Before we start talking about Git again, here are some tips to make you more productive with your final projects.

## Git Command Line Hacks

### *Git Aliases*

I assume you've been using Git for a while. What are your most common git commands? Let's find out.

```
$ history | grep git
```

If you've been using Git as much as you should, you've probably been using it a lot. What's your most common Git command? Mine is `git status`.

Typing out `git status` over-and-over-again can get tiring. Let's create a shortcut.

Let's create *aliases* in your *Terminal profile* (aka your "*dotfile*") for whatever Git commands you type most frequently. Here are some examples.

~/.profile

```
… some other lines of Bash code up here …
alias gst="git status"
alias ga="git add"
alias gc="git commit"
alias gcm="git commit -m"
alias gphm="git push heroku master"
```

You may have some other lines in this file as well. Don't worry about those (and don't delete them!).

Also the line order doesn't matter either, so you can write them in any order. **But don't put spaces around the = sign!**

YES: `gst="git status"`
NO: `gst = "git status"`

You can get away with extra spaces in Ruby, but not in Bash.

*What's a Terminal profile (aka. dotfile)?*

A dotfile, generically speaking, is a configuration file for a program in Linux. Lots of programs you have installed on your computer have dotfiles. And they're typically stored in your home `$HOME` directory.

Terminal

```
$ ls -al ~
```

If you don't have many dotfiles, that's ok. You should at least have one .profile file. (If you have a .bash_profile or .zsh_profile, let us know.)

A dotfile contains commands for configuring a program as it launches. Think of it as a Rails controller `before_action`, but for programs. This technique is used a lot in Linux-based file systems.

The .profile file, for example, configures your Terminal session. It contains lines of Bash code that you would like to execute every time you open a new Terminal window (or even a new Terminal tab).

## So what kinds of things can you do in these dotfiles?

It depends on the program you're configuring. Typically, you'll set or update Bash variables.

Remember those? Remind yourself how they work.

```
$ export HELLO="world"
$ echo $HELLO
```

These variables become part of the *environment* (`ENV`) and can be read by any running program. Think of the environment as a blackboard. Any program can write to it, and any other program can read from it.

## What are other examples of dotfiles?

Dotfiles are typically named after the programs they configure. Can you guess what these two files allow you to customize?

~/.gemrc
~/.irbrc

The .gemrc file allows you to customize the way the `gem` command works. For example, to make installing RubyGems faster, you can configure the `gem` command to skip downloading documentation for every RubyGem you install (who needs documentation when you've got Google?).

~/.gemrc
```
gem: -N
```

We'll revisit the other one (~/.irbrc) later for another tip (in the *Command Line 2* lecture).

### *Other Terminal Aliases*

The easiest way to optimize your productivity with Git (and the Command Line in general) is by *optimizing your keystrokes*.

The fewer keystrokes it takes to accomplish the same task, the more productive you are (and the less likely you are to make mistakes).

In fact, that's a common theme in advanced web tools: HAML, Slim, CoffeeScript, SASS, some parts of ES6… they all use fewer keystrokes for getting the the same job done.

`nil` is better than `null`
*~Ruby (if Ruby could talk)*

In the Terminal, the easiest way to reduce keystrokes is to create `aliases` for your most common commands. I've shown you how to do that with some Git commands above, but you're welcome to use it for whichever commands you use most frequently. Use this to generate an ordered list of your most common commands.

```
$ history | cut -c 8- | sort | uniq -c | sort
```

The `git status` command is usually #1 for me. My new `gst` alias is going to save me a bunch of time.

In this lecture, as a introduce new Git commands, I'm going to define new aliases for them. That should allow us to move more quickly.

Note that every time you edit your ~/.profile, you should start a new Terminal session to see the changes (either by restarting the app or simply opening a new tab). You can also `source ~/.profile`, but that can sometimes produce confusing results (i.e. an extra-long `$PATH`).

### Hacking the Prompt

Before we move on, let's hack the command line some more.

https://github.com/magicmonty/bash-git-prompt

This repo contains the source code for a Bash prompt customization that will give us more information about our current Git repo. Installation instructions are on the page.

In the Mac terminal

```
brew update
brew install bash-git-prompt
```

Add to Mac ~/.profile

```
if [ -f "$(brew
--prefix)/opt/bash-git-prompt/share/gitprompt.sh" ]; then
  __GIT_PROMPT_DIR=$(brew --prefix)/opt/bash-git-prompt/share
  source "$(brew
--prefix)/opt/bash-git-prompt/share/gitprompt.sh"
fi
```

In the Linux terminal

```
cd ~
git clone https://github.com/magicmonty/bash-git-prompt.git
.bash-git-prompt --depth=1
```

Add to Linux ~/.profile

```
GIT_PROMPT_ONLY_IN_REPO=1
source ~/.bash-git-prompt/gitprompt.sh
```

Start a new Terminal session to see the result. Do you notice a difference?

The first thing you'll notice is that it changes the style of your Terminal prompt. That's just a bonus.

What we really care about is that, when you're in a Git repo folder, this magic code will change your Terminal prompt to display your current Git *branch* (usually `master`). When you're not in a Git-initialized folder, it won't do that. Since we'll be talking about Git branches today, this Terminal hack will be useful for us to quickly see where we are in the Git tree.

### SDLC

Let's talk about the *software development life cycle* (SDLC). You'll commonly see references in SDLC in job postings. At least one of our hiring partners will expect you to understand this.

You can read the Wikipedia article about it: http://en.wikipedia.org/wiki/Systems_development_life_cycle. Unfortunately, in typical Wikipedia style, this article is wonky enterprise overkill. We're going to keep it simple.

This is how you develop software.

- Step 1: Come up with a **product specification**. What are you building? What features should it have? Agile user stories should be written at this step. There should be some lean validation here as well.
- Step 2: **Design** the product. This includes both visual design and system design. At this step you're drawing visual mockups and whiteboarding boxes and arrows for your data models. There should be some lean validation at this step as well.
- Step 3: Write **code**. The code you write should be deployed and tested in a "development" environment, typically your laptop.
- Step 4: **Integration**. Your code is integrated with your teammates' code in a "staging" environment. The team tests the staging site to make sure everything is working correctly with everyone's contributions.
- Step 5: **Release**. Once the testing is complete and the team is confident things are working the way they want, they release the code to the "production" environment where real customers can use it.

We've covered steps 1-3 pretty well. We haven't really talked about steps 4 and 5 much.

When you think of *integration*, you should think of `git pull`. Git forces you to pull your coworkers code before you can push your new code. The point of this requirement is to force you to *continuously integrate* your coworkers code with your own. After you pull, you should check if the new code breaks any of your new features in your "dev" environment. You should check for merge conflicts and run some quick tests to make sure everything is still working ok.

By pushing code to GitHub, you're adding your code to the pile of team code. When you deploy the code in GitHub, you should first deploy it to a staging environment to ensure that everyone's code actually plays well together.

### *What's a "staging" environment?*

Rails, by default, has *dev*, test, and *prod* environments. Let's review:

- The **development** environment is where you code.
- The **production** environment is where the application runs and your customers access it.
- The **test** environment is where you run your tests.

What about "staging"? Rails doesn't have a "staging" environment configuration by default. We'll explain why in a second. In a nutshell, a staging environments sits between dev and prod.

One way to think about the different environments is by thinking of the type of data they contain.

A dev environment typically has a few rows of data - just enough for you to be able to confirm that whatever you're working on is working correctly.

A prod environment has all the real customer data. That could be hundreds, thousands, even millions of rows.

The test environment could contains tons of data, but the data is usually fake and meaningless. The users might all look like user1@email.com living at 123 Main St,

Anywhere, USA. A test environment is not meant for humans. It's a place for fully-automated machines to check that things are working correctly.

The staging env contains a medium amount of realistic data. It's a place for real people to play with your app before it's released to production. Think of it as a demo site for all the non-technical people in the company to see what you're building.

In some companies, staging can also be a platform for manual quality assurance (QA) testing. Every day some poor shmucks may have to run through a 100-item script to test everything in your app. If they find a bug, they'll report it to the devs, who'll then attempt to reproduce and fix the bug in the dev env. When that's done, they'll push it to staging to be tested again.

Git is commonly used to manage the SDLC, particularly the last 3 steps.

Step 3: In your dev env, you're probably making commits and push/pulling from GitHub regularly.
Step 4: You'll use git to deploy your code to staging. The staging env is tested to make sure that everyone's commits integrate well together.
Step 5: After staging is verified, you'll use git once again to deploy to a prod env.

Typically each environment (dev, staging, and prod) is associated with a Git branch. To "deploy" code means to deploy a branch of code to a service (i.e. Heroku).

Why doesn't Rails have a staging env?

Since staging is a staging area for production, the staging env should be running in production mode. So, by default, Rails doesn't need a staging env configuration. But some companies have one anyway for various reasons.

### The Git Tree

Let's start a new git-backed app. Starting at your projects dir:

```
$ mkdir git2
$ cd git2
$ git init
```

We're not going to worry about Ruby or Rails today. Let's just create some blank files to track.

```
$ touch file{1,2,3}.rb
```

Don't forget to use your new git aliases. Feel the power.

```
$ gst
On branch master

Initial commit

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

        file1.rb
        file2.rb
        file3.rb

nothing added to commit but untracked files present (use "git add" to track)
```

This looks fine. Let's add these files.

```
$ ga .
$ gst
On branch master

Initial commit

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

        new file:   file1.rb
        new file:   file2.rb
        new file:   file3.rb
```

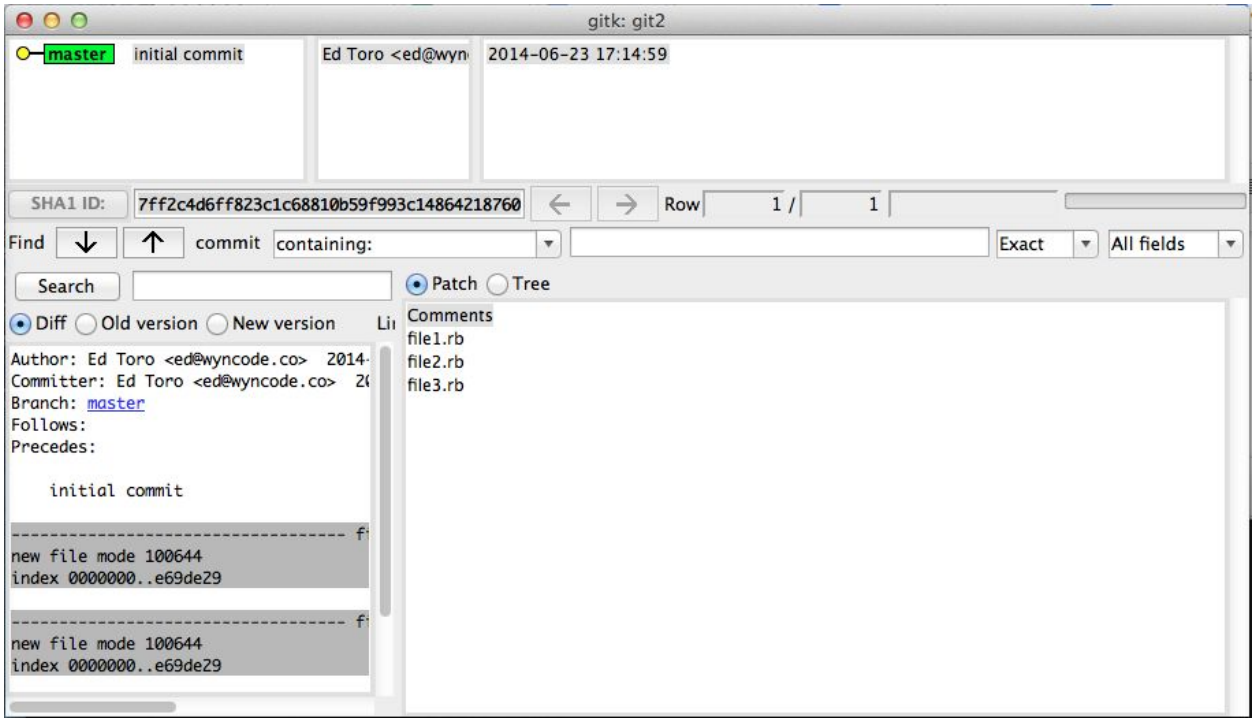
Now that we have `gst`, we should be checking the git status constantly. It's just so easy now.

This looks good. Let's commit.

```
$ gc -m "initial commit"
[master (root-commit) 7ff2c4d] initial commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.rb
 create mode 100644 file2.rb
 create mode 100644 file3.rb
```

Before we create a branch, I want to show you something new.

```
$ gitk --all
```




`gitk` is a git repo browser. It comes for free when you install Git. It's a GUI that allows you to examine your git repo tree.

If this is the first time you're seeing this app, use the menu to update your preferences to fix the fonts to your liking. Everything starts out really small by default.

Your commits are listed at the top. Your git tree visualization is in the upper left. The bottom section displays the details of the selected commit.

We're going to spend most of our time today focusing on the section in the upper-left. Try to make it big. (The window size adjustment is tricky.)

It's important to understand that the structure of a git repo can be represented as a tree. Whenever you work with complex git commands, it's going to be useful to visualize the tree. And `gitk` helps with that. Anything you can imagine doing to a tree, you can do with git.

The first commit plants the seed. That's not very interesting. Let's grow this tree. Quit the repo browser (with the menu Wish->Quit or Command+Q) and return to the Terminal. (The Terminal won't work until you fully quit `gitk`.)

```
$ touch file{4,5,6}.rb
$ ga .
$ gc -m "2nd commit"
$ gitk --all
```

Quit that. I'm bored of typing `gitk --all`, so let's add an alias.

~/.profile

```
...
alias gka="gitk --all &"
...
```

That ampersand opens up gitk as a background task so you can continue to use your terminal.

Don't forget to open a new Terminal window to activate the new profile.

```
$ gka
```

There, that's better. Now we're using the Terminal like pros.


So my git tree has two nodes in it right now.

Tag and Release

This app looks good enough to release. Technically there's no difference between "staging" and "production" at this stage. We don't have any customers yet who can be scared away with bugs. So let's release directly to "production".

To prepare for the release, let's create a tag. What's a tag? It's exactly what it sounds like. It's a tag attached to a portion of a tree. We'll see what that looks like in a moment.

What do we name our tag? Tags are typically named for versions of your software. You can use whatever naming convention you want: movie characters, Greek letters, numbers. I recommend using "semantic versioning" (http://semver.org/).

## Summary

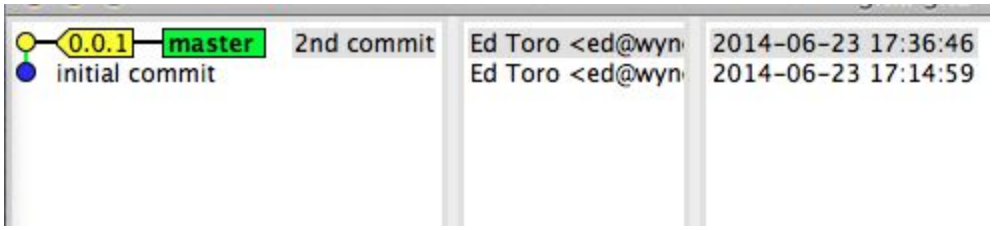
Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

We'll call the first pre-release-alpha-private version of our software 0.0.1, so we'll use that as our tag.

```
$ git tag "0.0.1"
$ gka
```




The tag appears in our git tree. And it literally looks like a tag. That's convenient.

To release this code, we need to push this repo somewhere. Usually we'd push it to Heroku (or some hosting service), but today we're going to push it to GitHub and pretend that it's Heroku. You typically wouldn't do this, but GitHub will give us more information about what our git repo looks like on the other end of the deployment.

Create a repo on GitHub and follow *half* of the instructions on pushing an existing repo. Just run this line.

```
$ git remote add origin
https://github.com/eddroid/wyncode_git.git
```

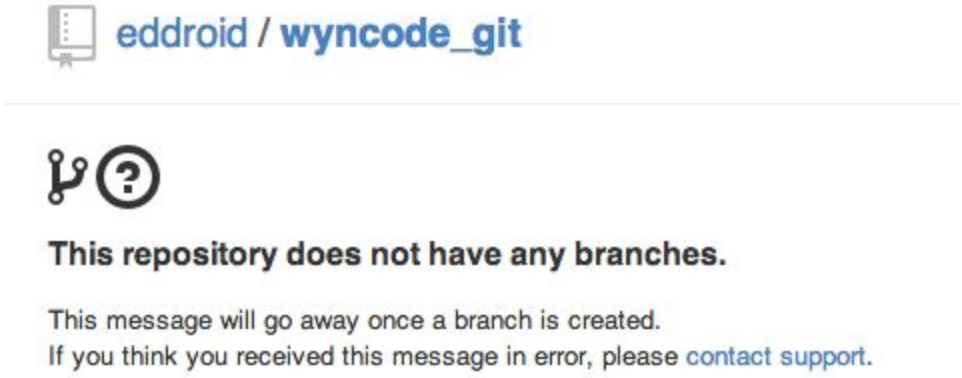
Your GitHub repo URL will be different.

Don't execute the command about pushing "master". We don't want to deploy the "master" branch. We want to deploy our newly tagged release. This is a "tag and release" deployment strategy.

```
$ git push origin "0.0.1"
```

This is much more readable than pushing `master`. Where are we pushing? To `origin` (which is what GitHub calls itself). What are we pushing? Version `0.0.1` of our app.

If you refresh the repo on GitHub, it doesn't like what we just did, but I don't care.

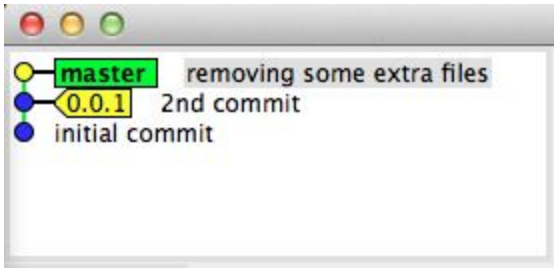



We're not testing deployments to GitHub. We're testing deployments to Heroku. And Heroku won't complain if I deploy a tag. It'll just happily deploy the code at that tag.

Patch Deployments

Let's start working on version 0.0.2 of our app. Let's strip out some features.

```
$ git rm file{4,5,6}.rb
$ gc -m "start feature: removing some extra files"
$ gka
```




`gitk` shows me that I've now moved beyond my tag and grown my git tree by one more commit. The tag stays attached to the commit in the tree where it was created. The tree is free to keep growing past it.

Let's pretend there's a bug in production. One of the files I deleted (`file4.rb`) is causing trouble. But I've already started working on replacing that file. And my feature isn't done yet!

I don’t want to tag-and-release from master (the top of my tree) because I’m not done with it yet. It’s not ready. I deleted `file4.rb`, but I need to do some other things as well.

How do I fix a bug in a file that doesn’t exist anymore?

I need to go back in time and fix `file4.rb`. Do you remember how to do that?

To go back in time, we need to `git checkout` to a previous commit. But a commit id hash is a long, ugly string. We can do better. Let’s `git checkout` our previously tagged release.

```
$ git checkout 0.0.1
Note: checking out '0.0.1'.

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 0899607... 2nd commit
```

By tagging the release, I was able to easily jump back to that point in time without having to specify a commit id.

In Git, everywhere we’ve used a commit id in the past, we can also use a tag name as well (if we’ve made one). As we’ll see shortly, branch names work as well.

Commit ids, tags, and branches are all considered *commitish*. That means anywhere in Git where you would use a commit id, you can also use commitish things like tags and branches. There are lots of commitish things in Git, but we won’t go over all of them.

For example, with a specially constructed commitish value you can “time travel” (checkout) to a point-in-time (like “yesterday” or “2 weeks ago”). You can also checkout to a position (i.e. “X commits ago”).

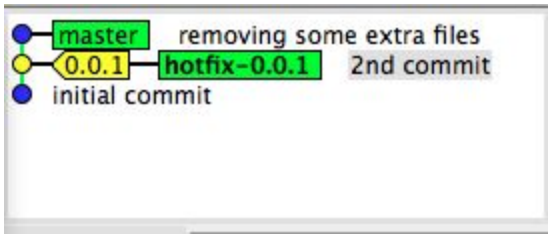
Back to work. Unfortunately, git is telling me that I’m now in a “detached HEAD” state. That’s bad. I’m in a part of my git tree that doesn’t correspond to any known *branch*. I’m in the middle of my tree. My new pimped-out Terminal prompt even updated to tell me where I am:

etoro@Eds-MacBook-Pro:~/src/git2[(**no branch**)]$

Being in a “detached HEAD” state is bad because it’s hard for git to keep track of my work. New commits should go on a branch. Commits that aren’t on a branch can be accidentally lost.

So let’s take Git’s advice and create a branch here. We need to come up with a branch name. Having to go back in time to fix an important bug in production is called a “patch” or “hotfix”.

```
$ git checkout -b hotfix-0.0.1 0.0.1
Switched to a new branch 'hotfix-0.0.1'
$ gka
```




Now my git tree has 2 branches (in green) and 1 tag (in yellow). I’m currently located on the `hotfix-0.0.1` branch (in bold). And since I’m on a branch, I’m free to start adding more commits and growing my tree in another direction.

We’ve traveled to this point in time for a reason. There’s an important file we need to update, and it exists at this point in time. By updating this file, we’ll branch the timeline and change the future.

Let's make that fix to `file4.rb` and commit.

```
$ echo "puts 'Hello World'" > file4.rb
$ gst
On branch hotfix-0.0.1
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:   file4.rb

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

I had asked in the last Git lecture that every time you see a modified file you should make a habit of checking the diff between that file and the one git knows about.

```
$ git diff file4.rb
diff --git a/file4.rb b/file4.rb
index e69de29..2f33010 100644
--- a/file4.rb
+++ b/file4.rb
@@ -0,0 +1 @@
+puts 'Hello World'
```

Let's take a quick detour.

You know what? I've never liked this `git diff` output. It's really hard to read. There must be a better way.

### *Diff Viewer Option 1: Meld*

There is an open source diff viewer called Meld we can use for this purpose.
http://meldmerge.org/

OSX

```
brew update && brew install caskroom/cask/meld
```

Ubuntu Linux

```
apt-get update && apt-get install meld
```

To launch meld to view the current diffs:
```
git difftool -t meld
```

To define a  shortcut for this diff viewer
~/.profile
```
alias gd="git difftool -t meld"
```

### *Diff Viewer Option 2: XCode FileMerge Diff Viewer*

For OSX users, XCode actually comes with a nice diff viewer. Here's some advice for setting that up: http://stackoverflow.com/questions/8155391/use-xcode-4-as-git-difftool

Let's create a file named opendiff.sh in the directory ~/bin/

```
$ mkdir ~/bin/
```

~/bin/opendiff.sh
```
#!/bin/sh
/usr/bin/opendiff "$2" "$5" -merge "$1" &
```

There are two more things we need to do to get this trick working. First we need to make this file an executable command.

```
$ chmod +x ~/bin/opendiff.sh
```

Then we need to tell git to use this command instead of the default to display diffs.

```
$ git config --global diff.external ~/bin/opendiff.sh
```

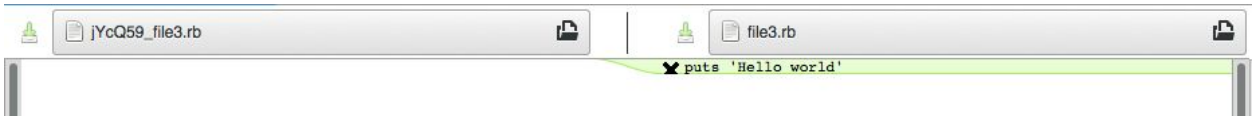
Setup an alias for this diff viewer.
~/.profile
```
alias gd="git diff"
```

Let's add some new aliases.
~/.profile

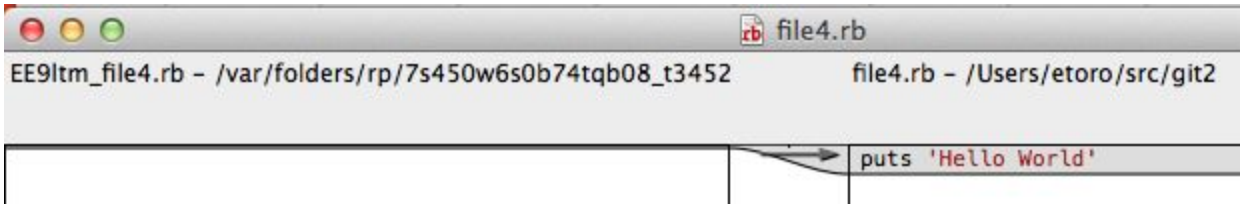
```
alias gco="git checkout"
```

Since we updated our Terminal profile, we need to open a new Terminal window to use our new aliases.

```
$ gd file4.rb
```




*Meld Diff Viewer*




*XCode FileMerge Diff Viewer*

Now that's a diff! I can see the file git knows about on the left, my file on the right, and a much clearer indication of what lines of code I've added and where.

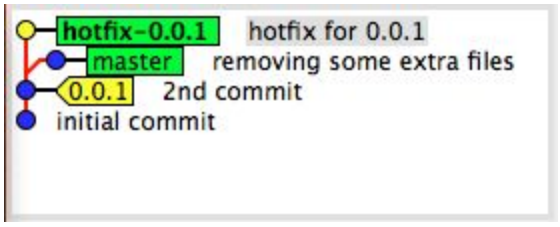
You probably haven't been using `git diff` like I asked you to. Hopefully this new tool will make it a better experience for you. If you get into the habit of `git diff`ing your files before you `git add` them, you'll catch some common bugs like:

- That time you accidentally mashed your keyboard and inserted some random text into the file.
- The logging code (`puts` and `p`) you added to debug a problem that you forgot to remove.

Whenever your `git status` indicates a modified file, get into the habit of `git diff`ing that file.

Back to work. This diff looks good. Let's add and commit this fix.

```
$ ga file4.rb
$ gc -m "hotfix for 0.0.1"
$ gka
```




Now our tree is beginning to look more like an actual tree. It has two branches, `master` and `hotfix-0.0.1`. The branches start at tag `0.0.1` and diverge. Each branch has a commit the other one doesn't.

The `master` branch isn't ready to be deployed yet because it has a half-finished feature on it. But we need to deploy a patch to production. So let's deploy from the hotfix branch. Tag-and-release.

```
$ git tag "0.0.2"
$ gka
```


Now we've got a new release tag ready to go. Let's deploy it.

```
$ git push origin "0.0.2"
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 273 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/eddroid/wyncode_git.git
 * [new tag]         0.0.2 -> 0.0.2
```

GitHub still doesn't like this and I still don't care.

**Hack, Ship, Sink**

Now let's finish up the work we started on the master branch. I'm still on the hotfix branch, which I can see in my customized prompt, in `gitk`, and by using the `git branch` command.

```
$ git branch -a
* hotfix-0.0.1
  master
```

Let's add a new alias: `alias gb="git branch"`

```
$ gb -a
* hotfix-0.0.1
  master
```

I need to switch to the `master` branch. `git checkout` is the ticket for traveling around your git tree.

```
$ gco master
Switched to branch 'master'
```

Not that `git checkout` isn't just about time travel. Technically there are multiple "presents" in this git tree, one for each branch. Instead, we should start thinking about `checkout` as a way to move around our git tree.

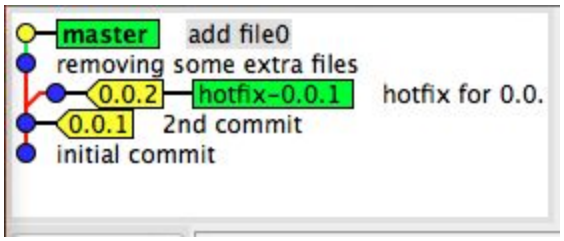
Now let's get back to work. What was the git command I recommended for helping to re-situate you after an interruption?

```
$ git log
commit f840ffdf552bf172fed7d5ba446fc4bc0856f8aa
Author: Ed Toro <ed@wyncode.co>
Date:   Mon Jun 23 18:33:08 2014 -0400

    start feature: removing some extra files
```

Right, I had just deleted some files to make way for a new feature. Let's finish this new feature.

```
$ touch file0.rb
```

```
$ ga file0.rb
$ gc -m "finish feature: add file0"
$ gka
```




I don't want to release just yet. We have a release planned for tomorrow and some of my teammates have some small features they'd like to try to get in.

The next task in my backlog is a "big new feature". It probably won't be done in time for the next release. So, instead of putting it on master, I'm going to put it somewhere else. I'm going to create a new "feature branch".

```
$ gb big_new_feature
$ gb -a
  big_new_feature
  hotfix-0.0.1
* master
```

Creating the branch doesn't switch over to it. Let's do that.

```
$ gco big_new_feature
Switched to branch 'big_new_feature'
```

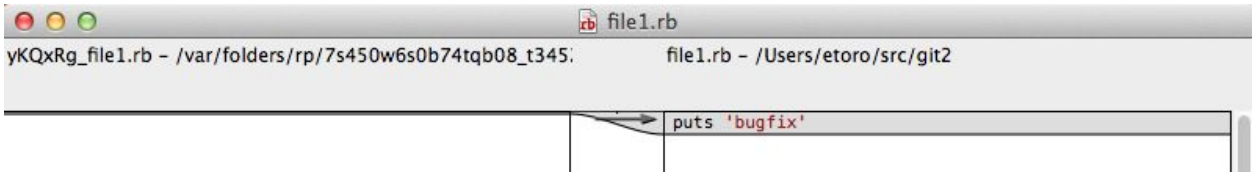
Let's get to work on this big new feature.

```
$ touch big_new_feature.rb
$ ga big_new_feature.rb
$ gc -m "start work on big new feature"
```

While working on my big new feature, I found a bug. No one seems to have noticed it yet, so I'm just going to fix it on my branch. I'm going to use the "campsite rule" - always leave the codebase better off than how you found it

(http://www.wolfgang-ziegler.com/post/Development-Practice-The-Campsite-Rule.aspx ).

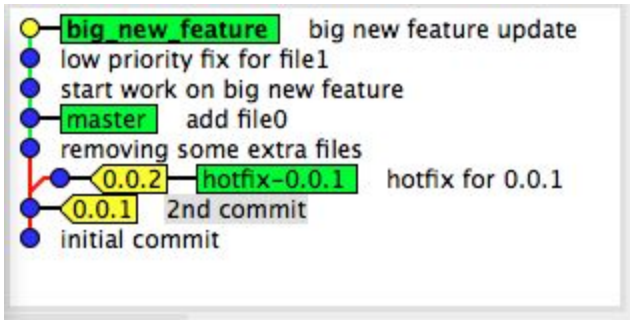
```
$ echo "puts 'bugfix'" >> file1.rb
$ gd file1.rb
```




```
$ ga file1.rb
$ gc -m "low priority fix for file1"
```

Now to continue with my “big new feature”.

```
$ echo "# this feature is going to be awesome" >>
big_new_feature.rb
$ ga big_new_feature.rb
$ gc -m "big new feature update"
$ gka
```




My “feature branch” is coming along nicely. I’ve got 3 branches in my tree now.

### *Git is a Cherry Tree*

Unfortunately, someone noticed a new bug in production. It’s the same bug I just fixed, the one in `file1.rb`. But my `big_new_feature` branch isn’t ready for production yet. I need to somehow pull my commit “low priority fix for file1” out of my branch and apply

it to the other 2 branches: `master` and `hotfix-0.0.1`. We're going to use 2 different strategies.

For `hotfix-0.0.1` we'll use `git cherry-pick`.

First I need to find the bugfix commit. I didn't tag it because it wasn't a release. And it's not at the end of a branch. `gitk` shows there are no easy-to-use labels at that point in the git tree. So I just need to get the long commit id string.

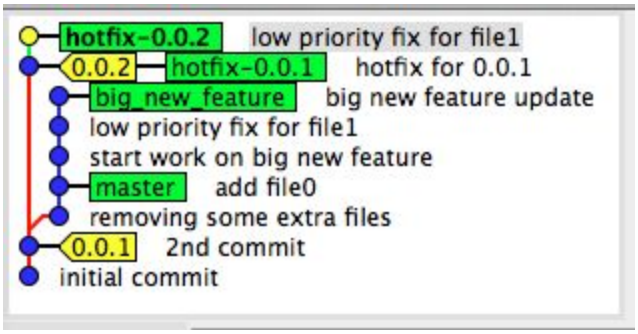
```
$ git log
…
commit 2083af6eee59518299af197444aa2a43fc35805b
Author: Ed Toro <ed@wyncode.co>
Date:   Mon Jun 23 19:45:11 2014 -0400

    low priority fix for file1
...
```

With that id saved away somewhere, I need to start working on a patch release. We've done this once already. We need to create a hotfix for version 0.0.2 of our app.

```
$ gco -b hotfix-0.0.2 0.0.2
$ gka
```

Now that I'm on my hotfix branch, let's *cherry-pick* the commit that fixes this bug.

```
$ git cherry-pick 2083af
[hotfix-0.0.2 8ff4bc0] low priority fix for file1
 1 file changed, 1 insertion(+)
$ gka
```



`git cherry-pick` *copies* a change I've made in one part of the tree and applies it to another part of the tree. So now I have two similar commits: same commit message ("low priority fix for file1"), same changes.

The `cherry-pick`ing process takes advantage of that confusing diff output we saw earlier - the one that looks more friendly to computers than to humans - the one we replaced with XCode's visual diff viewer. Git diff knows which lines of code get replaced by which other lines of code in a commit. The `cherry-pick` process creates a robot version of you, the coder, and repeats exactly the same actions you made when you fixed the bug the first time: add these lines, delete these other lines, and replace these lines with these other lines. With Git, you never have to repeat a commit. D-R-Y.

With a few lines of Git and not a single line of Ruby code, the `hotfix-0.0.2` branch is ready to go. Let's *tag-and-release* to production.

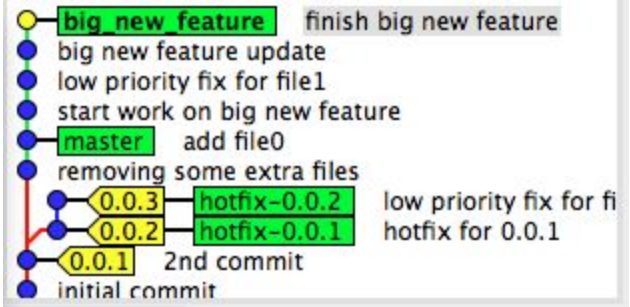
```
$ git tag 0.0.3
$ git push origin 0.0.3
```

Disaster averted. Now our app is at version 0.0.3, all patched up.

*Hack, Sink, Sink (continued)*

Where were we? Checking my git tree, it looks like I've started hacking on a feature branch. Let's finish that feature.

```
$ gco big_new_feature
$ echo "puts 'this feature is awesome'" >> big_new_feature.rb
$ gd big_new_feature
$ ga big_new_feature.rb
$ gc -m "finish big new feature"
```



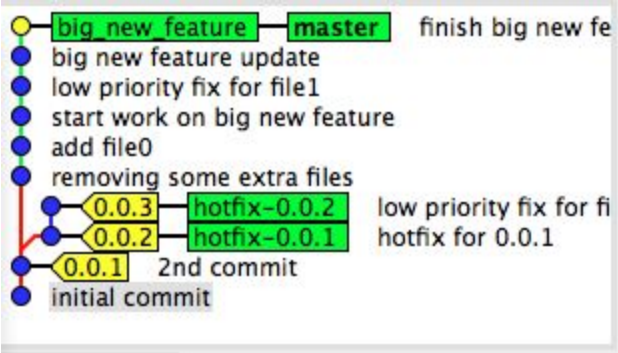
The big new feature is done. And it's done in record time. The next release hasn't even gone out yet. There's still time to bring this feature into master.

If I just wanted 1 commit from `big_new_feature` in `master`, I would use `cherry-pick`. For example, I could `cherry-pick` that low priority fix into `master` so my teammates can play with it.

But if I want *all* the commits in my feature branch to be in `master`, I should `merge` the branches together. So both my new feature *and* my bugfix will all flow into `master` at once.

When doing a git branch merge, the first step is to change over to the branch you're merging *into*. Then you merge the other branch in.

```
$ gco master
$ git merge big_new_feature
Updating e6cfce8..a40def3
Fast-forward
 big_new_feature.rb | 2 ++
 file1.rb           | 1 +
 2 files changed, 3 insertions(+)
 create mode 100644 big_new_feature.rb
$ gka
```




The `big_new_feature` branch and the `master` branch are now merged together. All the commits in `big_new_feature` were moved to the end of `master`. Note that they weren't copied (as with `cherry-pick`). The commits were actually moved.

*(TODO: What if someone added to `master` while we were hacking on our branch? Need an example of a more realistic feature branch merge.)*

The team has decided that the `master` branch is ready for deployment as version 0.1.0.

Unfortunately, because there have been a lot of bugs in production lately, some QA testers and managers would like to play with it first. We only have a production environment. We don't want to deploy demo code there. So we need to define a staging environment.

Typically the `origin` remote would be linked to GitHub. But today we've been treating `origin` as if it were a production server. So let's make that explicit by copying `origin` to a new remote named `production`.

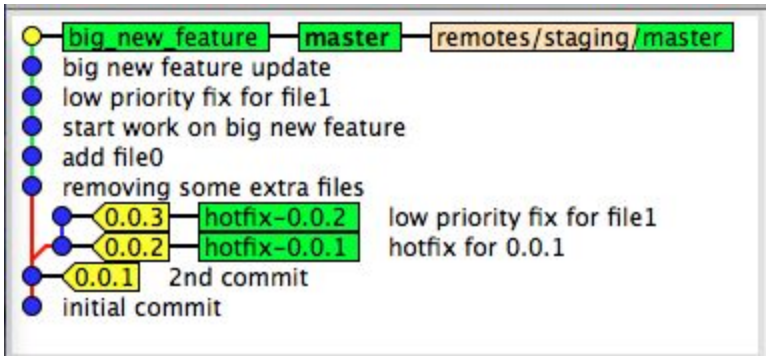
```
$ git remote add production
https://github.com/eddroid/wyncode_git.git
```

Now we can create a `staging` remote linked to another environment. Typically you would create a new Heroku app (or whatever host you're using) and add it, but we'll just take a shortcut and keep using the same GitHub repo in our example.

```
$ git remote add staging
https://github.com/eddroid/wyncode_git.git
```

The dev team has certified that the master branch is ready to be released as version 0.1.0. QA would like to test it first. So we need to deploy the master branch to the staging server.

```
$ git push staging master
$ gka
```




Note that, with our new git remote naming convention, the git commands become more readable. I want `git` to `push` (to where?) to `staging` (what?) the `master` branch.

This is the first time I've pushed a remote branch (rather than a tag). Remote branches appear in the git tree as well. They're just like local branches, only remote.

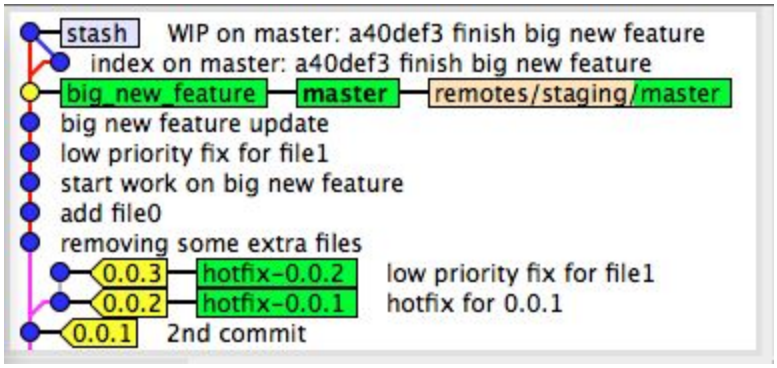
The big release is coming up soon, but I think I can fit another small feature in. I start working on it.

```
$ echo "puts 'Hello world\!'" >> file0.rb
$ gst
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:   file0.rb

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

Uh oh, another bug reported in production. I'm not ready to commit this code change yet. I'm still working on it. What can I do? I can `stash` it.

```
$ git stash
Saved working directory and index state WIP on master: a40def3 finish big new feature
HEAD is now at a40def3 finish big new feature
$ gst
On branch master
nothing to commit, working directory clean
```

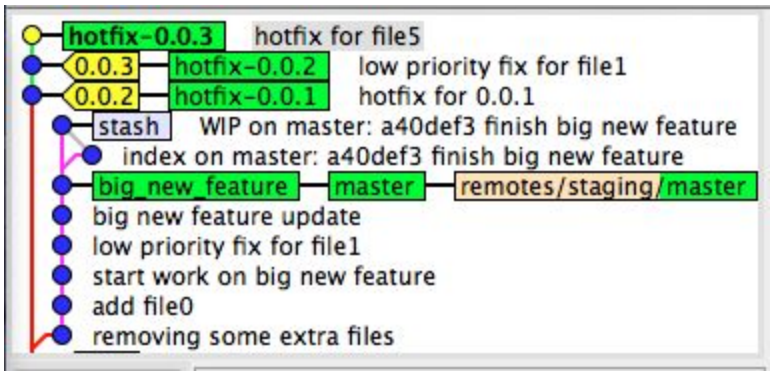
$ gka




The stash is a temporary branch I can use to store some work that I haven't committed yet. It's not a commit and it won't get deployed in the next push. It just takes whatever I'm currently working on and puts it to the side.

Now I can go work on that bugfix.

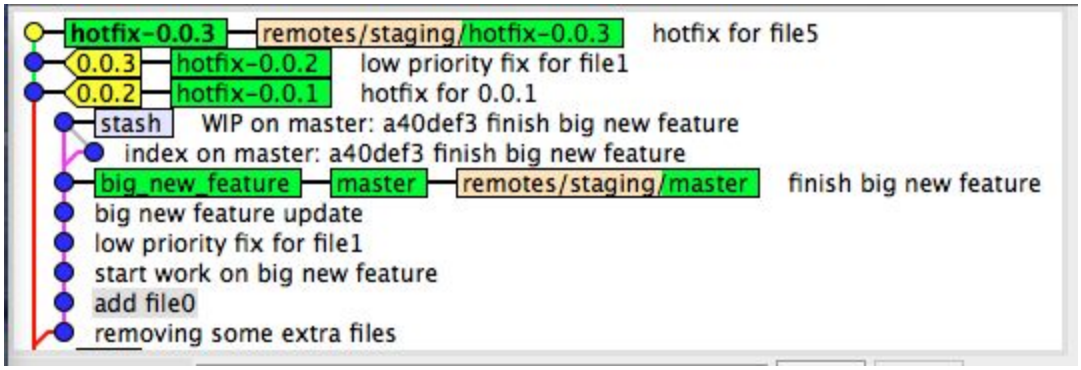
$ gco -b hotfix-0.0.3 0.0.3
Switched to a new branch 'hotfix-0.0.3'
$ echo 'puts "Hello Wyncode"' > file5.rb
$ ga file5.rb
$ gc -m "hotfix for file5"




I've got a staging environment now, so let's test this hotfix before deploying to prod.

$ git push staging hotfix-0.0.3
Counting objects: 8, done.
Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/eddroid/wyncode_git.git
 * [new branch]      hotfix-0.0.3 -> hotfix-0.0.3



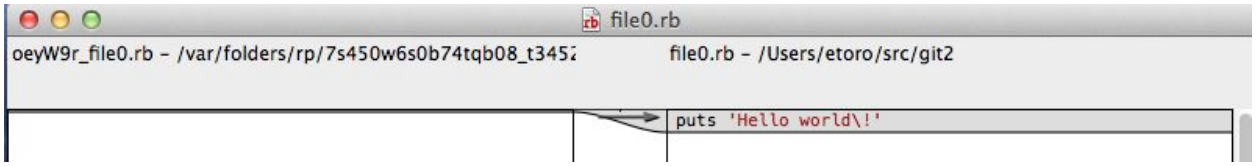

Now my staging remote is tracking the hotfix branch. Our team tests it and it checks out. So let's tag and release.

$ `git tag "0.0.4"`
$ `git push production "0.0.4"`
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/eddroid/wyncode_git.git
 * [new tag]         0.0.4 -> 0.0.4

The prod environment (origin remote) is tracking version 0.0.4 of the app. Let's go back to the master branch and pick up where we left off with that half-done feature.

$ `gco master`
Switched to branch 'master'
$ `gst`
On branch master
nothing to commit, working directory clean
$ `git stash list`
stash@{0}: WIP on master: a40def3 finish big new feature
$ `git stash pop`
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:   file0.rb

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (af5605b6126e8d745932e9067a5e130115d800ad)
$ gst
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:   file0.rb

no changes added to commit (use "git add" and/or "git commit -a")
$ git stash list
$ gd file0.rb
```




And now I'm back right where I left off. Let's finish this feature and commit.

```
$ echo 'puts "Hello world!"' > file0.rb
$ ga file0.rb
$ gc -m "last new feature before the big release"
```

Now it's time for the big release. Let's deploy the master branch to staging for one last test.

```
$ git push staging master
```

The team says everything checks out. Wonderful. Let's tag and release. This is a big release with new features, so let's make this version 0.1.0.

```
$ git tag "0.1.0"
```

```
$ git push production 0.1.0
```

It's the end of the day on a Friday, so everyone goes home early. Job well done.

Never release on a Friday!
https://twitter.com/davidwalshblog/status/893522540693475328

Except it's not. There's a bug in this release. And no one is around to fix it. You get a call late on Friday night. They want you to do something about it. They don't know what, just do something.

You don't have the time or desire to spend Friday night tracking down a bug. Instead, let's just roll this release back to the last good version. Version 0.1.0 is no good. All that testing on staging was for nothing. We need to go back to version 0.0.4.

Usually rolling back a release is not easy. In a Rails app you have to worry about rolling back migrations first. Sometimes a migration is irreversible. Sometimes you just have to pray things will turn out ok.

As far as the code is concerned, however, rolling back to a previous version is easy. Just deploy the last good version of the code.

```
$ git push production "0.0.4"
```

With Heroku, it's as simple as that. With a standard cloud server you'd have to login to the server remotely and run `git checkout 0.0.4`. But the end result is the same. Git stores all previous versions of your code. Returning to a previous version is as simple as doing a `checkout` back to a previous release tag.

*TODO: Rolling back database migrations before a code rollback.*

There are many Git workflow strategies. The most common ones like to use lots of branches and no tags. I like more tags than branches. Either way, the general concept is the same.

- One *release tag* (or *branch*) represents production.
- One *release branch* represents staging.
- Many *feature branches* may represent big features that are currently "in progress".

- Production releases are `tag`ged for easy reference.
- You can use `checkout` to jump to any tag or branch.
- You can deploy any tag or branch. (I prefer to deploy tags to production and branches to staging.)
- You can copy individual commits between branches with `cherry-pick`.
- You can merge branches together with `merge`.
- You can temporarily put your work on hold with `stash`.

And that's how you can use Git to manage the final steps of the SDLC.

In general, if you're ever confused about what is possible to do with git, think about the git tree. Anything you can do to a tree, you can do in git. Create, merge, relocate, or chop off branches. And pick cherries. Don't forget about the cherries.

p.s. It's sometimes useful to delete old branches when you're done with them, just to reduce clutter.

```
$ gb -a
  big_new_feature
  hotfix-0.0.1
  hotfix-0.0.2
  hotfix-0.0.3
* master
  remotes/staging/hotfix-0.0.3
  remotes/staging/master

$ gb -d big_new_feature hotfix-0.0.1 hotfix-0.0.2 hotfix-0.0.3
Deleted branch big_new_feature (was a40def3).
error: The branch 'hotfix-0.0.1' is not fully merged.
If you are sure you want to delete it, run 'git branch -D hotfix-0.0.1'.
error: The branch 'hotfix-0.0.2' is not fully merged.
If you are sure you want to delete it, run 'git branch -D hotfix-0.0.2'.
error: The branch 'hotfix-0.0.3' is not fully merged.
If you are sure you want to delete it, run 'git branch -D hotfix-0.0.3'.
```

Sometimes Git recommends you don't delete old branches. That's ok too. Those hotfixes weren't merged into master. Deleting those branches could cause those commits to be lost.

Feature branches are typically deleted, completing the Hack/Ship/Sink metaphor:

- *Hack*: create the feature branch and make some commits
- *Ship*: merge the branch
- *Sink*: delete the branch

## If there's time…

`git pull --rebase`
This helps to keep your git tree clean when you work with others and pull frequently. It skips a merge commit if it's not necessary. But if there's a merge conflict, things can get complicated. If you have problems, do a `git rebase --abort`, then try a regular `git pull` instead.

This all is similar to this popular git branching model (GitFlow):
http://nvie.com/posts/a-successful-git-branching-model/

Other Git workflow strategies:
https://www.atlassian.com/git/tutorials/comparing-workflows

`gitk` alone (without `--all`) hides the other branches, which can get confusing. Default to using `--all` until you get the hang of things.