Local Git Repo - #2 Branching

#git-workshop

Before talking about branches, let's take a closer look at the log.

Log

```
$ git log
commit ec6d394c51d18f4c54022dac257d3f53bc99470f (HEAD -> master)
Author: Demo <demo@berkeley.edu>
Date:   Fri Jan 12 16:32:46 2018 -0800

    Add Trebek pangram

commit 1c2db6becaed138ee21495b3ef16cd3eca2d7a05
Author: Demo <demo@berkeley.edu>
Date:   Thu Jan 11 22:34:19 2018 -0800

    Add dog pangram

commit f9e7d7b92637397d11e9dea5f0bef73d78058e5b
Author: Demo <demo@berkeley.edu>
Date:   Thu Jan 11 22:33:32 2018 -0800

    Add empty pangrams

commit 331d681bde3a71fcaaee133e18c6f010431dfd58
Author: Demo <demo@berkeley.edu>
Date:   Thu Jan 11 22:33:23 2018 -0800

    Add README.md
```

Each commit has a bunch of metadata, including your name and email, the date of creation, and the commit message you specified. Each commit also has a **commit id**, indicated next to the word *commit* on the first line of each commit.

By the way, commits can take up a lot of vertical space. A more compact representation can be obtained using the `--oneline` option:

```
$ git log --oneline
ec6d394 (HEAD -> master) Add Trebek pangram
1c2db6b Add dog pangram
f9e7d7b Add empty pangrams
331d681 Add README.md
```

This truncates the commit id's, but we only need a prefix.

In both cases, the current commit is indicated by (HEAD->master). HEAD represents the current location. "master" is the name of the current branch.

Branch

By default, we start on the master branch. To list all branches, use:

```
$ git branch
* master
```

You can see that there's only one branch, which makes sense.

To create a new branch, use `git branch` with a branch name.

```
$ git branch quack
```

We can check the current branch either using `git status` or `git branch` (the current branch is shown by the asterisk).

```
$ git branch
* master
  quack
```

We'll make a commit on master and a commit on the new branch. Committing on the master branch proceeds as we have done before:

```
$ echo "Quick fox jumps nightly above wizard" >> pangram1.txt # Note the two
'>'. This will append to the file
```

```
$ git add pangram1.txt
$ git commit
```

To commit on the new branch, we have to change to it first. We do this by doing `git checkout`. Then, we commit "as normal".

```
$ git checkout quack
$ echo "The jay, pig, fox, zebra, and my wolves quack" >> pangram1.txt
$ echo "Five quacking zephyrs jolt my wax bed" > pangram3.txt
$ git add *.txt
$ git commit
```

However, examining the log will reveal that we can't see the new commit on master from this branch, and vice versa! We have successfully split off onto two branching paths.

We can visualize this with `git log` as well:

```
$ git log --graph --all --oneline
* 19bccb7 (HEAD -> quack) Quacking
| * b676eb2 (master) Add more fox
|/
* ec6d394 Add Trebek pangram
* 1c2db6b Add dog pangram
* f9e7d7b Add empty pangrams
* 331d681 Add README.md
```

Merge

We've done some good work, but when all is said and done, we need to combine our work.

The way merge usually works is we start on a desired branch, and merge in changes from another branch.

```
$ git checkout master
$ git merge quack
Auto-merging pangram1.txt
CONFLICT (content): Merge conflict in pangram1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

A couple of things will happen here.

- `pangram3.txt` was added in `quack`, so it will be included in `master` when we merge. Nothing else needs to be done for this file.
- `pangram1.txt` was modified in `quack` and ALSO in `master`. We'll need to resolve the **merge conflict**.

Merge Conflicts

These are generally resolved using a text editor. You'll see something like this:

```
<<<<<<< HEAD
Quick fox jumps nightly above wizard
=======
The jay, pig, fox, zebra, and my wolves quack
>>>>>>> quack
```

Everything between `< HEAD` and `=` is the stuff in the current branch that conflicts, and vice versa for `> quack`.

Not every merge will be solved in the same way. In this case, we want to keep the changes from both branches. But in some cases, you might want to discard one or the other.

When we're done, we add and commit. Git will provide an automated commit message.

```
$ git add *.txt
$ git commit
```

Finally, we're done with the `quack` branch and we can delete it to clean up.

```
$ git branch -d quack
```

One last note

Don’t panic! Almost everything in Git can be undone, and merges are no exception. If you weren’t expecting a merge conflict, there’s no harm in backing out with `git merge --abort`.