Git Review

Follow along at https://git-scm.com/docs/gittutorial

Setup

```
$ git config --global user.name "Ashton Snelgrove"

$ git config --global user.email ashton.snelgrove@utah.edu
```

Initialize a repository

```
$ git init
```

Git will reply

```
Initialized empty Git repository in .git/
```

You’ve now initialized the working directory—you may notice a new directory created, named `.git (ls -a)`

Stage and commit

Add a file

```
$ echo "hello world!" > README
```

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with git add:

```
$ git add .
```

This snapshot is now stored in a temporary staging area which Git calls the "index". You can permanently store the contents of the index in the repository with git commit:

```
$ git commit
```

This will prompt you for a commit message.

Stage and compare

Modify some files, then add their updated contents to the index:

```
$ git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using git diff with the --cached option:

```
$ git diff --cached
```

(Without --cached, git diff will show you any changes that you’ve made but not yet added to the index.)

Status

You can also get a brief summary of the situation with git status:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1
    modified:   file2
    modified:   file3
```

If you need to make any further adjustments, do so now, and then add any newly modified content to the index.

Viewing project history

At any point you can view the history of your changes using

```
$ git log
```

If you also want to see complete diffs at each step, use

```
$ git log -p
```

Often the overview of the change is useful to get a feel of each step

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

Sensible commits

Commits should make small changes.

Commits should have a message that reflects the changes.

You should commit regularly.

You can always squash the commits down later.

Managing branches

A single Git repository can maintain multiple branches of development. To create a new branch named `experimental`, use

```
$ git branch experimental
```

If you now run

```
$ git branch
```

you’ll get a list of all existing branches:

```
 experimental

* master
```

The `experimental` branch is the one you just created, and the `master` branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on.

Switching active branches

```
$ git switch experimental
```

to switch to the `experimental` branch. Now edit a file, commit the change, and switch back to the `master` branch:

```
$ git commit -a
```

```
$ git switch master
```

Check that the change you made is no longer visible, since it was made on the `experimental` branch and you're back on the `master` branch.

Diverging branches

You can make a different change on the `master` branch:

```
(edit file)
```

```
$ git commit -a
```

at this point the two branches have diverged, with different changes made in each.

Merging

To merge the changes made in `experimental` into `master`, run

```
$ git merge experimental
```

Remember, your active branch is master. If the changes don’t conflict, you’re done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
$ git diff
```

will show this. Once you’ve edited the files to resolve the conflicts, add them and commit.

Remotes

Bob wants to work with Alice, so he clones her repository.

```
bob$ git clone /home/alice/project myrepo
```

Git is distributed, so Bob has **his own complete copy** of the repository.

```
bob$ git remote -v
```

```
origin /home/alice/project
```

A remote is a reference to a different copy of the repository.

Alice can add her own remote

```
alice$ git remote add bob /home/bob/myrepo
```

Pull and fetch.

Bob has made some changes, and Alice wants them.

```
alice$ git pull /home/bob/myrepo master
```

Her repository now contains the commits in branch master. A pull will automatically try and merge the commits in.

If instead, she only wants the commits but not update her branch, she can fetch.

```
alice$ git fetch /home/bob/myrepo master

alice$ git branch

bob/master

master
```

Push

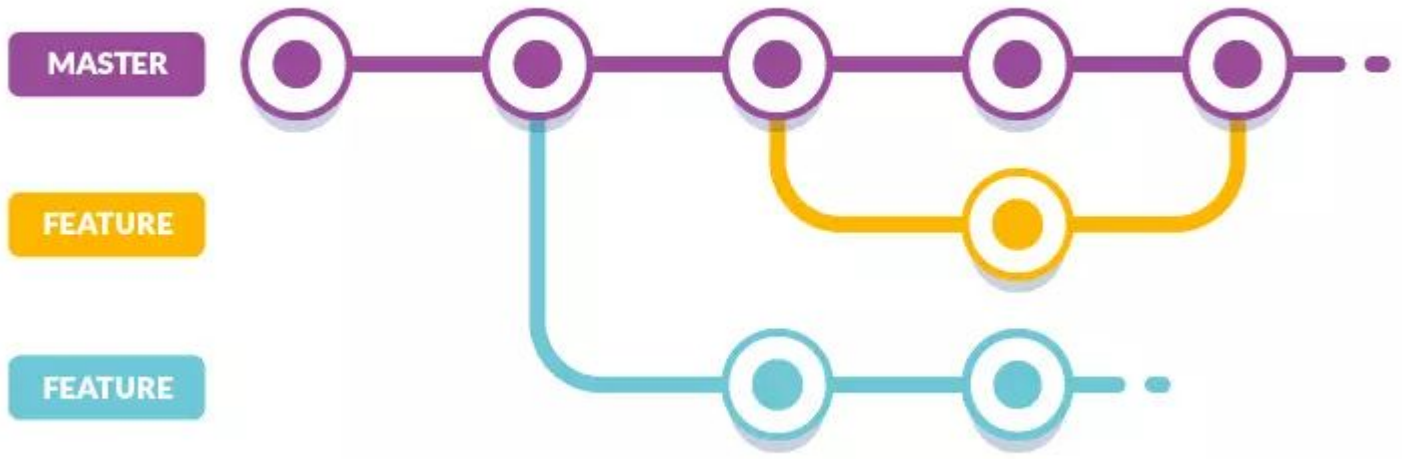
Bob could also push his changes to Alice.

```
bob$ git push origin master
```

He needs to be careful, because he'll need permission, and if Alice is working on master, it won't work!

If we have a third copy of the repository, stored someplace like github, we can both push and pull to that copy without stepping on each others toes.

Branching workflows



Feature branches

The working version of the code is a branch usually named "master" or "main".

You will create a new branch when working on a new feature or bug fix.

Give it a sensible name to match what it is changing. e.g. bugfix-segfault-in-uart

Do your work on the branch.

When the work is ready, you will merge your changes back into the master branch.

Pull request

When your feature is finished, you will make a request to pull your branch into the main branch.

This is called a pull request (PR) or merge request (MR). These are for all practical purposes synonymous.

Whoever is responsible for maintaining the main branch will review the PR, and if it is acceptable, pull the code into the main branch.

Short lived branches.

Branches should be as short lived as possible. The main branch is always changing, and a long lived branch is more likely to diverge from the main branch. Merging a major development branch into a master that diverged a year prior is a nightmare.

Feature branches should cover one change. This helps the PR reviewer understand what is changing and why without having to wade through multiple changes.

For example, if you wanted to do a whitespace or formatting cleanup, do it as its own PR.