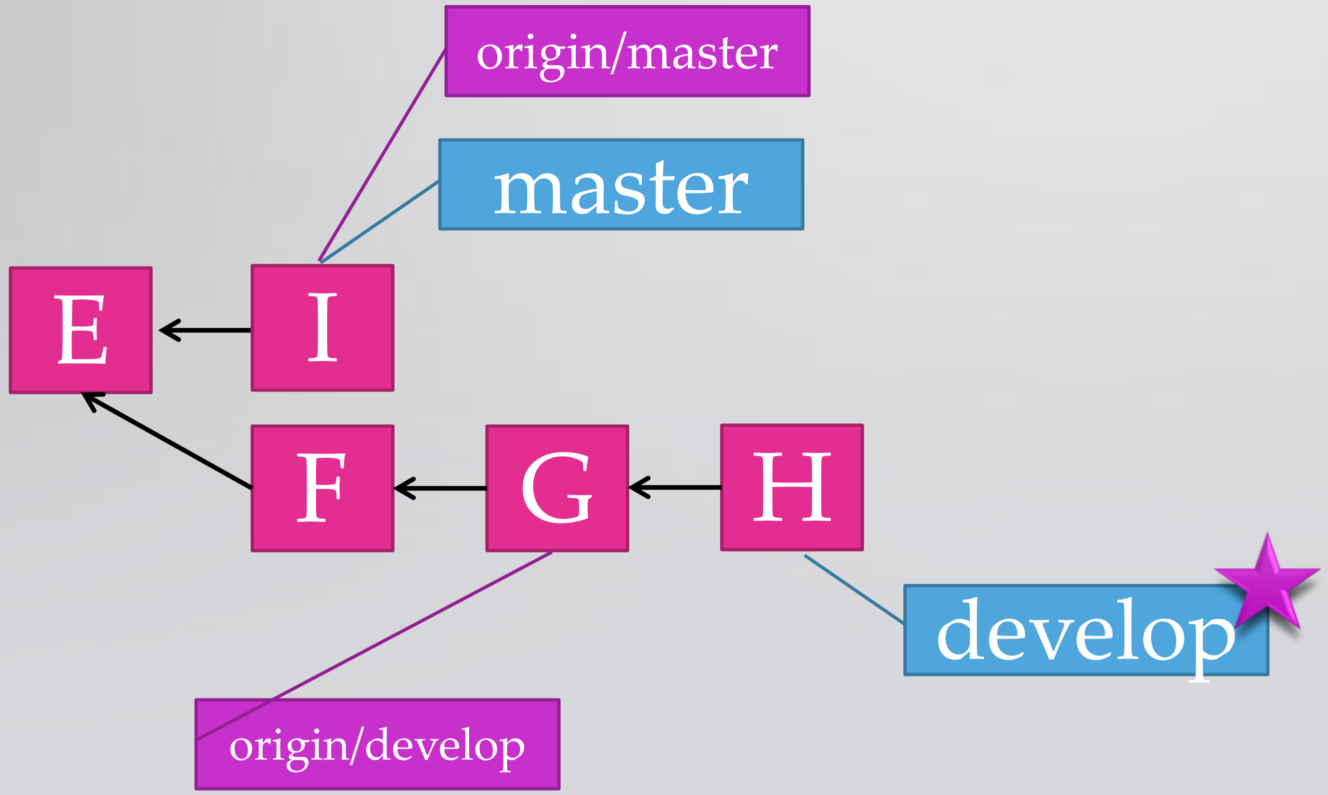
```
> git checkout develop
```

Branches Illustrated



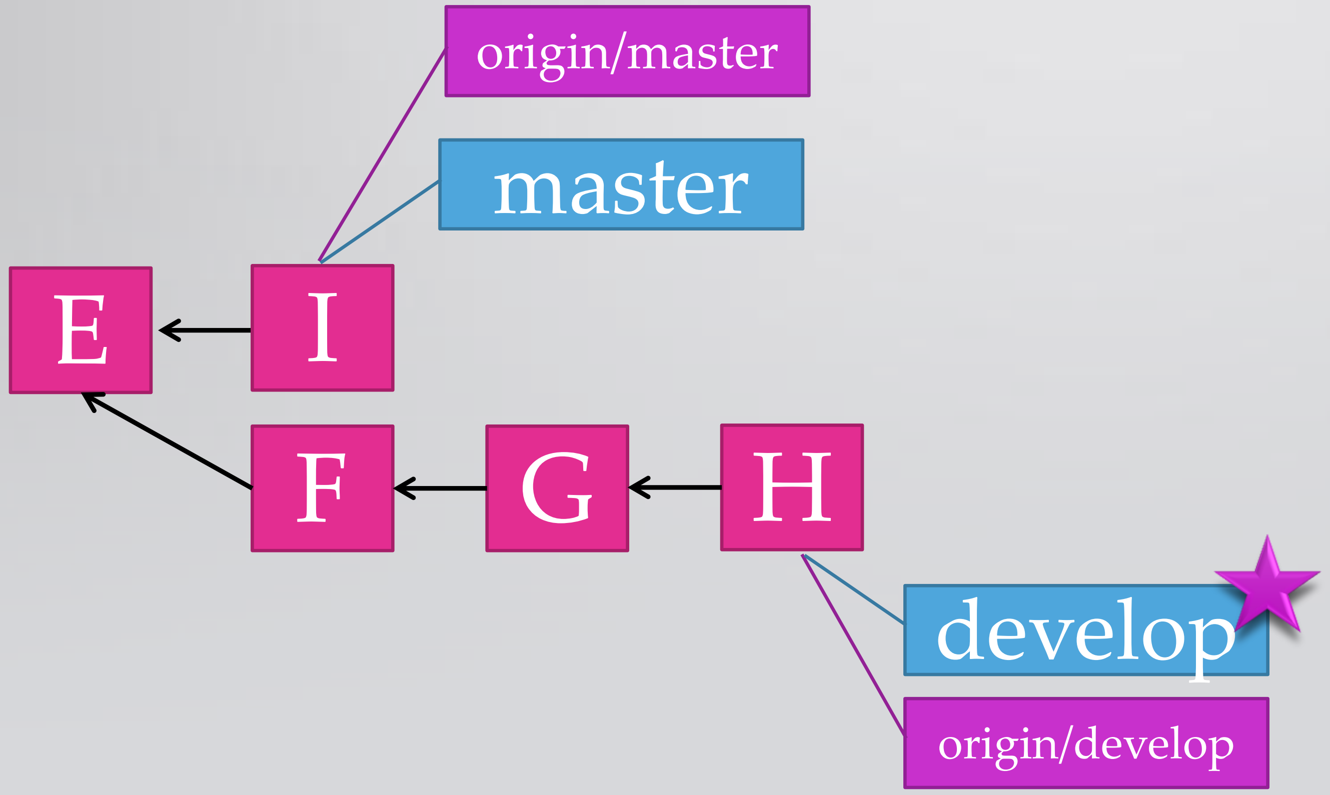
```
> git merge idea (fast forward merge)
```

Branches Illustrated



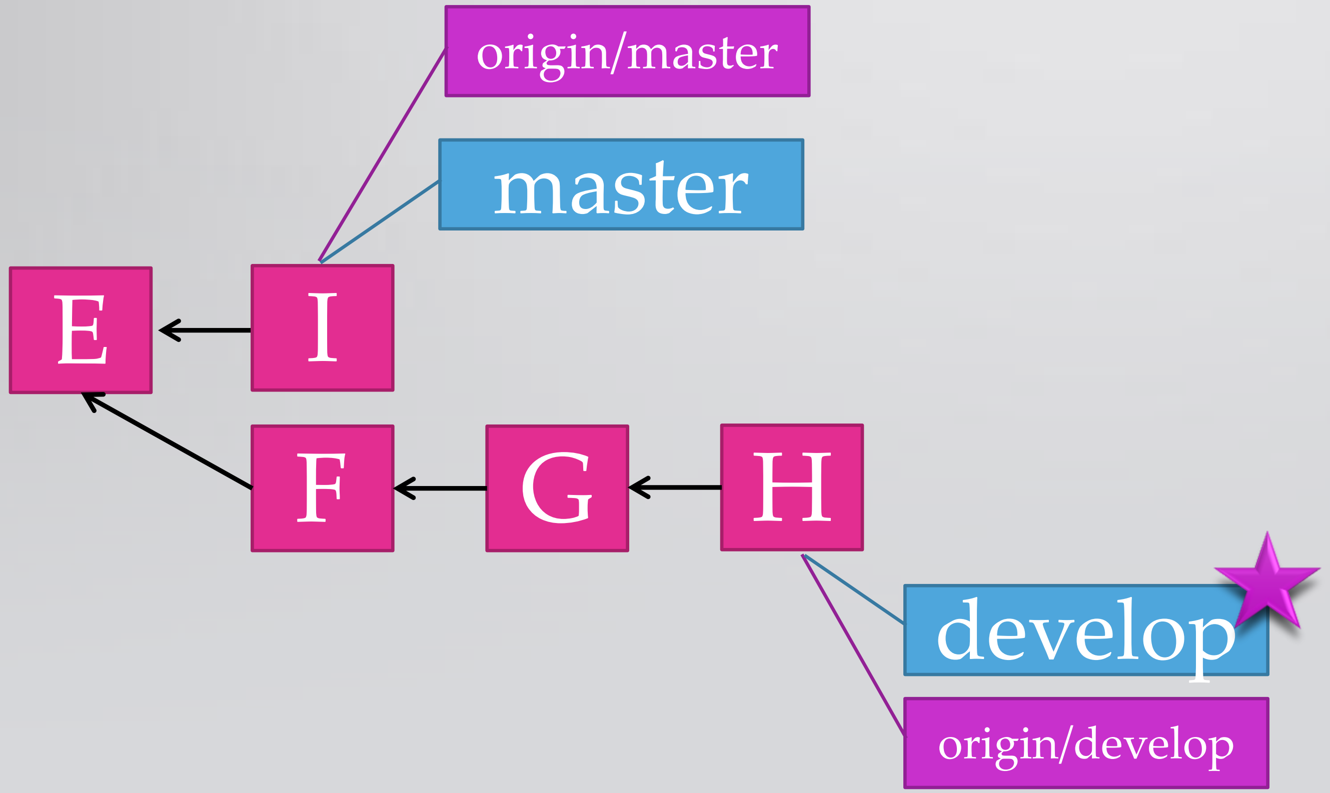
```
> git branch -d idea
```

Branches Illustrated



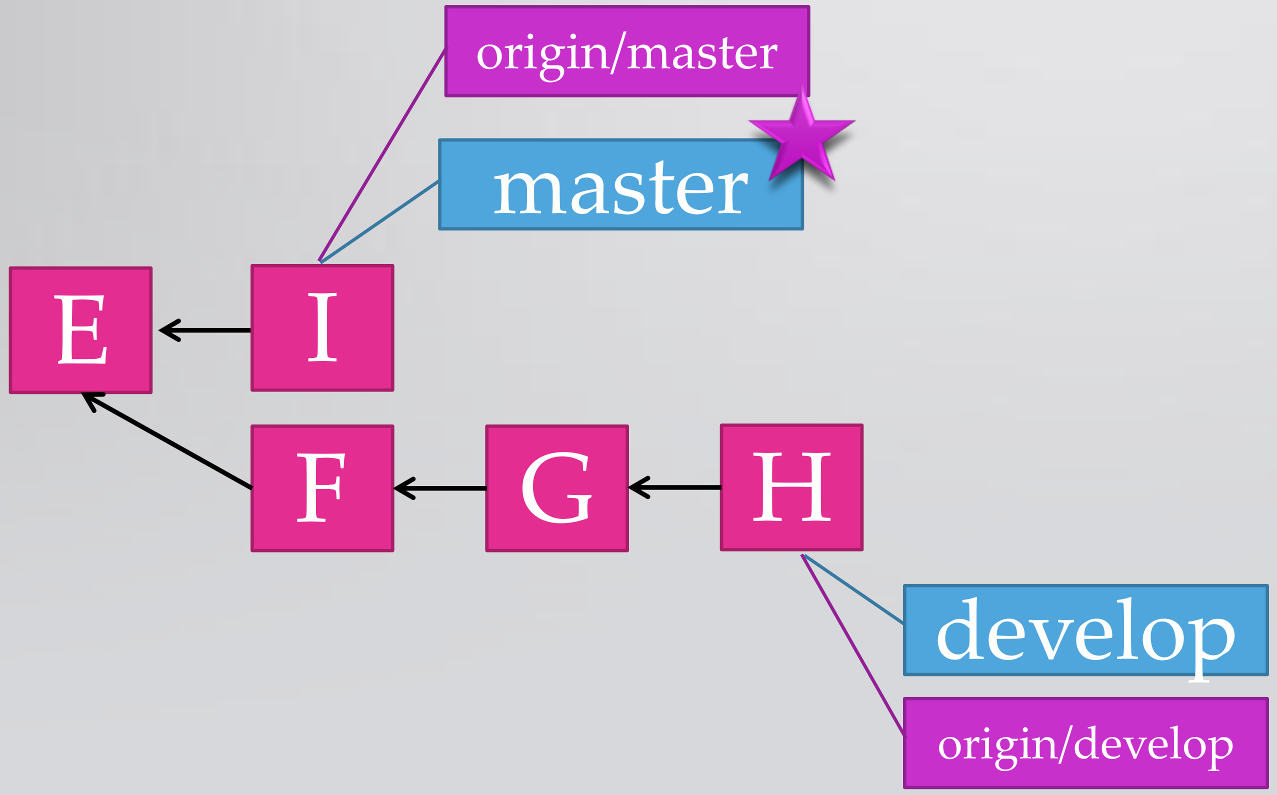
```
> git push origin develop
```

Merge Flow vs. Rebase Flow



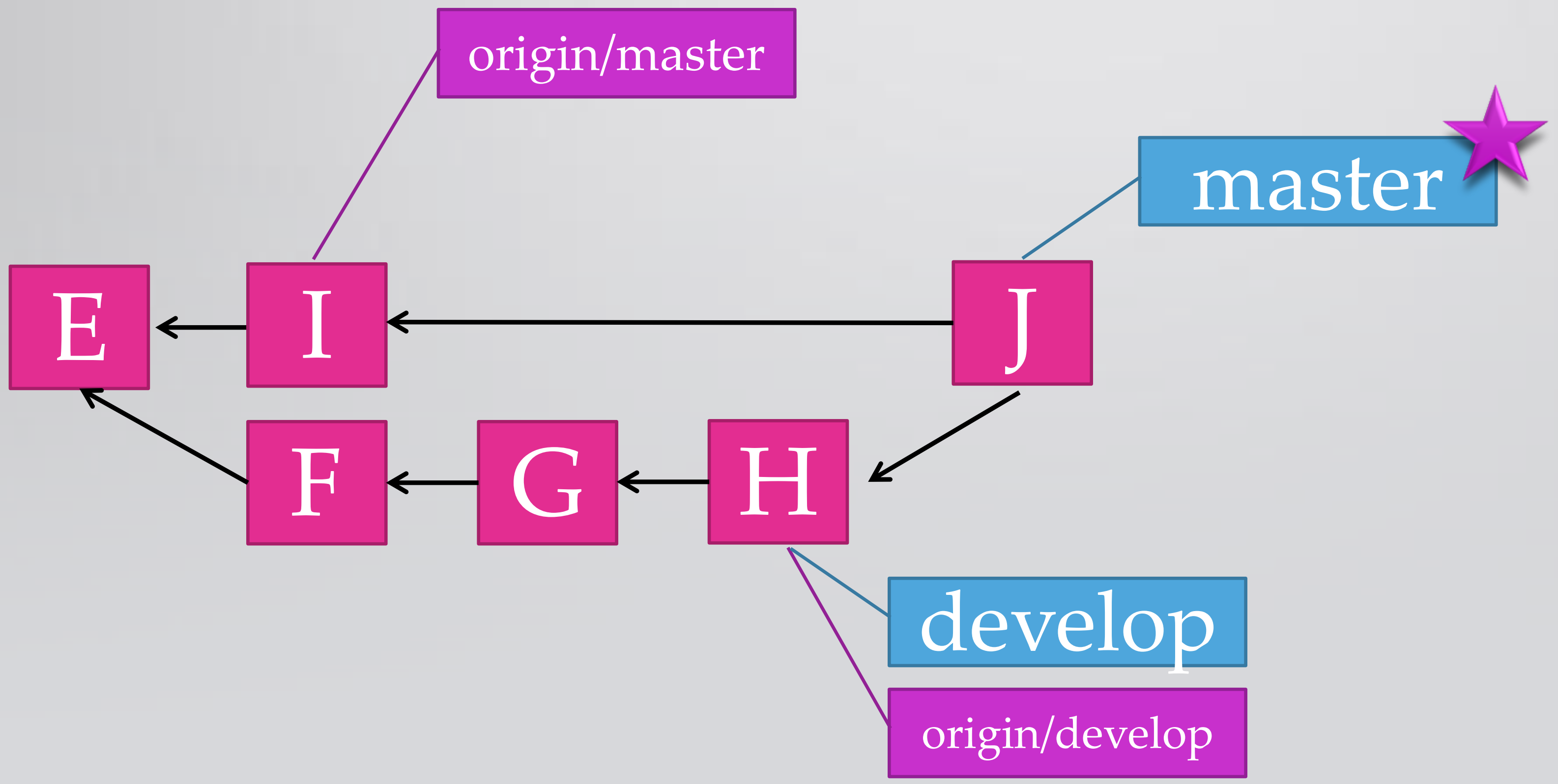
```
> git push origin develop
```

Branches Illustrated – Merge Flow



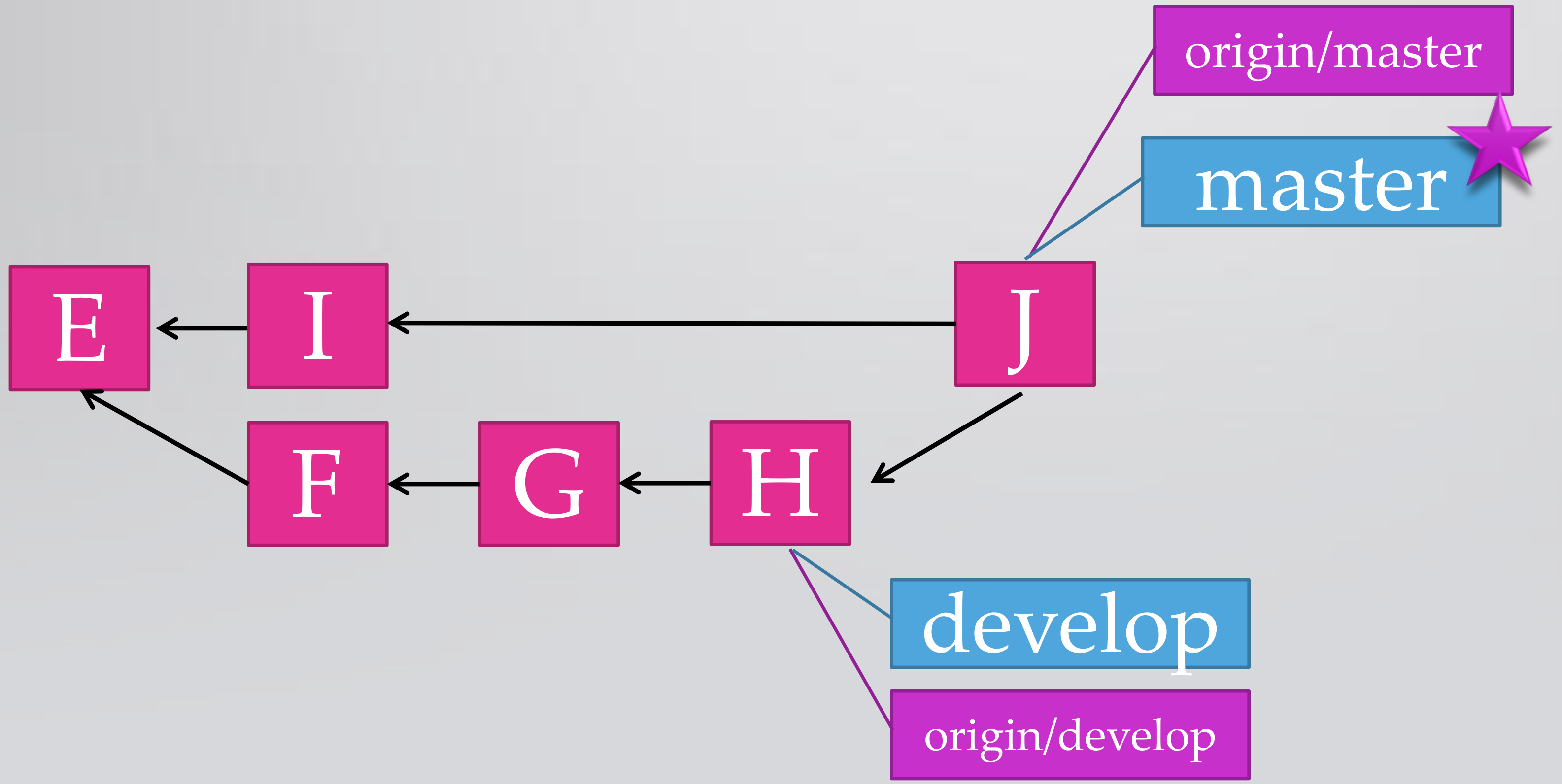
```
> git checkout master
```

Branches Illustrated – Merge Flow



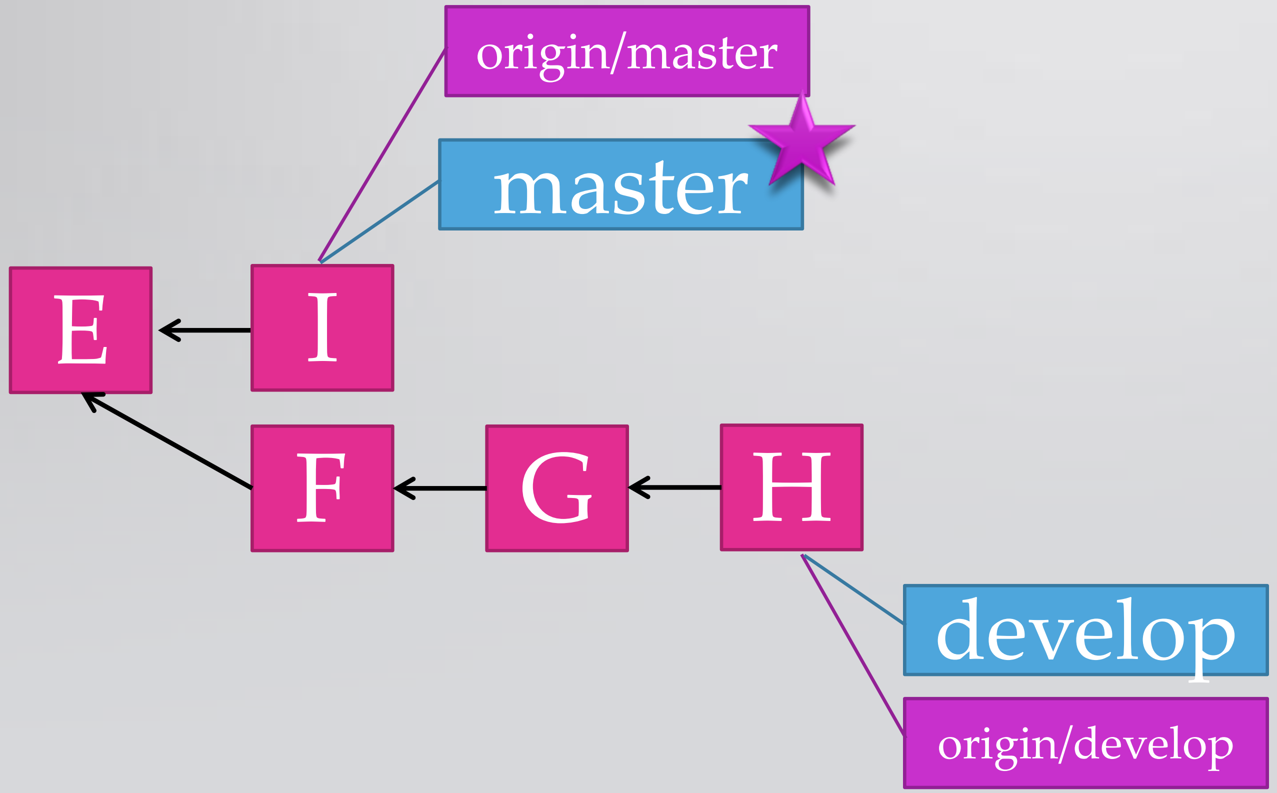
```
> git merge develop
```

Branches Illustrated – Merge Flow



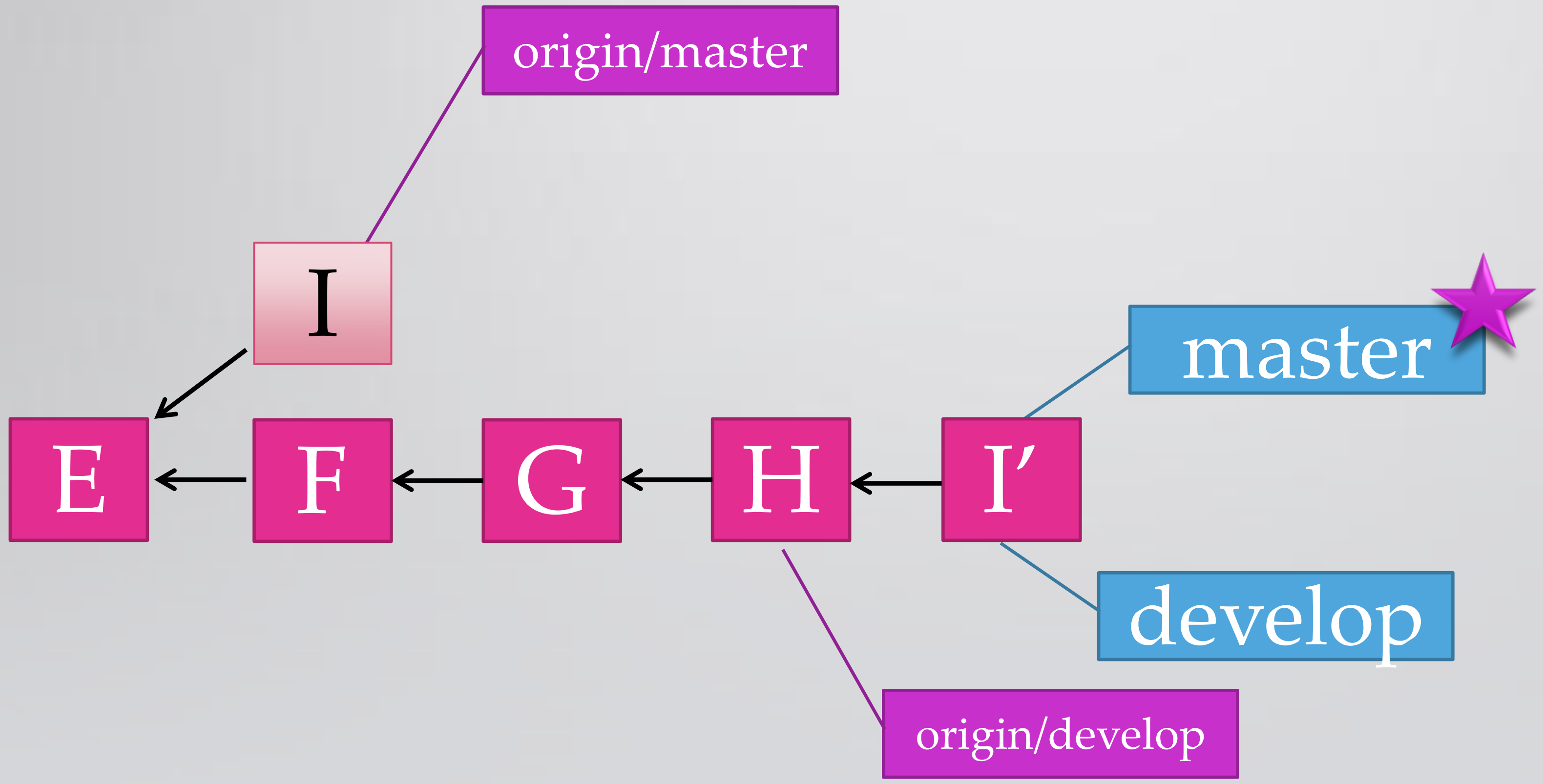
```
> git push origin
```

Branches Illustrated – Rebase Flow



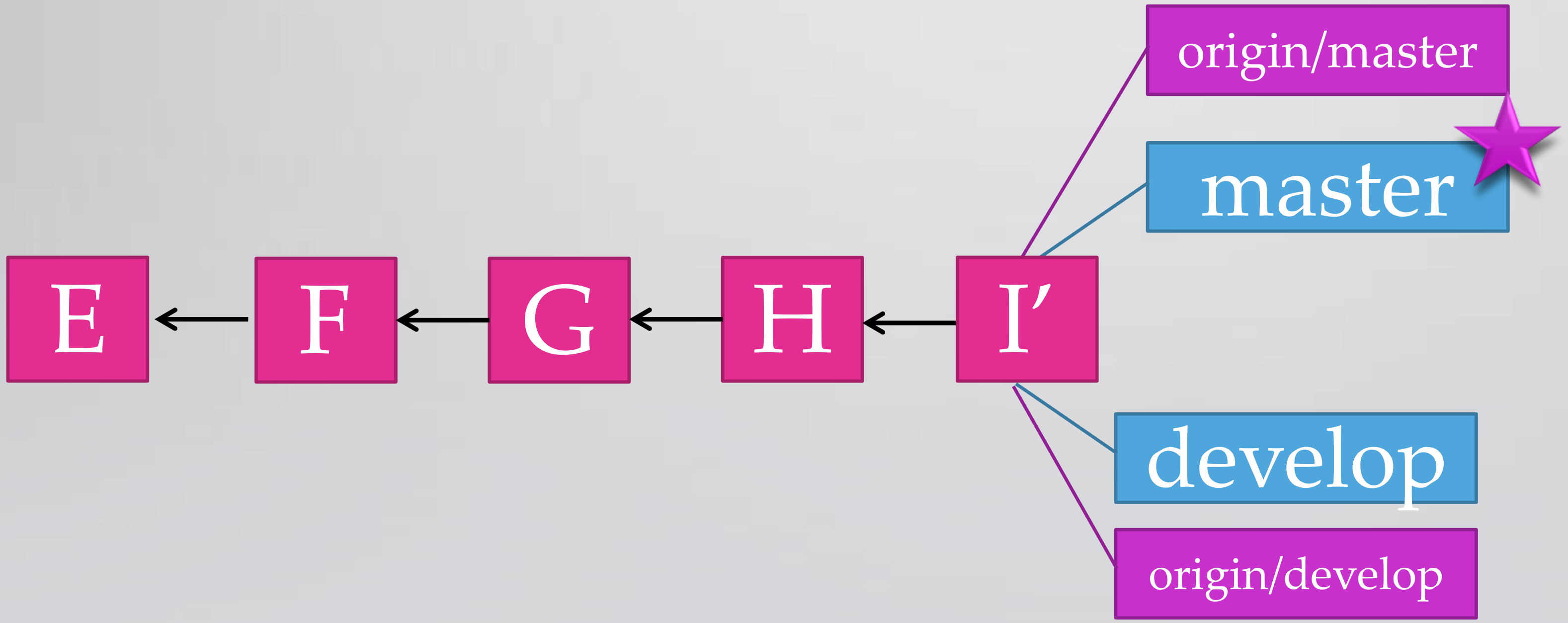
```
> git checkout master
```

Branches Illustrated – Rebase Flow



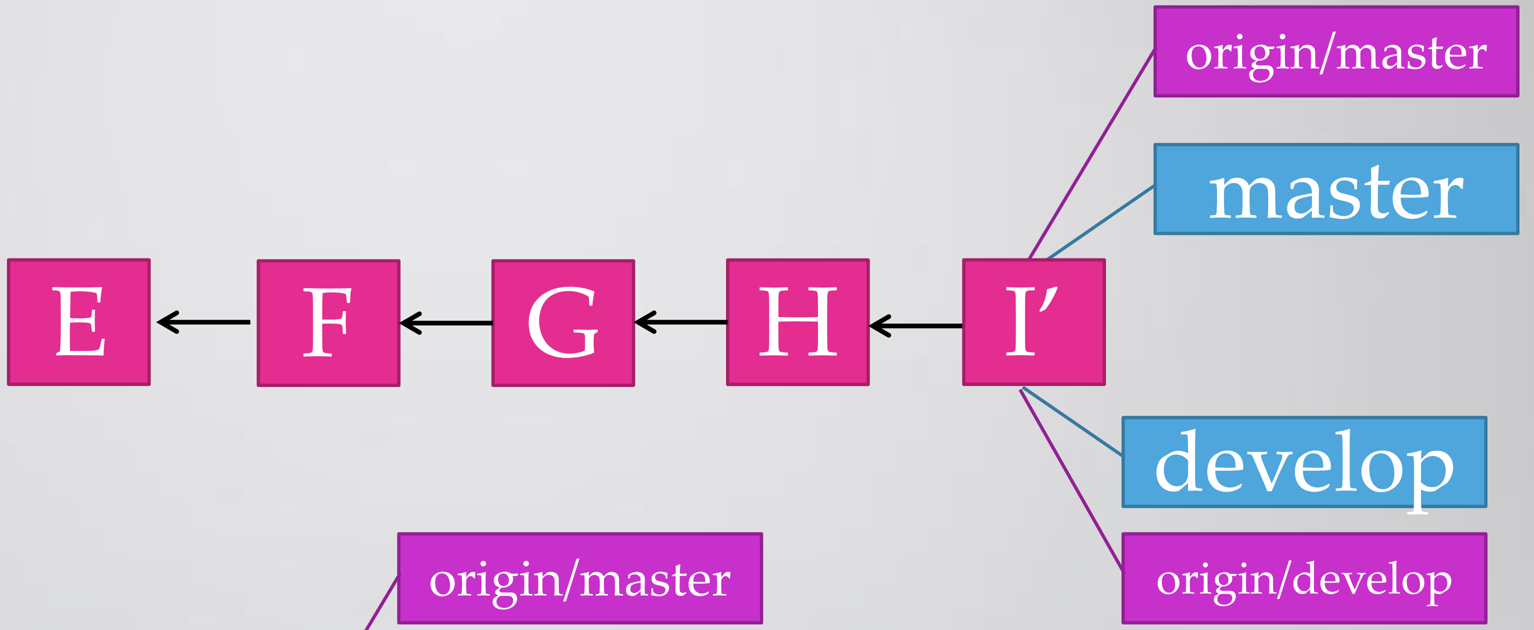
```
> git rebase develop
```

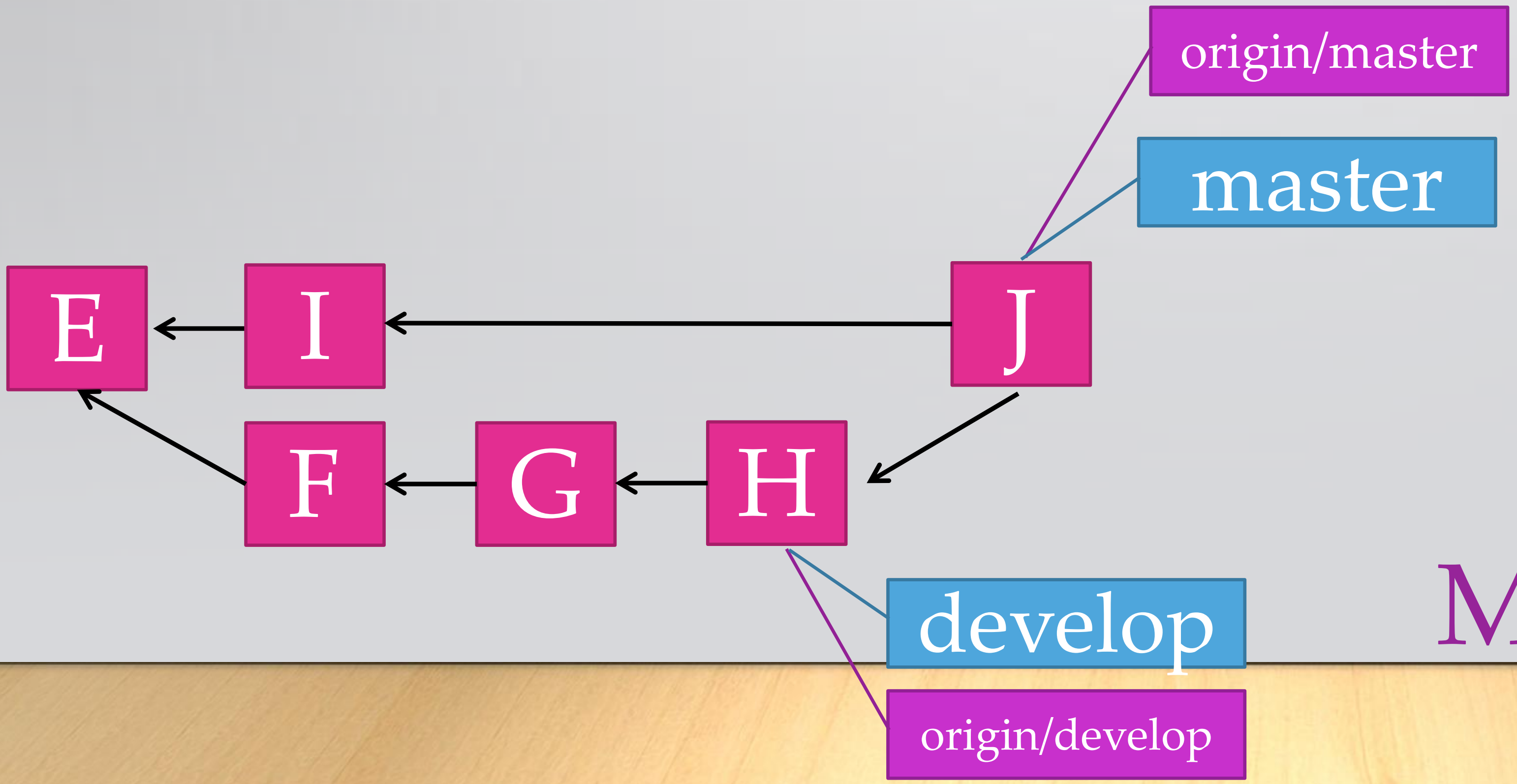
Branches Illustrated – Rebase Flow



```
> git push origin
```

Rebase Flow





Merge Flow

Keeping it simple

- **If you:**
  - **Make sure you are current with the central repository**
  - **Make some improvements to your code**
  - **Update the central repository before anyone else does**
- **Then you don’t have to worry about resolving conflicts or working with multiple branches**
  - All the complexity in git comes from dealing with these
- **Therefore:**
  - **Make sure you are up-to-date before starting to work**
  - **Commit and update the central repository frequently**