Scientific and Technical Computing
Git Basics

Version 2013-1 as of 9 September 2013

Git Homework, BACKGROUND

Reading: The first three chapters of Git are great. (Go to git-scm.com → Documentation)
Read Getting Started (1.3 git-scm.com/book/en/Getting-Started-Git-Basics)
Read Git-Branching (3.1 git-scm.com/book/en/Git-Branching-What-a-Branch-Is)

Information summary and simple experiments:

When Git initializes a repository (git init) the default name for the (only) branch is, master. It is a moveable pointer that moves along as you do commits, pointing to the present commit (version). HEAD is the name of a pointer to the present branch pointer (that is, it points to the latest commit pointer of whatever branch you are working in). See Figure 1 below. HEAD is useful when switching between branches that are evolving simultaneous, again: the HEAD pointer points to the latest commit in the branch you are working in. We will see how to change branches blow, it is easy!

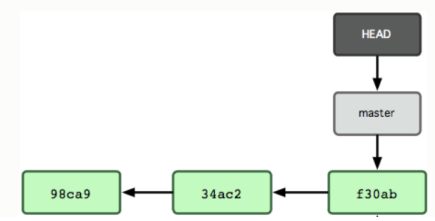

Figure 1. master and HEAD pointers pointing to latest commit.
[Image credit: http://git-scm.com]*

Branching:

A new branch is created (git branch <branchname>) first, and then it is given its own version of the files by a checkout command (git checkout <branchname>). The first step creates a new pointer to the present tree, the second step moves the HEAD pointer (working tree pointer) to <branchname>, and resets your working directory to look like the snapshot of the commit of the branch you checked out (in this case they are the same snapshot).

You can switch back and forth, committing SEPARATE changes in each branch simply by using checkout <branchname>. So, if you make a branch called testing, you can switch back and forth between your original working tree (default name is "master") like this:

```
$ git branch testing    #this just makes the branch, nothing more
$ git checkout testing  #this changes HEAD pointer to b1 and makes
                        #working tree (pwd) contain content of testing
$ …                     #make changes, and commit
$ git checkout master   #reverts files to master tree  (HEAD points to
master)
$ …                     #make changes, and commit
$ git checkout testing  #now working tree (pwd) contains testing contents
```

Figure 2 show what the repository looks like after the above commands.

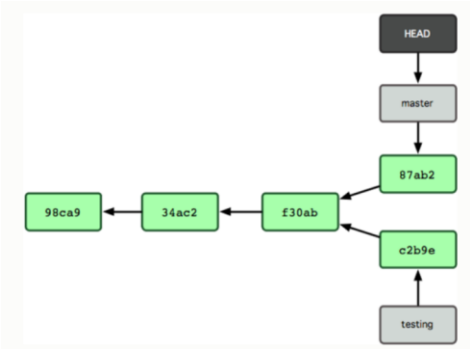

Figure 2. master and testing branches after branch & 1 commit each. Branch pointer (HEAD) shows the default branch (master) as the working tree branch in use.

What can go wrong? Before you checkout a new branch you should always clean up your working directory by adding and committing any modified files (or if you only have staged files from an add without a commit, commit them). Otherwise Git won't let you switch branches.

Forget which branch you are in. Use `git branch` to see the branches for the repo; the starred one is the branch you have checked out. You can also use `git status` and see the name provided on the "# On branch" line.

```
$git branch
*testing
  master
$git status
#On branch master
nothing to commit, working directory clean
```

There are other ways:

```
$ git --symbolic-ref HEAD
$ git rev-parse symbolic-ref HEAD #gets the blob (hash #)
```

Typically, you merge into the branch you are working in (have checked out). So, to merge the testing branch into master, check out the master branch and merge testing into master as shown here:

```
$ git checkout master
$ git merge testing
```

In this case a three-way merge which includes a common ancestor is formed by the merge command automatically creating a new snapshot (commit) as show in Figure 3 below.

Remove the testing branch with `git branch` when you are finished with it, using the -d (delete) option and the branch name. Here is how to remove the testing branch:

```
$ git branch –d testing
```

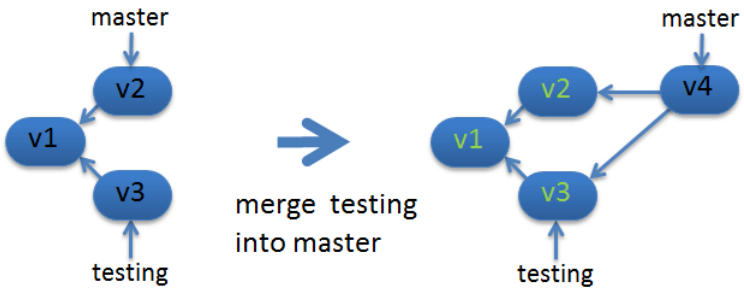

Figure 3. Merging testing into master uses 3 parents and creates a new snapshot.

If you have changes in each branch that have modified the same parts of files, the automerge will not complete, and you will be asked to fix conflicts and perform a commit to complete the merge. This can be painful, bit Git makes it relatively simple, and there are tools to help you sort out the differences. You can see the files with `git status`; the conflicts are MARKED in the file like this:

```
$ git merge testing
CONFLICT (content): Merge conflict in prog.c
Automatic merge failed: fix conflicts and then commit the result.

Conflicts in prog.c:
<<<<<HEAD
   Things here are in master
===========
   Things here are in testing
>>>>>>> testing
```

You can use an assortment of tools to see the differences and merge them. Execute "git mergetool" to see available diff tools. Forget which files are conflicted (unmerged), use the `git ls-file` command with the –u (unmerged) option? You'll get 3 in the list, one for each of the 3 branches that are involved (green versions in Fig. 3).

```
$git ls-files –u  # or try git diff or  git status or git ls-files –u | awk '{print $4}' | sort |uniq
 ### 1 prog.c
 ### 2 prog.c
 ### 3 prog.c
```

In the difference file keep either section, or combine them; no matter however you fix it, just remove the chevrons and === lines to designate that it has been fixed. Stage the file (`git add prog.c`, this marks it resolved) and commit it (e.g. `git commit -m "fixed conflicts"`). You are working in the master (of the merged branches), there is no need to do anything else, except remove the testing branch pointer:

```
$ git branch -d testing.
```

A remote branch is a reference to the state of a branch on a remote repository. This reference has the form <remote>/<branch>. When you clone from the remote repository, your local repository has a copy of the remote master, named original/master (and you cannot move it!). It will look something like this:

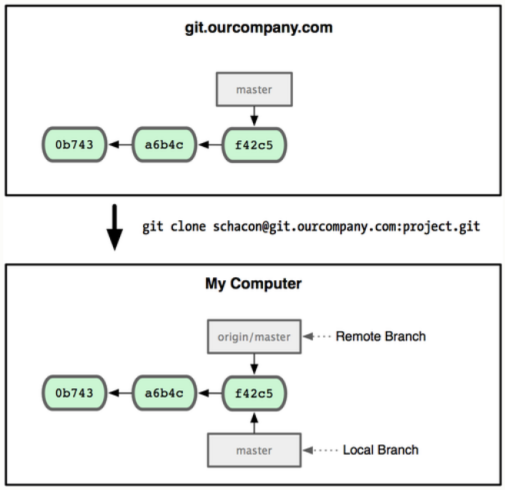

Figure 4. Cloning a remote repository.

If somebody else pushes commits to the remote repository that master will move forward, and your local remote branch will not change until you synchronize. You use fetch to synchronize YOUR LOCAL repository:

```
$ git fetch original
```

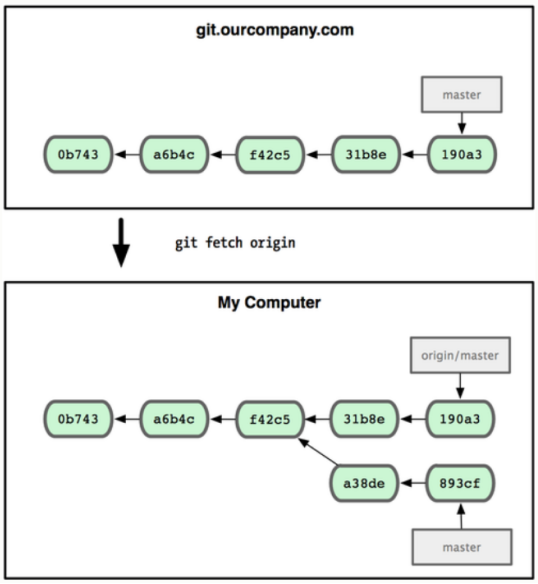

Figure 5. Synchronizing with remote repository (after another user has committed updates).

You can also do the same thing for another (cloned) repository with git remote add—but that is a bit more complicated and we won't discuss that here.

You may be the only one that has write access to your repository, and others may just read it, to get the latest package version. You can push your changes to the remote system (where others can only access it—this is setup through an access control list at the repository). The syntax is:

```
$ git push [repository] [branch]
```

"[ ]" means optional. Here, `repository` is the name of the remote site (required if you can have several repositories you have added to your remote list), and `branch` is the branch in common (that others may be working on). The branch name actually gets expanded to a refspec that looks like this refs/heads/testing:refs/heads/testing, it means take my testing and make it the remote's testing. If anybody else wants a copy of the new branch, they can acquire the branch pointer information with a fetch (pointer cannot be modified), and to merge the testing into their local repository they must run `git merge origin/testing`. But if you want your own testing branch to work on you "base it off your remote branch" with a checkout:

```
git checkout –b testing origin/testing # create testing branch;
                                       # copy server version(origin/testing1 into it.)
```

The checkout command (used to get a remote directory) automatically creates a "tracking branch" which automatically detects which local and remote branch to use when executing a push and pull command. `git push` knows which server and branch to push to. `git pull` knows the local and remote reference and then automatically merges in the corresponding remote branch. Sweet, no arguments!

To delete a remote branch named testing1, use the command:
`git push origin :testing1`
(from the syntax git push <remote> <local_branch>:<remote_branch> the meaning is: put nothing in the remote branch.)

Getting started with remote repositories:

You have already created the STC repository at bitbucket.org. You will be asked to push your local git repository to bitbucket, declaring it to be a remote repository and the origin.

Your remote git directory is called STC.git. The .git extension has special meaning. It is a "bare" repository-- it does not include a working directory of files, but only a .git directory. You will tell your local directory that for your local git repository (directory you are in with a .git directory) you want to create a copy remotely at a site with an origin/master pointer (instead of master). The remote connection information is added to a list with the `git remote add` command. Next you will push up your repository to bitbucket. Here are the commands:

```
$ cd <my local git directory>
$ git remote add origin ssh://git@bitbucket.org/<bb_username>/stc.git
$ git push -u origin --all
```

Use your bitbucket login name for <bb_username>, that should be FirstnameLastname (as we asked you to use when creating an account on the first day of class). We don't use tags, but for those that do, you will have to push them too, by executing push -u origin --tags.

The push report should look something like this:

```
$ git push -u origin --all
Counting objects: 36, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (28/28), done.
Writing objects: 100% (36/36), 3.03 KiB | 0 bytes/s,
done.
Total 36 (delta 11), reused 0 (delta 0)
To ssh://git@bitbucket.org/milfeld/STC2.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

But you may get the following error:

```
$ git push -u origin --all
Warning: Permanently added the RSA host key for IP address
'131.103.20.168' to the list of known hosts.
Permission denied (publickey).
fatal: Could not read from remote repository.
```

In this case you will need to create an SSH keypair (private and public) and include your public key in the bitbucket repository. To create the pair, execute the following command and respond with a return for the passphrase (for passphraseless access).

```
$ ssh-keygen  -b  1024  -f   $HOME/.ssh/rsa_id_bb
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

This will create a key pair rsa_id_bb and rsa_id_bb.pub in $HOME/.ssh. They are your private and public keys, respectively. cd to $HOME/.ssh and cat the file.

```
$ cd $HOME/.ssh
$ cat rsa_id_bb.pub
```

Copy the contents of rsa_id_bb.pub into your copy/paste buffer and paste it into your bitbucket "add key" window, as described next.

In a browser, go to bitbucket.org and log in. Click on the avatar (the silhouette of a bust) in the upper right. Select "Manage account". In the new window select SSH keys in the left navigator area. Click on the "Add key" button, and paste the contents of your rsa_id_bb.pub key in the "Key*" text box area. (Put a convenient name in the Label text area, as a reminder where it came from.) Select "Add key". You should be set up. You can copy your private key (rsa_id_bb) to $HOME/.ssh on any machine that you want to have a local repository and push from. Test your setup again with:

```
$ ssh -T git@bitbucket.org
```

If this fails (generally for macs), execute:

```
ssh-add $HOME/.ssh/rsa_id_bb
```

See https://confluence.atlassian.com/pages/viewpage.action?pageId=270827678 about starting an ssh-agent if ssh-add does not work.