Git & GitHub

Configuring Git

There are 3 levels at which Git can be configured.

1. System level configuration (Default) - apply to every user of this computer
2. User - apply to a single user
3. Project - on a project by project basis (we usually use this)

To change a configuration setting, use the Git command line tool with the following

`git config --<level to be modified>`

e.g.

System level: `git config --system`

User level: `git config --global`

Project level: `git config`

Types of configurations you can make:

```
git config --global user.name “Ben Grunfeld”    //sets usernam
e
git config --global user.email “benjamin.grunfeld@gmail.com //
sets user email
git config --list                               //shows what y
ou did
git config user.name                            //shows userna
me
git config user.email                           //shows user e
mail
```

To see where they are located:

`cd ~` returns you to the main user directory
`ls -la` shows you the directory listing (same as dir in windows)

There you will find a file called `.gitconfig`
Because it's a dot file, it will try to hide it from you, so you will only be able to see it in the command line tool, you won't be able to see it in the finder

To see what's inside, use: `cat .gitconfig`

Tell Git What Text Editor You'll Be Using

```
git config --global core.editor “name of the editor we want to
 use”

e.g.
git config --global core.editor “emacs”
```

`-w` after launch, wait until textmate is done until you keep going with what you were doing `l1` means start at line 1

so

```
git config --global core.editor “mate “-wl1”
```

will choose TextMate with the above settings

Text Color:

If we use this command, Git will try to use colors to convey meaning in text.

```
git config  --global color.ui true
```

Installing Auto Completion

this is only for Mac, since Windows should already have auto completion installed

Download from Github

```
cd ~
curl -OL https://github.com/git/git/raw/master/contrib/complet
ion/git-completion.bash
Rename File
mv ~/git-completion.bash ~/.git-completion.bash
```

Edit the File
Now we want to edit the .bash_profile

```
nano .cash_profile

if [ -f ~/.git-completion.bash ]; then
source ~/.git-complation.bash

fi
```

Now auto-completion is turned on and you can check that by typing

`git h`

and then pressing tab. It should auto complete and become

`git help`

Initializing a Repository

Once you've configured Git, it's time to initialize a repository, and tell Git that it has to start tracking something.

To do this, we use the command

```
git init
```

The first step is to choose what directory you want to set up as a repository. I did this in
Finder -> Document -> First_Git_Repo

The .git directory is the directory where Git stores all of its tracking information

No matter how many levels deep you go, Git will always store its information in the top level in the .git directory. Think of it as Git's workspace.

If you want to get rid of Git for that directory, it is as simple as deleting the .git folder, since this deletes all the tracking information.

To check what's inside, use the command

```
ls -la .git   //this will print out the contents of the direct
ory.
```

Usually you won't want to touch anything inside here, since it's Git's job to edit stuff in here, but you may want to edit the config folder in there, since it holds the per-project configuration settings.

Git Commit

The first commit tells Git that it has to start tracking things. We need to make some type of change so that we have something to commit.

I created a text file and saved it in my directory, then use the following command to tell Git to add all changes to my project.

```
git add .
```

The dot means "all the changes that were made in this entire directory and push all of them up to the staging index"

```
git commit -m “Initial commit”
```

SHORTCUT: If you want to add something and commit it at the same time, i.e.

bypass the staging step, you can use:

```
git commit -a
```

The issue is that it simply grabs everything in your working directory that hasn't been added, adds it, and then commits it. Also, files that are not tracked and files that are being deleted do not get included in this. So it works well with modifications, but not with deletions, or un-tracked files.

With a message:

```
git commit -am "Changed phone number"
```

The Basic Work Flow

So the above steps constitute the basic work flow:

1. Make Changes

2. Add the Changes

3. Commit the Changes to the Repository with a Message

Commit Message Best Practices

- Short single-line summary (less than 50 character)
- Optionally followed by a blank line, and then a more complete description
- Keep lines of more complete description to less than 72 characters
- Bullet points are usually asterisks (*) or hyphens (-)
- Can add ticket tracking numbers from bugs from support requests
- Can develop shorthand for your organization - but EVERYONE must agree to it/adopt it!

e.g.

```
[css,js]
“bugfix:  ”
“#38903 -  ”
```

If you’re just making a small change, then a single-line summary should do it. But if you’ve made changes to multiple files, with many different types of changes, then you may want to document that in the more complete description

Before, we just used the double quotes to house the message, but that can become tricky with multi-line messages. For that, you may want to use a text editor.

Bad: “Fixed typo”
Good: “Added missing > in project section of html”

Bad: “Update login code”
Good: “Change user authentication to us Blowfish”

Checking the Git Log

To check the Git log

```
git log     //checks the Git log
```

This will bring up a log of all the changes made, plus their messages.

Each commit has a unique ID, the authors name, their email, date/time committed, and the message.

```
git log -n 5                    //returns the last 5 commits
git log --since=2012-05-16      //returns every commit made si
nce 5/16/2012
git log --until=2012-05-16      //returns every commit made un
til 5/16/2012
git log --author="ben"          //return commits from any auth
or named Ben
git log --grep="init"           //return every commit with the
 word init in it                                  //e.g. "init
ial file upload"
```

Author: author name doesn't have to be the full or exact name.

Grep: "Global Regular Expression Search", and will allow you to search for any part of a commit message.

This way if you have a convention in your commit messages (as above) then you can use grep to search for all bug fixes with. E.g.

```
git log --grep="bugfix:"
```

Git Commands

Git Status

Git Status reports the difference between the repositorty, the staging index, and the working directory. i.e. It will tell us the difference between those 3 different trees.

```
git Status
```

Git Diff

In the unix world, it is very common to use a program called diff in order to compare 2 files. Git uses that as the term to show us a diff between an old version and a new version.

```
git diff
```

This will compare what is in the working directory with what's in the staging index.

E.g.

```
Ben:GitTest bengrunfeld$ git diff
diff --git a/1st_file.html b/1st_file.html
index bb405f1..45c87ee 100644
--- a/1st_file.html
+++ b/1st_file.html
@@ -1 +1 @@
-<h1>This is my 1st file. Change 2!</h1>
\ No newline at end of file
+<h1>This is my 1st file. Bongo Bongo Drums</h1>
\ No newline at end of file
```

The --- denotes the old version The +++ denotes the new version

Then it goes on to tell you all the textual differences between the files.

In a long document (e.g. 300 lines), it will only show you the lines that changed.

The @@ -1 +1 @@ tells you what line numbers have changed. (-) old file, (+) new file.

You can check for all differences in all files in the repo vs. working directory

E.g.

```
git diff                    //checks the difference between al
l files
git diff 1st_file.html      //only checks for differences betw
eens versions
                            //of 1st_file.html
```

Wrapping Long Lines

If the results that came back have multiple lines, or very long lines that go off the terminal window, you can enter a command that wraps lines.

`minus sign (-) + SHIFT + s + RETURN`

Coloring Words

```
git diff --color-words filename.txt
```

This causes the changes to appear side-by-side, where only they are colored. Easier to spot differences this way.

Checking The Difference Between Files in The Staging Index & Files in The Repository

```
git diff --staged            //will check diff between repo and
 staging index
```

Whereas `git diff` checks the difference between the staging index and the working directory, `git diff --staged` checks the difference between the repository and the staged index.

Deleting a File

There are 2 ways to delete a file with Git.

Way #1

1. Add a file to the staging index, then commit it.
2. Delete the file using the Finder
3. Got to Git and if you type `git status`, it will show you the file has been deleted
4. Use `git rm filename.txt` to indicate to the staging index that you want it removed

5. `git commit -m "deleted the file"` to commit the delete to the repository.

Way #2

1. Just do a straight `git rm filename.txt`, although this does a UNIX remove, not a "lets put it in the trash" remove.
2. `git commit -m "removed the file completely"`

Moving and Renaming Files

Way 1

Similar to deleting files, the first way we can do it is in the Finder, and then update the changes via Git.

1. Change file names in Finder
2. If you type `git status` now, it will tell you that there is one file that hasn't been added, and there's one file that has been deleted.
3. Type `git add newfilename.txt` and then `git rm oldfilename.txt`
4. `git status` will now tell you that the file has been renamed. If over 50% of the file's data is the same, it will acknowledge it as a rename.
5. Finish with a good ol' regular `git commit -m "changed filename"`

Way #2

In Git, a rename and a move use the same command. This is very similar to UNIX.

1. `git mv oldfilename.txt newfilename.txt`
2. If you run `git status` now, you'll see that it acknowledges that a file has been renamed.
3. `git commit`

Moving a File

To move a file, use the same command.

1. Create a folder in Finder
2. Type `git mv oldfile.txt new_dir/newfile.txt`
3. If you type `git status`, you'll see that it has been added to the staging index
4. `git commit`

Undoing Changes in Git

This applies to undoing changes you've made to your working directory, the staging index, or even changes that have been committed to the repository.

Undoing Changes to the Working Directory

If you make a change to a file (e.g. delete a div full of content in an HTML file), and then decide that that was a mistake, you can get Git to undo the change. If you type in `git status`, it will show you that the file has been modified. Then if you type in `git diff`, it will show the exact text that has been modified.

To bring the repository version back, use

```
git checkout <filename.txt> or <directory_name>
```

`git checkout` goes to the repository, grabs the thing that has the same name as what we put in to the command, and then places it in our working directory.

But say for instance you have a directory called *production*, but you also have a branch called *production*, then `git checkout production` will check out the branch, NOT the directory that you wanted. The way to deal with this is to use:

```
git checkout -- production
```

The `--` ensures that you stay within the current branch.

Undoing Changes to the Staging Index

This is also referred to as 'unstaging' things that we've staged there.

If you're putting together a commit of multiple files, and you accidentally add a file that you actually don't want to commit, you can remove it from the staging index without changing the file in your working directory. This has the effect of taking whatever is in the staging index and placing it in the working directory. Here is the code:

```
git reset HEAD filename.txt
```

Undoing Changes to the Repository

This is also referred to as amending commits.

All the information regarding a commit is referred to via it's SHA-1 hashed value. If we change anything in the files in the commit, the SHA-1 hashed value changes. So if we change something a couple of commits back, it will break everything (destroy data integrity) all the way up the chain, since each commit points to the SHA value of its parent.

What we CAN do is change the lastest commit, since nothing depends on it yet. The commit at the end is still editable, but once there's a commit after that, you cannot edit it anymore. You can only edit the latest commit.

When would we use this?

If we made a commit with a whole bunch of files, but then we realized that we wanted to change something in one of the files that had been committed, because commits represent change sets, or a grouping of related changes, then we would do the following:

1. Make the changes you wanted to your file
2. Add your file to the staging index with `git add filename.txt`
3. Commit the change with the `amend` option, which tells Git to make a

change to the last commit performed.

.

```
git commit --amend -m "Updated camping gear list"
```

But even if you only want to update the message on the last commit, but not actually change any files, as long as there is nothing in your staging index, you can use the same command to update the previous commit message. E.g.

```
git commit --amend -m "Updated extreme camping gear list"
```

Making Changes to Older Commits

With `amend`, we can change details about the last commit made, but if we want to make changes to *older* commits, the best thing to do is to make *new* commits. i.e. commits that undo what was done in those older commits.

This maintains the data integrity of git, but also ensures that the log file is accurate.

In order to make new commits that undo changes:

1. We could manually make those changes and then commit the result
2. We could checkout an old copy of the repo that has the unchanged file and then use that.

.

```
git checkout 6c85e313b136664 -- 1st_file_eva.html
```

We only used the first 10 or so numbers from a SHA value, and the `--` means that we want to explicitly stay in the current branch.

This will move the file into the staging index, where you can mess around with it. This is different from other checkouts. With regular checkouts, it will put the file into your working directory, but when you make a checkout from a revision, it will put it into your staging index and it will have the prefix of `modified:`.

If we do `git diff --staged`, we'll be able to see the difference between the existing file in our repository, and the file in our staging index.

If we were to commit it like this, it would revert to the old file, which is in the staging index. Functionally, this is the same as undoing the old change.

It's good practice that when you're reverting a particular commit to refer to the SHA for that commit. So just copy and paste the SHA you were using into the commit message.

To Undo the Changes for a Commit - Completely & Totally

Use:

```
git revert
```

What `revert` does: it will take all the changes that are there, and flip them around. It will do the exact opposite of what the changes were. If something was added, it will remove it.

Use part of the SHA value, like we did above, to identify the commit to revert. E.g.

```
git revert 6c85e313b136664
```

You can also use `-n`, which will cause git to stage it, but won't make the commit. Then you can write your own message, make your own modifications, etc, and have more control.

If you need to revert something more complex, you'll need to do a **merge** between the current branch and the new set of changes you're trying to merge into it.

Undoing Multiple Commits with Reset

It's powerful, but very dangerous, so use with caution.

```
git reset
```

`git reset` allows us to specify where the HEAD pointer should point to. Normally, we just let Git manage where the HEAD pointer is pointing to for us. e.g. with normal commits.

Here, we're telling Git we want to be in control, and that we want to move the HEAD pointer to a specific place, and that's where Git will start recording from now on. That's where Git will start making its commits.

With Audio, if you rewind and then hit record, it records over everything that comes after. `git reset` works the same way. Once you specify a point for HEAD to point to, it will start recording from there, and everything that comes after it will be erased.

`git reset` ALWAYS moves the head pointer.

```
git reset --soft
```

`--soft` moves HEAD to the specified commit, but does not change the staging index or working directory. Just moves the pointer. It's the safest of all these options, which is why it's called soft. i.e. Move the pointer and do NOTHING else. If we rewind backwards, the staging index and working directory will contain the files in their later revised state. The repo will be set back to an earlier version.

```
git reset --mixed
```

`--mixed` is the default. It moves the HEAD pointer to the specified commit and then changes the staging index to match the repository. It does not change your working directory though. So the working directory still contains all the changes that we've made. So we haven't lost any work. It's just waiting for us to stage it and commit it.

```
git reset --hard
```

`--hard` is the most dangerous of all. It changes your staging index and your working directory to match the repository at the specified commit. That means any changes that came after that commit are completely oblierated.

E.g.

```
git reset --soft e7a2c8694e98       //will do a soft reset to
that commit
```

To see what the HEAD pointer is pointing to

If we're on the master branch

```
cat .git/HEAD                       //will show the directory HEAD
 is pointing to
cat .git/refs/heads/master          //will show the SHA value HEAD
 is pointing to
```

Best Practice

Before you use `git reset`, copy the screen of `git log` to a text file and save it, so that you can grab the SHA values if you want to go back to the latest commit, and undo the rollback.

As long as you have the later commit SHA value, you can go back to it.

Unstaging a File Explained

To unstage a file, we use

```
git reset HEAD <filename.txt>
```

What this does is that it pulls the file from the HEAD to make the staging directory match it ***Need More Clarity!!!!***

Removing Untracked Files from our Working Directory

If there are a lot of files in our working directory that we haven't added, and we want to get rid of there, there is an easy way to do this. We could delete them one by one, but there is a fast way that Git gives us to accomplish this.

```
git clean
```

This by itself won't do anything. It will issue a fatal error and complain that it need either a `-f` or a `-n` option.

`-n` is a test run. It then tells you the files that it would remove. `-f` will force the clean, throwing out any files that haven't been added. Files will be permanently deleted - i.e. not stored in trash can.

Ignoring Files

When we add a file to the working directory, it is then tracked by Git as being unstaged, so if we enter `git status`, it will show us that the file is unstaged. But what if this is a log file that is constantly changing? Then `git status` would always bring attention to it, which could be kind of annoying.

What we need is a way to tell Git to ignore this file, so that we can live our lives in peace.

Solution:

1. Create a file called `.gitignore` in the main project directory (in my case GitTest).
2. This file will provide Git with a set of rules it can use to know which files to ignore, and which files it can use for commits.
3. We can list the files we want to ignore, or we can use a basic version of regular expressions.

Permitted Regular Expressions

? * [aeiou] [0-9]

We can also negate expressions by putting a ! first.

E.g.

```
*.aspx                   //ignore all .aspx files
!sample.aspx             //but DON'T ignore sample.aspx. Still
track it please.
/project/logs/           //ignore all files in the logs directo
ry - must have
                         //trailing slash (/)
#                        //comments the line out
```

You can use a text editor to create the file, but because it has a preceding dot, it's hard to see in the finder, because that means it's hidden. Better to use a command line command. For windows, just use Notepad.

```
nano .gitignore          //will create a new file in the curren
t directory
```

Enter in the rules (as above) about which files you want Git to ignore.

Good Stuff to Ignore

- compiled source code
- packages in compressed files (zip, tar.gz, gz, disc images, etc)
- logs and databases (i.e. files that change regularly)
- operating system generated files (.DS_Store, trashcan, etc)
- user uploaded assets (e.g. images, PDF's, videos)

Two great resources that deal with this issue:

Git Help Article on Ignoring Files This gives you good guidelines on what to ignore in general.

Git Repo About GitIgnore This repo suggests what to ignore by language (e.g. Javascript)

Globally Ignoring Files

We can configure Git to globally ignore files. That means Git will:

- ignore files in all repos
- settings not tracked in repo
- user specific instead of repo-specific (only configured on my machine)

User specific configs are great because if I'm using a Mac and someone else is using Windows, it means that my computer will be configured for me, and they won't suffer.

```
git config --global core.excludesfile ~/.gitignore_global
```

The file can be named anything that you want, and it can be in any directory that you want, you just have to tell Git where it is.

```
~               //represents your user directory in UNIX
```

Ignoring Files That Have Already Been Tracked

Previously, we told Git how to ignore files that hadn't been added yet. Now we will learn how to tell Git to ignore files that are currently being tracked. Git will NOT ignore a file that was already added previously, but that later had a rule in the `.gitignore` file put in.

First you need to commit the added file. Only afterwards can you tell Git to ignore any future changes that are made to it.

If we just add the filename to `.gitignore`, Git will still track the file. What we need to do is tell Git to stop tracking the file.

One way we could do this is by removing it by `git rm file.txt`. But this would remove the file altogether. If we only want to tell Git to stop tracking it, but not destroy it, we could use the following:

```
git rm --cached file.txt
```

This will tell Git to remove the file from the staging index, NOT the repo or the working directory, JUST the staging index. This will cause the file to not be tracked.

Then you'll need to perform a commit to seal the deal.

SUMMARY: to ignore a file that has already been added, you need to

1. create a rule for it in the `.gitignore` file
2. tell Git to stop tracking it with `git rm --cached`
3. commit the change to seal the deal

Tracking Empty Directories

One thing that surprises new Git users is that Git does not track empty directories, because Git is designed to be a file-tracking system. The moment there's a file in the directory, Git will start tracking it.

So to track empty directories, we need to put a file in them.

Most people use `.gitkeep` in the empty directory. It can even be empty.

In UNIX, we can use `touch`.

`touch` is a way in UNIX to create a file that doesn't exist. e.g.

```
touch exampledir/emptydir/.gitkeep
```

If you use `git status`, it will then see the directory, then you can add it and then commit it.

SUMMARY: Git keeps track of files, not directories. Directories are only a way to get to files.

Navigating the Commit Tree

The tree is the structure of files in the Git repository, similar to a directory in your file system. In Git, Treeish means something that references part of the tree. It's suffixed with 'ish' because that something that vary widely.

A `tree-ish` is a reference to a commit, because that commit in turn references the tree (i.e. the Git repo and all the files that are in it at that point). Essentailly, it's just something that points at a commit.

Tree-ish Methods:

These are all considered Tree-Ish'es:

- use the full SHA-1 value
- use a short version of th SHA-1 value (at least 4 char, but 8-10 better)
- HEAD pointer
- branch reference, tag reference
- ancestry: use any of the methods above, then refer to that object's ancestry

Using Ancestry to Point to a Commit

```
HEAD^
af3jklt5^
master^
```

The carrot (^) at the end of the value is used to reference it's parent (think of it pointing up).

```
HEAD^^
af3jklt5^^
master^^
```

Means 2 generations back (^^), but already it would be easier to just use the tilday (~) notation.

```
HEAD~1
HEAD~
```

We can also use the tilday character (~) by itself to indicate one generation, or with a number to specify how many generations back to go (e.g. parent-1, grandparent-2, etc)

Navigating the Tree Listings

Like the `ls -la` command in UNIX, Git accepts an `ls-tree` command but required a tree-ish, meaning something that points to a commit.

```
git ls-tree <tree-ish>          //lists the tree in the repo,
but requires a                                   //tree-ish
```

e.g.

```
git ls-tree HEAD                //will list what is in the rep
o that HEAD is                                   //pointing t
o.
```

So if we were to check this repo out with `git checkout`, that list of files is what we'd receive. It may not match your working directory exactly if you've used `.gitignore` on some files or directories.

We can also use

```
git ls-tree master              //points to the latest commit
of the master                                    //branch

git ls-tree master video/       //checks what is in the video
directory in                                 //master


git ls-tree master^ video/      //checks master's parent video
 directory
```

Left of the object's SHA-1 value, you can see once of two options - `tree` or `blob`. A `blob` is a file. A `tree` is a directory.

e.g.

```
100644 blob 45c87eeabd3609358e1b64a68d404aafc2623862 1st_file_
eva.html
100644 blob 2e20015d653caca69188e291c6649f467e21653a 2nd_file_
eva.html
040000 tree 53653461fa6d841c372af34e64bdc21484715be5 test_dir
```

You can use the SHA value, so if it's a directory like test_dir above, we could use:

```
git ls-tree 53653461fa6d84
```

And it would list the contents of that directory (test_dir)

Getting More From Your Log File

There are a LOT of information in the Git logs.

```
git log --oneline
```

Possibly the most useful log command, because it gives us a one line list of what's in our log file and still gives us part of the SHA

```
git log --oneline -3                    //will return 3 commit
s
git log --oneline -5                    //will return 5 commit
s
```

Returns something similar to `git log --oneline` , but with the full SHA.

```
git log --format=oneline
```

We can also filter by time, like so:

```
git log --since"2013-03-20"             //will return all comm
its since date
git log --after"2013-03-20"             //will return all comm
its since date
git log --until"2013-03-20"             //will return all comm
its until date
git log --before"2013-03-20"            //will return all comm
its until date
```

We can also combine the commands:

```
git log --since"3 weeks ago" --until"5 days ago"
git log --since3.weeks --until5.days
```

We can filter by author:

```
git log --author="ben"                  //will show commits by
 all bens
git log --author="ben grunfeld"
```

We can also use GREP (Global Regular Expression):

```
git log --grep="update"
```

Here you can see why having really good commit messages is helpful

We can also specify a range using SHA-1 values, so starting at a particular SHA-1 value and returning every commit until another SHA value. To do this, we use `..` to indicate that we want a range.

```
git log faj38gj..au9938jf --oneline
```

We can use the oneline option to make things clearer.

```
git log faj38gj..
```

Means everything from this commit until the latest commit.

```
git log fjakl893.. pilot.php
```

Means from this commit until the latest commit, what has happened to the `pilot.php` file. i.e. give me the logs that affect that file.

Will return the commits connected to that file, plus info on whatever changed.

We can find out more info about the commits by using

```
git log -p                          //p stands for patch
```

This will show us a `diff` of what changed in each commit

We can even combine these options like so:

```
git log -p fjakl893.. pilot.php
```

To get statistic about what changed in each commit, we can use:

```
git log --stat --summary
```

These can be used together or alone.

```
git log --format=oneline
```

This returns something similar to `git log --oneline`, but with the full SHA. We can also do:

```
git log --format=short
git log --format=medium         //default
git log --format=full
git log --format=fuller
git log --format=email          //format useful for sending em
ails
git log --format=raw            //just the raw info
```

And here's a cool one: This shows us a graph of our commits. Will show us branches and merges.

```
git log --graph
```

Now SUPERFUNKY!

```
git log --oneline --graph --all --decorate
```

Will show us where the HEAD is pointing and the name of the current branch.

Examining a Particular Commit

To see what changed in a particular commit, we use:

```
git show <SHA Value>
```

e.g.

```
git show fj32890jf
```

This shows us a `diff` as well.

`--- dev/null` means that it didn't exist before

We can also use `format` on `show`. E.g.

```
git show --format=oneline HEAD
```

Here, we've used the HEAD as the SHA. (You can use blobs, trees, tags and commits).

To see the same thing one generation back:

```
git show --format=oneline HEAD^
```

If we passed in a tree, it would show us the names of everything in the tree. This is the same as using `git ls-tree --name-only <SHA>` .

We must a TREE-ISH with `show` . We can't use a file-name as a tree-ish. We must use its SHA instead.

Comparing Two Different Commits

This doesn't compare the commit snapshot (i.e. the changes that were made that were stored with that commit), it compares the directory that that commit references. I.e. The state of all the files in the repo at that point in time.

When we compare commits, we're actually comparing 2 directories, and seeing what has changed between those 2 directories. It might be over time (commit made after 5 days) or we can compare 2 different branches to see how they differ.

To do this we use

```
git diff
```

If we pass in a SHA, `diff` will show us the difference between our working director, and the directory at the point that that commit was made. E.g.

```
git diff fdsa89j3
```

So we can compare where we are now against a previous point in time.

We can even pass in the name of a file to see the difference between that file in our working directory, and that file in the directory of the commit that the SHA references. e.g.

```
git diff fdsa89j3 blubber.aspx
```

We don't just have to use our working directory, we can compare any two commits.

We can pass in 2 different tree-ish'es using a range.

```
git diff nei943nf3..jdin39dnc

git diff nei943nf3..jdin39dnc blubber.aspx
```

You can pass in any tree-ish here.

```
git diff nei943nf3..HEAD

git diff nei943nf3..HEAD^^
```

These ones are kind of useful

```
git diff --stat --summary nfa3j4j4..HEAD
```

This following one ignores any changes that were made that were purely whitespace:

```
git diff -b 8j3edoc8..HEAD
git diff --ignore-space-change 8j3edoc8..HEAD
```

The second longer option `--ignore-space-change` is exactly the same as typing `-b`. It will ignore is someone changed 1 space to 2 spaces, or 5 spaces to 10 spaces.

```
git diff -w 8j3edoc8..HEAD
git diff --ignore-all-space 8j3edoc8..HEAD
```

These 2 options are also exactly the same, and they tell Git to ignor ANY changes made to space.

This is important because usually, changes to space aren't something we're interested in, so with this, we can filter them out.

Branching

Using branches is good for:

- Trying out new code - if it doesn't work, then you delete it. If it does work, then you fold it back into the master directory.
- Collaboration - very easy to create a branch of a particular feature and build it out while everyone else is still working on the master branch, then merge it back in without anyone being affected

When we create branches, we're still going to have just one working directory. All the files that we're working with will still be in that same project folder as before.

When we switch branches, Git will use *Fast Context Switching*. It's going to take all the files and folders in the working directory and make it match what's in the branch.

It will swap out the two sets of changes.

So if we're working in our master branch, and then we change to the photo sharing feature branch, now our working directory will have all of the photo sharing feature changes in it. If we switch back to our master brach, all those changes will go away, and will be replaced by the files and folders that are in the master branch.

Git will handle swapping out all those files for us, making all those changes.

Branching Commands

```
git branch
```

Will show you all the branches present in your local repository.

If you haven't created an branches, this will show up.

```
* master
```

The single asterisk (*) lets you know that this is the currently checked out branch, aka where you are right now.

As we said earlier, the reference for the HEAD pointer is stored in the `.git` folder.

To find out where HEAD is pointing, we use

```
cat .git/HEAD
```

This returns:

```
ref: refs/heads/master
```

HEAD is telling us that it is in the master branch, and we go in there to find out what the SHA value is.

If we do the following, it will show us a directory listing with `master` in it, but if we create more branches, this is the directory where we will see them.

```
ls -la .git/refs/heads/
```

Returns:

```
rwxr-xr-x   3 bengrunfeld  staff  102 Apr 23 23:24 .
drwxr-xr-x  4 bengrunfeld  staff  136 Apr 17 19:29 ..
-rw-r--r--  1 bengrunfeld  staff   41 Apr 23 23:24 master
```

To find where the HEAD is pointing, use

```
cat .git/refs/heads/master
```

This will return a SHA value with the current HEAD pointer.

If we use

```
git log --oneline
```

We will see that the most recent SHA is what HEAD is pointing at.

To Create A New Brach

We just use `git branch` and then type the name of the new branch.

```
git branch my_first_branch
```

There can't be any spaces, and don't use punctuation. Letters, numbers and underscores only.

Then you can check if its there with

```
git branch
```

And you can also check for it with

```
ls -la .git/refs/heads
```

At this point, before you've made any commits with the new branch, if you check where HEAD is pointing with

```
cat .git/refs/heads/my_first_branch
```

it will print out the same SHA value as before, because even though you've created a new branch, you haven't made any commits yet, so HEAD hasn't changed yet. So you have 2 branches pointing at the same commit.

So if you do

```
cat .git/HEAD
```

it will still point to the master branch, because you haven't made any commits yet. Master is still the currently checked out branch.

Switching Branches

To switch to another branch, use

```
git checkout new_branch_name
```

Git will then print out

```
Switched to branch 'new_branch_name'
```

You can then confirm that with

```
git branch
```

and

```
cat .git/HEAD
```

Creating a Branch and Switching to it at the Same Time

To create a branch and switch to it at the same time, use

```
git checkout -b
```

`-b` means create a new branch and switch to it at the same time. Kind of like "check this branch out and create it at the same time"

Branches should have useful meaningful names.

Nested Branches

If you create a new branch WHILE you are in another branch, it will create a branch of the branch you are in.

So if you are in the `new_javascript` branch and you create a branch called `new_jquery`, that will be a branch of `new_javascript`, NOT `master`.

This is where it's great to use

```
git log --oneline --graph --all --decorate
```

because it tells you how many branches there are, and where the HEAD is currently at. Very useful.

Switching Branches with Uncommitted Changes

Your working directory needs to be mostly clean before you can switch branches. Mostly clean means there can't be any conflicts between two branches. But if you have a file in one branch that does not exist in another branch and you try to switch, you'll be fine, and when you come back to the first branch, the other file will still be there.

If your working directory is not mostly clean, meaning there are files in both branches with the same file name, Git will not let you checkout the other branch. It will throw you a snooty error.

You have 3 options if Git throws you an error saying you have uncommitted changes.

1. You can remove the file by using `git checkout -- filname.txt`
2. You can add the file to the staging index and then commit it
3. You can stash the change, meaning you can save it until later

All 3 of these options will remove the snooty error Git has thrown you and will allow you to switch to another branch.

Comparing Branches

Before, we said we could use `git diff` on any tree-ish, and a branch is a tree-ish.

So to compare 2 different branches, we use:

```
git diff master..my_first_branch
```

Git will then tell me what are the differences between the tip of `master` and the tip of `my_first_banch`.

You can do this in either direction. i.e. parent -> child, or child -> parent.

Remember the nice feature of diff which is `--color-words`

```
git diff --color-words master..my_first_branch
```

Remember, we're not just passing in a branch, we're passing in a tree-ish, so we could see the difference between ancestors of branches by using:

```
git diff --color-words master..my_first_branch^
```

Checking if One Branch Completely Contains Another Branch

We can also compare branches to find out if one branch completely contains another branch.

I.e. whether everything in a certain branch has been merged into the current branch.

```
git branch --merged
```

That will show us all brances that are completely included within this branch. What this does is that it goes back up the ancestor chain of the current branch to see: does it have the tip of master in it. If it has the final commit of master, then it has all of the ancestors as well.

Renaming Branches

To rename a branch, use

```
git branch -m old_branch_title new_branch_title
git branch --move old_branch_title new_branch_title
```

It's not actually moving it, it's renaming it, but these are really the same thing. Git considers a rename to be a move, the same way that unix does.

Deleting Branches

To delete a branch, use

```
git branch -d branch_title
git branch --delete branch_title
```

Git has some checks in place to make sure you don't do something stupid.

1. You can't delete the branch that you're currently on. You need to be in another branch to delete it.
2. If you've made commits in the branch you're trying to delete, but haven't merged that branch another branch, Git will throw you a warning, and ask if you really want to delete it.

To delete that branch, even if it hasn't been merged into another branch, use

```
git branch -D branch_title
```

Using the capital `D` here tells Git that you're ready to take off the safety of using the lower-case `d`, and really destry `branch_title`.

Changing the Command Prompt to Show You The Branch

To configure the command like to show you what branch you're currently in.

Unix Prompts

To see what your current UNIX prompt is, use

```
echo $PS1
```

To change your UNIX prompt, use

```
export PS1='<whatever you want the prompt to be'
```

e.g.

```
export PS1='>>>>'
export PS1='benjy$ '
```

etc

To show our current branch in the command prompt, we use:

```
export PS1='$(__git_ps1 "(%s)") > '
```

`__git_ps1` is a function that will show you the current branch you're in

`%s` is the what of printing that current branch out

What KS recommends is using

```
export PS1='\W$(__git_ps1 "(%s)") > '
```

The `\W` will tell you what directory you're in.

Unfortunately, this `export` command is only active as long as the window is open. If you exit, then return to the terminal window, the prompt will have gone back to what it was before.

To keep this as a constant change, you need to edit your bash profile, or your bash rc file.

So....

```
cd ~
nano .bash_profile
```

Merging Branches

To merge changes made to another branch

1. Checkout the branch that we want to merge the changes into (the receiver)

2. Then use:

.

```
git merge <branch-name>
```

That's it. It will return `fast-forward` and then tell you that it has merged.

To check if it's worked, use `git diff master..<branch-name>` and it will be completely the same.

We can also use `git branch --merged` and it will show us that <brach-name> is full incorporated into master.

Because merges can get hairy if there are multiple changes, you always want to start with a clean working directory. You don't want to have any uncommitted changes in there.

That will give you a good clean workspace where you can work out problems with your merges.

Fast-forward Merges vs. Real Merges

If you haven't made any additional commits in the receving branch and then you merge in the new branch, this is called a fast-forward.

How The Fast-Forward Works

It goes into the branch that will be merged (the pitcher), starts at the most recent commit and then works its way backwards up the ancestor tree up to the branch the the pitching branch was created out of, and all the while checks whether it has the HEAD pointer of the current branch.

If the HEAD pointer is pointing at the commit where the branch was made, it just moves the branch commit into the main timeline, because there was no need to make a new commit.

Basically, a fast-forward simply places the latest commit of the branch you're merging (the pitcher) at the tip of the branch you're merging into (the receive), because there's no point to do it any other way.

Fast-Forward Methods

```
git merge --no-ff <branch-name-pitcher>
```

The `--no-ff` option forces Git to create a merge commit anyway. It says, "don't do a fast forwar. Make a real commit".

```
git merge --ff-only <branch-name-pitcher>
```

This says to Git, do a merge ONLY if you can do a fast-forward. If you can't do a fast-forward, then just abort.

A True Merge

Is when commits have been made to the receiving branch since the branch was created.

```
git merge <branch-name-pitcher>
```

Merge Conflicts

A conflict occurs when there are 2 changes to the same line or set of lines in 2 different commits because then Git can't decide which one to use or how to merge them together.

This is the biggest headache with working with branches.

If there is a conflict. The branch will say (master|MERGING). It will mean I'm in the middle of a merge.

it will say

```
both modified file.txt
```

It will tell you to add or rm on the problematic file.

If you go into the file with nano, you'll see the Git marks the first branch with

```
<<<<<<<< HEAD

    //block of old text

============

>>>>>>>> Branch-Name

    //block of new text

============
```

Resolving Merge Conflicts

There are 3 choices of how to resolve merge conflicts

1. Abort the merge
2. Resolve the conflicts manually (usually best option)
3. Use a merge tool

.

1. Abort the Merge

   git merge --abort

This will totally abort the merge and nothing will get merged.

1. Resolve the conflicts manually (usually best option)

Go into the file, make one copy of the text you want to keep, then delete the text underneath `<<<<<< HEAD` and get rid of all the Git markings.

Then do an `add` and a `commit`.

Git has a standard commit message for resolving conflicts, so instead of using `git commit -m "message"`, it's best to just use `git commit`.

1. Use a merge tool

To use a merge tool, type

```
git mergetool --tool=""
```

To find all the different options available

```
git mergetool
```

Strategies to Reduce Conflicts

- Keep lines short
- Keep commits small and focused
- Beware stray edits to whitespace - don't make unecessary changes to whitespace
- Try to merge often
- Track changes to master (as things change with master, merge them into your new branch, even multiple times over multiple commits, then eventaully you can merge new branch back into master)

The Stash

The Stash is a place where we can store changes temporarily without having to commit them to the repository. Much like putting something into a drawer to save it for later. The Stash is not part of the repository, staging index, or working directory. It's a special 4th area in Git, separate from the others, and the things that we put into it aren't commits, but they are a lot like commits. They work in a very similar way. They are still a snapshot of the changes that we were in the progress of making, just like a commit it. But they don't have a SHA associated with them.

It is used when you make a change to a file in a branch, and then attempt to checkout another branch without adding and then committing the changes. But you aren't ready to commit your changes, so you want to save them instead.

You can actually use stash anytime you want to take some stuff and shove it in a drawer.

The Stash Command

You stash something by using the following command:

```
git stash save "message goes here"
```

The save message is for you benefit. It's not something that anyone else is going to see.

What it does after it puts those changes into the Stash is that is runs `git reset --hard HEAD`, which takes what's in our repo and puts it into our staging index and our working directory, so they are the exact same as where the HEAD is pointing at the moment.

Taking a Look At What's In The Stash

To see a list of what's in the Stash, we use:

```
git stash list
```

What will come back will be something like:

```
stash@{0}: On <branch-name>: message goes here
```

This is the way to refer to this item in the Stash. e.g. `stash@{0}`.

The Stash will be accessible even when we switch branches. So if we realize that we're on the wrong branch and don't want to make a commit there, we can stash the changes we want, then change branches and then pull the changes out of the stash and commit them to the proper branch.

We can see more information about each of those stashes with:

```
git stash show stash@{0}
```

It will then bring up a diff stat, which is a stat about what changed in this file.

If we want to see more information, we can use the `-p` option, which will show it to us as a patch.

A patch is a section of code that you can apply to different things to modify them and change them.

This basically says, show us the sets of changes. It brings up the typical `diff` that we're used to seeing which shows us the sets of changes. This is very similar to `show` ing a commit.

Retrieving Changes From The Stash

To take changes out of the change and into the working directory, whatever branch we're in.

Like with merges, there might be conflicts, and like merges, you can resolve them in a similar way.

```
git stash pop
git stash apply
```

Both of these commands will pull what is in the Stash out and put it in the working directory.

The difference between them is the `stash pop` also removes it from the Stash as well.

`git stash apply` leaves the change in the Stash.

The idea of using `apply` is that if you want to use the same change to many different branches, you can keep it in the stash and then go from branch to branch and apply that change.

Most of the time, you're going to use `pop`.

After `pop`, you need to specify which stash you want to pull out. If we don't say, then by default it's going to pull the first one. i.e. `stash@{0}`.

Deleting Items From Within the Stash

To delete an item from within the Stash without committing it, use

```
git stash drop stash@{0}
```

To check that it's gone, use

```
git stash list
```

To delete EVERYTHING in the the Stash (really deletes everything!!):

```
git stash clear
```

Using Remote Reposities (remotes)

Until now, we've done everything on our local computer.

Once we put the changes that we've made on a remote server, other people can download them, make changes of their own, and upload them up to the remote server, which we can then download and work with.

A remote server simply runs another copy of Git. The only difference is that the remote server runs a copy of Git software that allows it to communicate with our client at the same time.

But the repo where those commits are stored are simply running Git.

Uploading Our Changes to a Remote Server (a PUSH)

A `push` is when you take your commits and put them on a remote server so other people can see them. The process is called a push. "You pushed them up to the remote server."

At that point, the remote server creates the same branch, with the same commits with the exact same commit ID's pointing to them.

Git them makes another branch on our local computer that is typically called origin/<branch-name>.

That's by convention, but we can change that name if we really want.

`origin/<branch-name` is a branch on our local machine that references the remote server branch, and it always tries to stay in sync with that.

When we make more commits to our master branch, or whatever branch we're working on, we then make another `push` and this pushes our changes up to the remote server.

The origin/master branch then updates as well to mirror the remote server.

When other people make changes to the remote server and contribute them there, we need to pull those changes down so that we know about them. The way that we do that is called a `fetch`. So we `fetch` the changes. At that point they come into our origin/<branch-name> branch, because what we're doing is keeping those in sync.

Fetch is essentially saying, sync up my origin/<branch-name> with the remote server version. But it does NOT bring it into our master branch. Now our computer knows about the change and we have it locally, but it won't be in our master branch until we do a merge. At that point, it will be brought into our master branch and we'll be back in sync.

The HEAD Pointer With Push

When we make a push, it creates the commit on the remote server and adds a pointer that points to the most recent commit on the remote server, THEN adds ANOTHER pointer locally on our machine that points to whatever HEAD is now pointing to locally.

But when we do a fetch and the commit is stored in the origin/master branch, the origin pointer will move to the new commit, but the HEAD pointer from the regular master branch will NOT MOVE! It will only move this new commit if we merge origin/master with master.

Usual Workflow Working with a Remote Server

When you're working with a remote server, the workflow you'll usually go through is:

1. Do your commits locally
2. Then you'll fetch the latest from the remote server
3. Get your origin branch in sync
4. Then merge any of the new work you did into what just came down from the server
5. Then push the result back up to the server

Setting up a Git Account

GitHub is a Git host. It has great bells and whistles like being able to graph commits made.

To Create a New Repo on GitHub

1. Click the "create new repo" icon on the top right hand corner of the screen.
2. You can add a readme file or a .gitignore file depending on what language you're programming in.

Then to sync the GitHub repo with your local repo, you need to grab the address that's shown in the HTTP address bar at the top of the page. i.e. NOT the SSH address.

To Sync Our Local Repo with the GitHub Repo

1. Go into the main directory of your GitHub project on your local machine.

Then you can use

```
git remote
```

This will list all the remotes that Git knows about. If it doesn't know about any, it will return blank.

```
git remote add <alias> <url>
```

This will add a remote, using as the alias of what we will name our remote, plus the URL of where Git can find it.

You can copy and paste this from GitHub. GitHub suggests using the alias `origin`, although you can change that if you want.

This will put in a new remote called `origin` that points to that remote server at that URL.

We don't have to call our remote `origin`, but by convention, you call your primary `origin`, but you can call it `github` or anything else for that matter.

You CAN have more than 1 remote for your project. If you have several different remotes for your project, you'll definitely want to give each of them a different name.

```
git remote -v
```

Will list the remotes that we know about with a bit more info. It will show us the remotes with the URL that it will use for fetching and the URL that it will use for pushing.

Typically these are the same, but they don't have to be.

It stores these in .git/config

To remove a remote:

```
git remote rm <alias>
e.g.
git remote rm origin
```

This will simply remove a remote.

Cloning a Remote Repository onto your Local Machine

This assumes that we don't have a local copy to start with. This circumstance would come up if someone said - "come and collaborate on a project with me. Grab my code from GitHub". Or an Open-Source project.

Regardless of how you get to the project, what you need to know is how to locate it. GitHub tells you that information in the HTTP/SSH address bar.

```
git clone https://github.com/kevinskoglund/explore_california.
git
```

This says, make me a local copy of this repository. Clone it on my machine.

It will create a new directory, and the name of that directory will be `explore_california`.

If you already have a directory of the same name, Git will throw you a fatal error and abort.

To clone the remote repository and give it a different directory name, use

```
git clone https://github.com/kevinskoglund/explore_california.
git my_version
```

It only brings down the master branch by default. You can change this though.

Tracking Remote Branches

When you have a local branch that stays closely in sync with another branch, it's called **tracking**. The idea is that we want to keep the 2 closely in sync.

Tracking is really common with remote branches and works in a similar way.

With a little bit of configuration, we get to save ourselves a bit of typing, by letting the master branch know what remote branch should it be using when it's doing its fetch and its push. We won't have to specify each and every time. It will have a default setting by knowing its tracking branch.

Using Git Push

When we do `push`es, what we're pushing is a branch. So we're on a branch, and we're telling GitHub, push this branch up to the corresponding branch on the remote server. To do this, we use:

```
git push -u origin master
```

Theoretically, it's:

```
git push -u <alias> <branch-name>
```

It counts up the objects we have, compresses them, so we can send a small packet of data, then it writes those objects on the other side. Once it's done with all of those, it creates a new branch for us.

Then it notes:

```
Branch master set up to track remote branch master from origin
```

That's what the -u option does for us.

Where are remotes stored

After we've used the `-u` option on a push, we can find the remotes with an attached branch with:

```
ls -la .git/refs/remotes
```

This is where the remotes are stored. If you look inside remotes with

```
ls -la .git/refs/remotes/origin
```

You'll see a directory called master.

And if we look inside of master, we'll find a SHA. Use:

```
cat .git/refs/remotes/origin/master
```

Seeing the Remote Braches

To see your regular branches, we use

```
git branch
```

But if we want to see our remote branches, we use

```
git branch -r
```

To show you both your local branches and your remote branches, use:

```
git branch -a
```

Git Push -u

You're pretty much going to always want to use the `-u` option.

If we don't use

```
git push -u
```

it does not track any remote branch. All it does is push our code up there. And that's it. It doesn't keep any kind of reference that this is the branch we'll be working with in the future.

The `-u` option says, push our code up there, but make a note of this branch, because we'll be using it frequently.

When we use `git clone`, it does note the branch.

If you use

```
cat .git/config
```

we'll be able to see that our branch, master, is set up to track the ref/heads/master that is on origin. That's how you know its a tracking branch.

If you have a branch that's not tracking, and you want to make it tracking, then you have 3 choices:

1. Add the listing manually in `.git/config`

2. Use

.

```
git config branch.<branch-name>.remote origin
git config branch.<branch-name>.merge refs/heads/master
```

1. Use

.

```
git branch --set-upstream <branch-name> origin/<branch-name>
```

`set-upstream` is short for `-u` with the push command.

Pushing Changes to a Remote Repository

To check the log of the alias branch (our local copy of the remote branch), use:

```
git log --oneline origin/master
```

You can compare with

```
git diff origin/master..master
```

To push changes, use:

```
git push origin master
```

But because we set it up before as a tracking branch with `-u`, we only need to use

```
git push
```

This also works with `git fetch`.

Fetching Changes from a Remote Repository

`Fetch` goes to GitHub and grabs the lastest copy of the repository.

```
git fetch origin
```

If we only have 1 remote repository, we can just use

```
git fetch
```

A fetch DOES NOT bring files into your working directory. It brings them into your alias branch (e.g. origin/master). In order to see those changes, you need to merge origin/master with another branch, like master.

Best Practice re Fetch:

1. Always fetch before you work (first thing in the morning)
2. Fetch before you push (before you make a push, check what's there)
3. Fetch often (it's not destructure, so do it often)

Merging Fetched Changes

Origin/Master is a branch like any other branch. The only difference between it and other regular branches is that we can't check it out, because Git needs to keep it in sync with what's on the remote server.

To bring files in from origin/master, we use:

```
git merge origin/master
```

Make sure you do a fetch first, because you're only merging from your local copy of the remote branch.

We can also use a shortcut which is

```
git pull
```

`git pull` is equal to `git fetch` + `git merge`

It does it all in one step.

Checking Out Remote Branches.

Remote branches are exactly like regular branches, with one exception: we can't check them out.

We can create a branch, and then specify where we want the information to come from. Usually this would be HEAD (default), but we can specify a commit or another branch.

```
git branch some_name origin/some_name
```

Git will then issue us a message that it is tracking a remote branch from origin.

Or, we can use:

```
git checkout -b some_name origin/some_name
```

And this will create the branch and grab the info from the target and issue the same message as above.

Pushing to an Updated Remote Server or Branch

Git NEVER tries to do a merge during a push. Instead Git says, some new stuff has come in and I'm not sure what to do about it, so you need to fetch the changes that are on the remote server, sort it out on your end, and come back and try again.

So we do a fetch, then we have to merge our changes in with origin/master, and then we can push again, and the remote server would accept it.

If you can't push to the remote server, you'll need to fetch, then merge, then push again.

Deleting a Remote Branch

We'll be telling GitHub that it should erase one of the branches in its repo. There are 2 ways to do this. Here is the old way:

```
git push origin :some_name
```

The colon will have the effect of deleting that branch name.

If you now do

```
git branch -r
```

you'll see that it's gone.

Remember that if you have this as a local branch, THAT won't be deleted, because you've deleted the remote branch, not the local one.

The newer way to delete a branch is:

```
git push origin --delete some_branch_name
```

Enabling Collaborators on Your Project

This also covers how you can become a collaborator on an open source project.

To enable other collaborators on your project, go to the project page and click Settings from the top menu, then type the GitHub username of the person we want to collaborate with us.

Then just click add.

Once you add them to the project, GitHub will send them an email with the project name and it's URL.

If you want to make an open source project, it works a little bit differently. Not everyone in the world can make commits to the project itself. It would be a total free-for-all if they could.

Instead, a limited number of people have WRITE access, but everyone has READ access.

Instead, the way that you need to make changes is by making a FORK. Before you make a fork, you need to decide what changes you want to make. Look at the network and make sure no one else is already working on that change. Look at issues to make sure no one has talked about this problem. Then post an issue there to make sure other people know you're working on it.

Then click FORK, which makes your own version of the project on your own GitHub repository. This version is now no longer part of the main version, and this one you WILL have WRITE access to. Then clone the repository, make your changes to it locally, just like you usually would, commit those changes up to your version of the project, and then when you're ready, go back to the GitHub page for the main project and issue a PULL request.

A PULL request is like raising your hand and saying "I've got something here that I want to show you." You submit a message with your request, so you identify what the problem was that you saw, or what feature you decided you wanted to add, talk about how you want to do it and why you think it's good for the project, and if you make the case effectively and your code looks good, then they'll accept your changes, and incorporate them into the main project. They'll grab your branch and merge it in.

And then everyone will have access to your new feature. That's how you enable collaboration on a project using Git.

Setting Up Aliases (Keyboard Shortcuts) For Common Commands

Aliases can be dangerous, especially for beginners, because you need to focus initially on using the actual commands.

We can set up an Alias (aka Keyboard Shortcut) in any Git config file, but it makes the most sense to have it in our user global config file.

There are two ways to enable keyboard shortcuts.

1: We can go in and edit the file manually

2: We can use

```
git config --global alias.<keyboard shortcut> <command>
```

e.g.

```
git config --global alias.st status
```

This says, if you type `st`, it will consider it as if you wrote `status`.

If the command has a space in it, you need to wrap it in double quotes ("").

You can now check out the config file with `cat .git/config`. You can go into the file and copy the format, and create new shortcuts.

These aliases have become standard with Git users.

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.br branch
git config --global alias.df diff
git config --global alias.dfs "diff --staged"
```

You can still use options with these. E.g.

```
git br -r
```

And for Kevin's SUPER command:

```
git config --global alias.logg "log --graph --decorate --oneli
ne --abbrev-commit --all"
```

Using SSH for Remote Login

Every time we did a `git fetch` or a `git push`, we had to log in with our credentials. This gets old fast.

There are 2 ways to get around this:

1. Have a keychain program that will store your username and password and then Git will be able to go to the keychain program, get your username and pw and send it to the remote. Git has a help page on how to use this called "password caching"

2. Use SSH keys. We have a bit of code and we put it up on the GitHub server. Then when we make a request, Git automatically sends a bit of code along with the request and uses those to auth me. GitHub has a help page on "setting up SSH Keys".

Using GUI's

Some people want to use GUI's after they've mastered the command line.

- GitWeb - free
- GitX

END OF FILE

Almost