Git tutorial

There are many different versions of source control. I'll give you a tutorial for using git, a distributed and popular protocol developed in 2005. This tutorial is meant for a one-person project. There are many more complicated problems with source control when you have multiple people working on the same project. I will mention some of them as we go along.

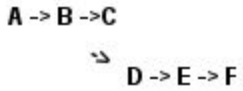
Git Concepts

Commits

The basic concept in git is a commit. A commit in git is a single, atomic, set of changes to one or more files. Git stores changes to individual files in the form of what is called a 'diff'. Each diff in a commit describes the changes between that file before the commit, and that file after the commit. Every commit is uniquely identified by a string of characters called a hash. This hash is globally unique to each commit within a repository. As you make changes to your files, you will add them to what is called the 'staging area' using 'git add'. The staging areas is where git stores changes to files that are about to be committed permanently. When you run the 'git commit' command the changes you added to the staging area are combined into a commit and saved to your local repository.

Branches

Git organizes individual commits into what are termed branches. Each branch is a sequence of one or more commits. For example, three commits, A, B, and C could form a branch like A -> B -> C. Every git repository has at least one branch, including the primary branch known as 'master'. It is possible to create new branches that split off of existing branches. For example, you can have a branch called 'poker' that splits off of your 'master' branch. In terms of commits, this could look like



Where A, B, and C represent commits on the 'master' branch and D, E, and F are commits on the new 'poker' branch. As you are getting used to basic git commands you should stick with a single branch, but later on you can use branches to work on different features of your code in isolation.

Repositories

Commits and branches are part of repositories. Git is what is termed a distributed version control system. What this means is that in larger projects, every developer working on the project has a local repository to which they actually make commits to. Then when a developer has finished their work locally, they can push their changes to a remote repository for everyone else to use. It's important to understand that when you are working with git, the commands you use only affect the state of your local repository, until you execute a 'git push' command (which you'll learn about later). The local repository is stored only on your machine, and no one else can see the changes you make to your local repository until you 'push' the changes to a remote repository (More on this later). This distributed design allows developers to rapidly make changes to their local repository and not worry about breaking the code that is stored in a global remote repository that other developers are using.

The remaining sections describe the most common and useful git commands and how you would use them as part of a version control workflow.

Git Init

The command you will be using is called “git” and it is installed in every lab machine at Purdue. First you’ll want to initialize your local repository. This can be done with the command:

git init

```
abravo@data:~/cs240/blackjack/solution$ git init
Initialized empty Git repository in /home/u89/abravo/cs240/blackjack/solution/.git/
abravo@data:~/cs240/blackjack/solution$
```

Be smart about where you run this command. Only do it once and do it in a directory where every subdirectory can be added to your repository. For the blackjack project, I would suggest you run it in the directory for which you can see the “blackjack”, “solution”, and “lessons” directories. That way you can always keep this whole package with you everywhere.

Next you’ll be working on some files. For right now. I’ll create a file called file.cpp which will contain an empty main function. After you have finished working on a file and want to save your work, you want to commit it. Commits are checkpoints, the more commits you have, the more changes you have to revert your work for a “last working copy”. You will need to learn when to commit.

If you are working on a function for the first time, commit when you have made significant work on that function. If the function is small, you should probably wait until you have written the entire thing. If the function is long, work on it incrementally and split up your commits into stages. If you are patching or working on a broken function, make changes and commit only when you are sure the function works and you want to save your work. This will set up your checkpoints such that you will can always revert to an incrementally working copy of your program. The last thing you want is to look backwards through a long sequence of commits until you find a working program.

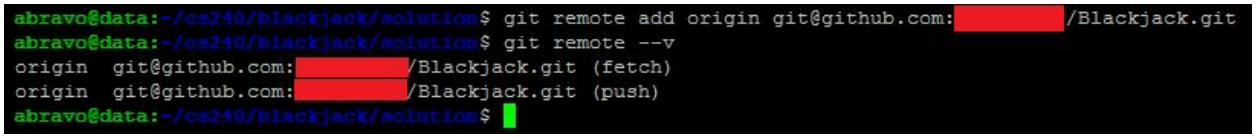
Git Remote

Your git repository must exist somewhere. I'll use my personal GitHub as an example. After creating a local repository (repo) on your machine, GitHub creates a remote repository for you to send all of your files to as a central store. You still need to direct your local git repository to send the files to the remote repository. In this case the repo is called <username>/Blackjack.git You'll type the command:

git remote add origin git@github.com:<username>/Blackjack.git

To check everything went as expected, use

git remote --v



You should now see your branch (origin). Later on you'll find out how to make other branches. If you ever wish to remove a remote repo location, use

git remote remove <remote_name>

e.g. git remote remove origin

To add a remote branch:

Git remote add <remote_name> <remote_URL>

Again you can check your remote repository locations with git remote --v.

Git Status

Before you are ready to commit, you should check to see if you have any untracked files. This is done through the status command:
git status

```
abravo@data:~/cs240/blackjack/solution$ git status
On branch master

Initial commit

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

        file.cpp

nothing added to commit but untracked files present (use "git add" to track)
abravo@data:~/cs240/blackjack/solution$
```

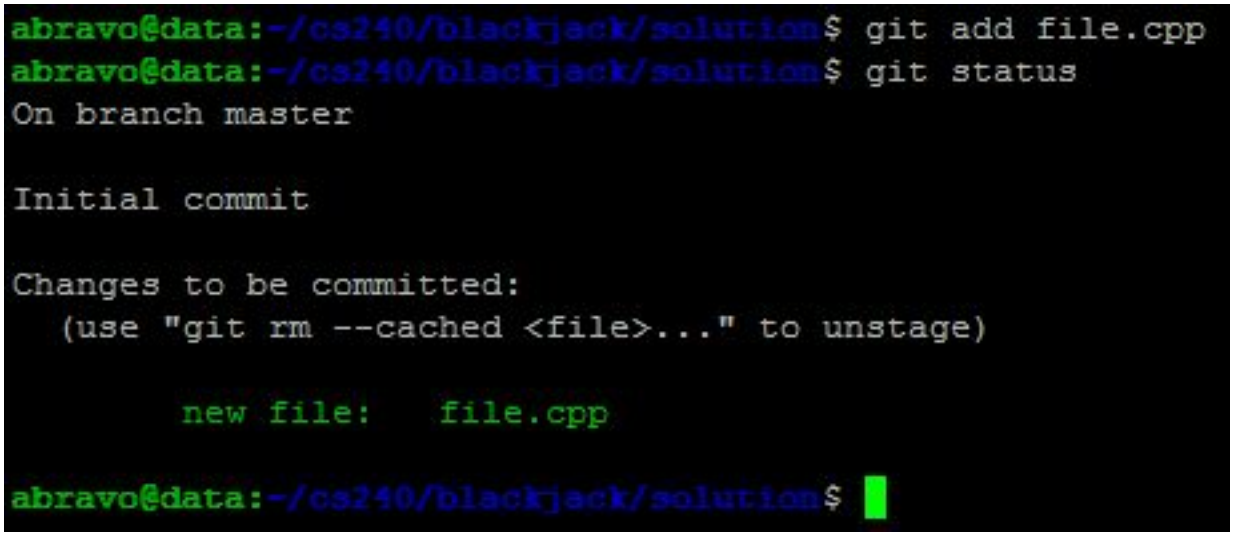
This cyan color means a brand new file has been created and is currently untracked. Untracked means that git is not managing that file, and will not provide version control as you make changes. A green colored file would be a file that you have already added to the staging area in order to be committed to the local repository. A yellow colored file is a file which has been modified but has not been added to the staging area in order to be committed to the local repository.

You may have different colors pop up for you. If you wish to copy mine, I will insert a sample gitconfig file. In your local directory at purdue, create a file called “.gitconfig” and use lines similar to the ones in my git config file which I have included in this tar. It is important that you don’t forget this period in front of “gitconfig”. In UNIX this will make the file hidden, hence you would need to use “ls -al” in order to see it. Please fill up the file with your given information and place the it in your home directory “cp .gitconfig ~/.gitconfig”

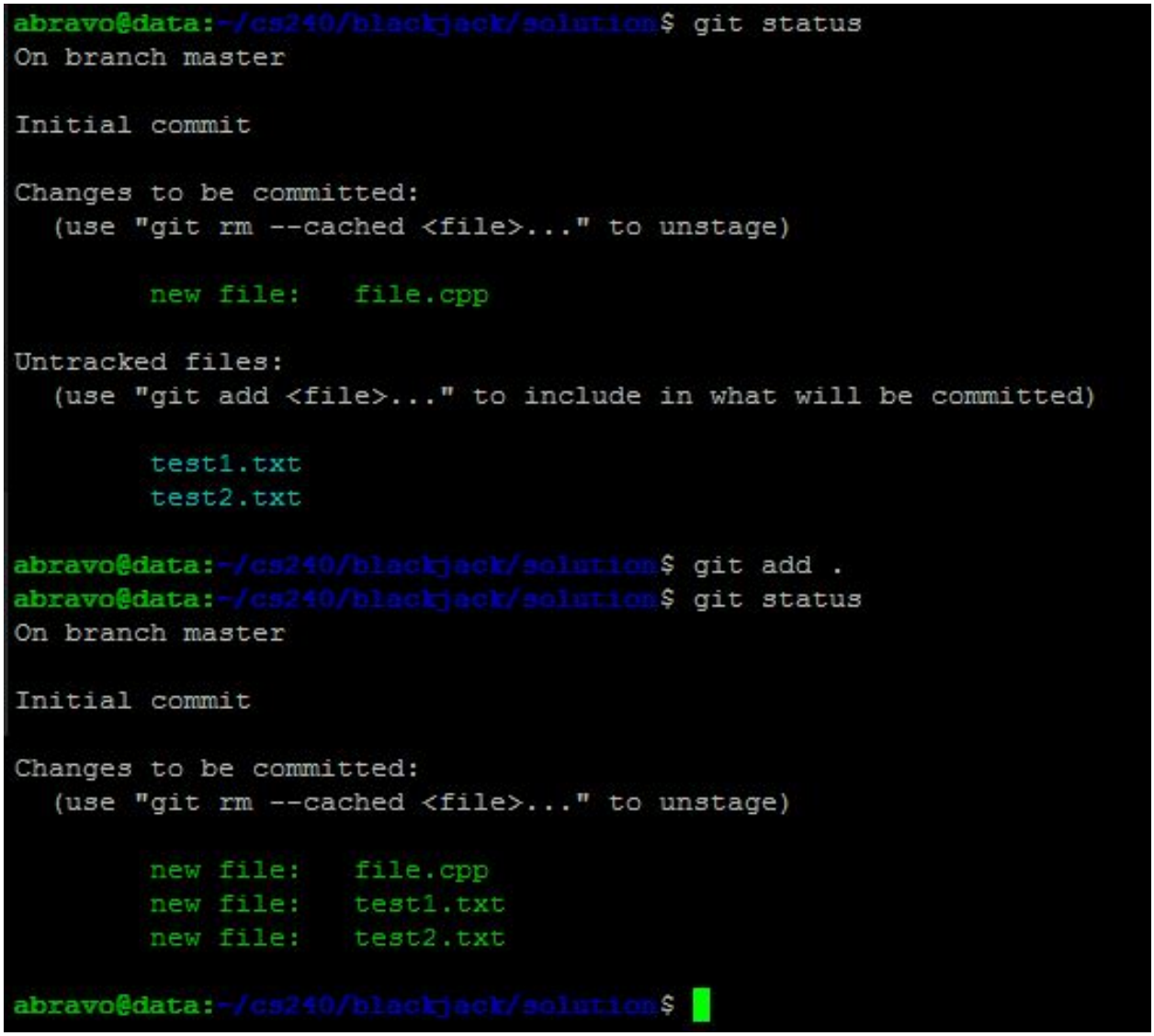
Git Add

Now that you know which files are currently in need of tracking, you will need to add them to your staging area so you can commit them later. This is done through the command:
git add <files>

You should be very careful here, only add what you wish to add. You'll find yourself doing "git status" at every single step in between git commands as a way of making sure you are uploading the correct files. **Note**: don't forget about the wonderful use of tab completion. Begin typing the name of your file and press tab to let UNIX complete the filename for you. It saves you time and it let's you know whether a file exists in your current directory.



If you wish to add all of the files at once (first of all be careful), but you can use the command "git add *" or "git add ."



As a reminder: Be very careful when you implicitly add every file. There may be files you did not intend to add.

Note: When adding files to git, avoid adding executable files or binaries. Git and other source control mechanisms create commits using things called differences/deltas/diffs. They have a snapshot of a file before the commit and compare it to snapshot of the file after the commit then they store the differences between individual lines (separated by newline characters) in the two files. This works great for text files which only change a few lines at a time, but pretty bad when you have an executable which changes and takes up a lot of space since the executables don't have lines, and you practically change everything at the same time. If you have a makefile, always run "make clean" before adding and committing.

Git rm

Oh no, you have messed up and added a file you didn't intend to! To remove it, you can use the git rm command. As git is saying, you can use the command

git rm --cached <file>

to unstage a change (unmark a change from being uploaded) from the staging area. So long as the '--cached' flag is present, git will not delete the file or any of the changes you've made. Instead it will only remove it from the staging area so those changes are not added in the next commit.

```
abravo@data:~/cs240/blackjack/solution$ git status
On branch master

Initial commit

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

        new file:   file.cpp
        new file:   test1.txt
        new file:   test2.txt

abravo@data:~/cs240/blackjack/solution$ git rm --cached test2.txt
rm 'test2.txt'
abravo@data:~/cs240/blackjack/solution$ git status
On branch master

Initial commit

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

        new file:   file.cpp
        new file:   test1.txt

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

        test2.txt

abravo@data:~/cs240/blackjack/solution$
```

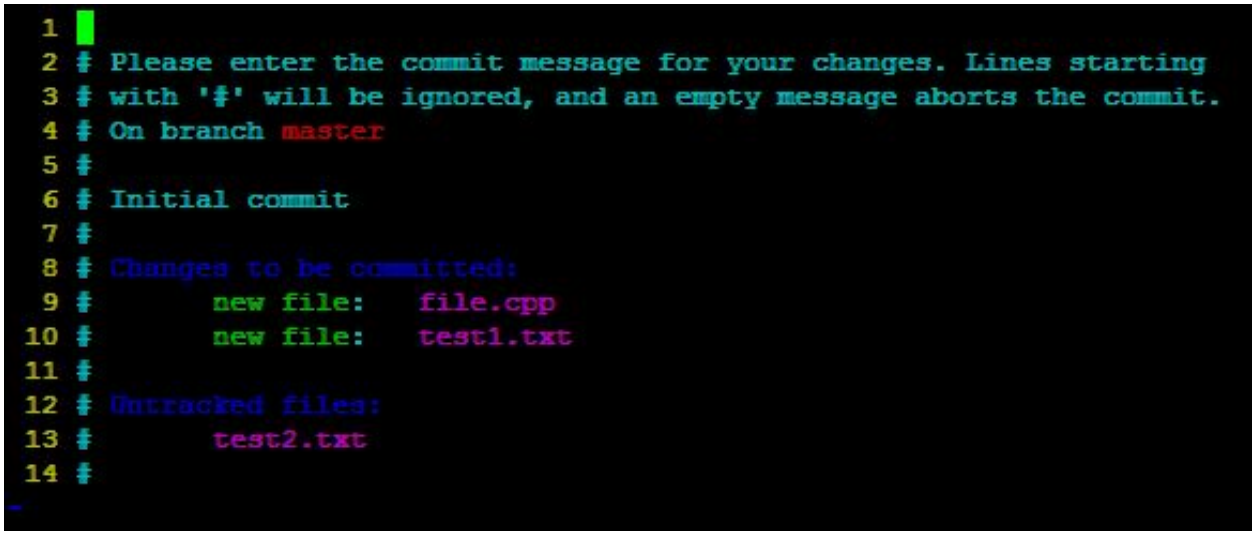
There you go, that file will now not be committed.

Git Commit

Finally we get to the actual use of git. When you are finally ready to save your files, you will create your checkpoint.

For a verbose commit (Note: the best way to commit and the way you must do it in the industry) Use the command:
git commit

A screen in your terminal editor will come up. For me the default editor is set to vim, hence all of the vim commands will work. (inserting with i, saving with :wq, or quitting with :q!). If you messed up and meant to include an extra file make sure you quit without saving. The commit won't be created and you can go back as if the commit command had never happened.



You'll be presented with a message like this. As the message states, any line that starts with # will be considered a comment and you will not see it in your log. This page let's you double check which files are being submitted and which are not. Let's fill it up a little with some good information.
After you save and quit from this editor screen, you will see a message like this

As you can see, you should be very descriptive when you make commits. The title itself is the most important. In 50 characters you should describe exactly what you are doing. Like the title of an essay, you should describe the main point of the commit. The reason why you want to keep the title short (even though there's no hard limit) is because most users use terminals to view these messages and it is possible for these titles to get cut off. Similarly, small terminals are usually 80 characters wide, hence you should at least avoid going past the 80 character mark for those users. After you exit you'll see this:

```
abravo@data:~/cs240/blackjack/solution$ git commit
[master (root-commit) 0e9d592] Descriptive title with < 50 characters.
 2 files changed, 4 insertions(+)
 create mode 100644 file.cpp
 create mode 100644 test1.txt
```

As a note, that mode you see “100644” stands for the permission flags of your file. If somebody were to download your file from the internet, it would have the flags 644 or “r w _ r _ _ r _ _ “. This can be changed to something else if you wish (755 for example if you wish for something to be executable and readable). You’ll learn more about these flags in CS 252.

If you accidentally saved and quit when you didn’t mean to, you can always edit your most recent commit on the current branch with

git commit --amend
Note: There is a double hyphen (--). Always note how arguments to commands are used.

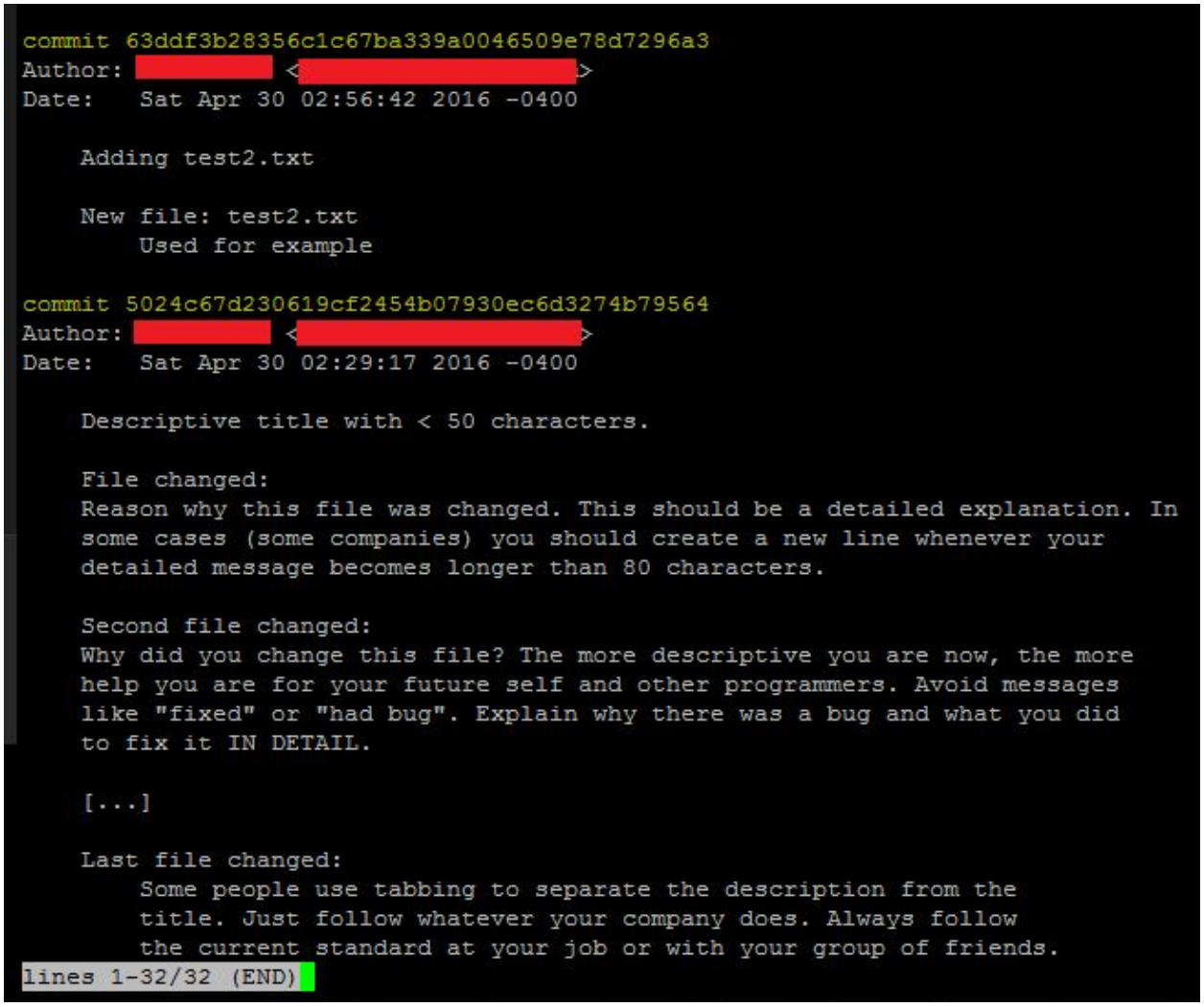
Finally, in some cases (usually quick code reviews at work) you may wish to skip on the descriptive part. For those situations, you can simplify your life with

git commit -m “commit title”

Now there is no reason to go into the editor and you expect your change to be so small that no other programmer would complain against it. There is nothing more infuriating to team members than a lazy programmer who does not have descriptive commits to describe why a change was made or what it does. Commits help the code review process since reviewers understand the reason and the goal of the commit. Do not be a bad programmer, you are representing Purdue.

Git Log

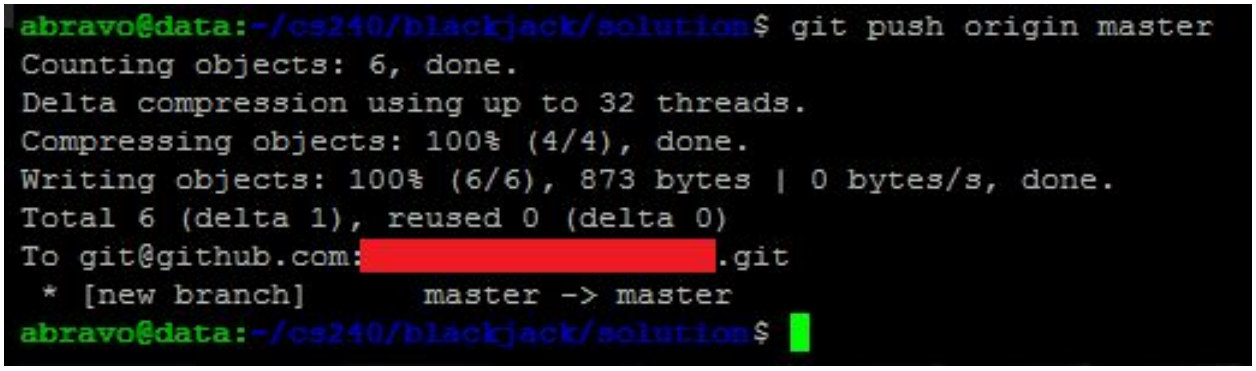
Now that we have some commits in our local repository. We can check our commit history with **git log**



git log is one of the most useful commands. It lets you read the history of what has happened, but most importantly it also gives you the hash of the commits. As described earlier, hashes are the unique identifiers of your commit and you will use them to revert backwards to your checkpoints in case you ever messed up.

Git Push

Finally we want to make sure that your commits make it up to the remote repository. Your changes are currently saved locally, but saving our code to the remote repository is why we want source control in the first place. It allows us to always get the latest code anywhere in the world for our project, it let's us be sure that our code is safe and won't be lost, and if you are using a location like GitHub or bitbucket, it allows the website to provide you with important information about your project. This could be statistics, lines added, lines deleted, collaborators, and much more. You'll have to explore those on your own. Use the command
git push <remote_repository> <branch>



Seeing this message means everything was correctly pushed and you are good to go. You'll never lose your work ever again. If something goes wrong (like a merge conflict) there will be many more things you'll have to do. Again this tutorial is only for a single person. Group projects will have constant merge conflicts which you'll have to figure out on your own.

Because git works on differences, every difference needs to be chained together. For example imagine the following commits, where A, B, and C are commits in the order:
A->B->C

If a git push ever fails, it's usually because you don't have the latest commits on your local repository. Perhaps your local repo only has A->B, instead of A->B->C which is what the remote repository contains. If you had a commit D in your local repository like A -> B ->D, git expects the following transition of commits: A->B->C->D
What would happen if D wasn't an addition, but rather a correction? A->B->D

In that situation, there are many things you could do, a git revert would work well, but if you want to fully cut off a commit from ever showing in your history (for one reason or another), you would have to hard reset to a given commit, make your new change, and before you push you would need to force a push

git push origin master --force

Note: be **extremely** careful here. Only use the --force flag if you absolutely know what you are doing and you wish to override something for good. You will not get to go back and any work you've deleted (either from you or a coworker) will forever be lost. This obviously bypasses the whole point of source control. Hence you must have a good reason for wanting to do this. If you are ever working with a team, there better be a good reason why you are doing this.

Git Pull

The twin command for git push would be to pull all of the latest information from the remote repository down to your local copy. When working with source control (especially with a team) this is probably the most important command. Working by yourself, you'll rarely use this command since you are the only one who is working on the project. You'll probably only use it if you switch computers and the local copy of the computer is out of date. I just wanted to mention that whenever you work on a team, this command needs to be run before you ever commit any files or wish to make any changes. Use the command
git pull [<remote_repository> <branch>]

This simple command and it will place all remote repository changes on top of the files you have. If your coworker and you worked on the same file, this will cause something called a "merge conflict." That is to say, git doesn't know whose changes are most recent. The person who caused the merge conflict must also fix it. They get to decide what changes are the latest and which ones are not. I suggest researching more about how to fix merge conflicts but that's beyond the scope of this tutorial.

If you have made a commit before pulling and you don't wish to lose the commit, you can run the command

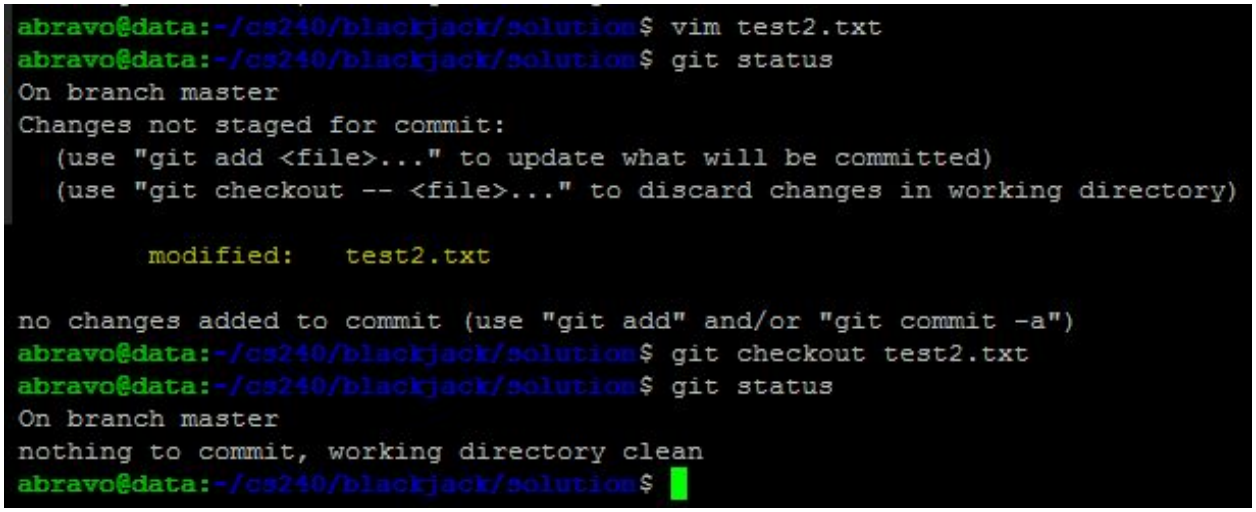
git pull --rebase.

This will keep your commit as the head. If there are no merge conflicts, you will be able to push as usual. Otherwise you will need to fix the merge conflicts first before you push.

Git Checkout

Sometimes you wish to go back in time. You messed up or want to go back to the way things used to be. This is the strength of source control. First we will explore the easiest case. Assume you have modified a single file and wish to revert a file back to the latest commit (discard any changes you made to that file). Simply do

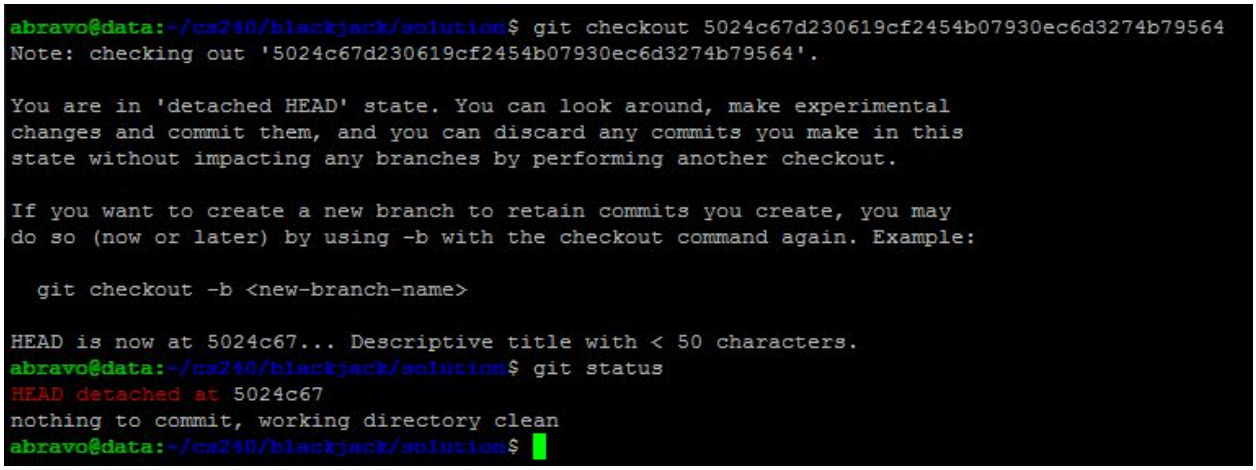
git checkout <file>



As you can see, after checking out a file we have thrown away any changes we were currently working on. Obviously only use this command when you intend to lose your changes.

Let's assume you didn't just want to lose the changes on one file, but you were working on a class project, you passed 80% of the test cases you've created. It's late at night and you've been making changes trying to solve those last 20% of the test cases, but your score only keeps getting lower. Now you are at a 60% and you can't remember what you did to change back. You've made 10 commits incrementally changing your project in a new direction but to no avail. In those situations you will probably wish to jump back to the state where you passed 80% of the test cases.

Using git log, you will find hashes of old commits. Then simply use
git checkout <hash>

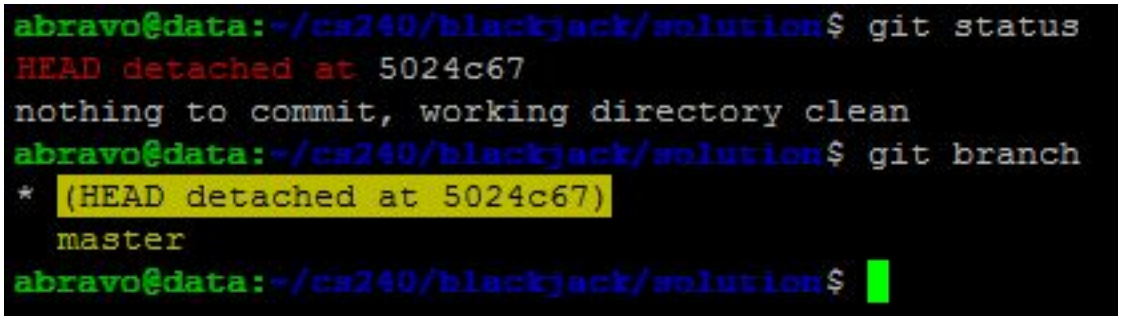


You'll notice whenever you try "git status" now you'll see that your head is detached. As funny as that sounds, it just means git knows there are technically more commits that have been submitted but you are not currently at what it considers to be the true "latest" commit; however, doing "git log" will show the current latest commit to be whatever hash you chose. You are in a new place called a "branch" (that explains why we call it a work-tree). These are splits in the code possibilities that are concurrently happening.

Git Branch

Now you have some branches in your work tree. To see your current branches, simply type **git branch**

This command will display any branches you have. The current branch will be marked with an asterisk (*)



If we want to leave and delete this temporary branch, we can switch branches with “git checkout master”, which means we jump back to our master branch, remember your temporary branch will be deleted.

If you wish to create a permanent branch, we simply type “git branch <name>” and the branch will appear

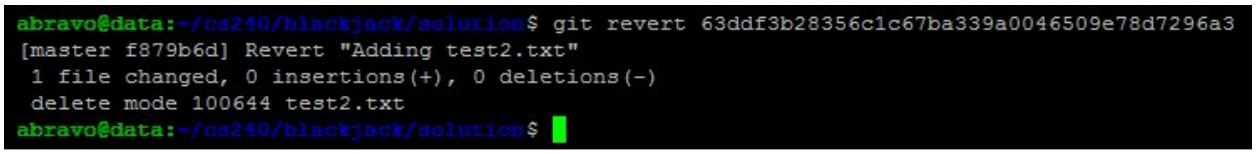
Again to switch branches, use “git checkout <branch_name>”. Your changes will only affect one branch at a time. When you are working by yourself, this will rarely be used; however, when you are experimenting with multiple changes or you are working with teammates, you should keep one branch as the “live” or “release” branch and one developing branch where you are actively making and testing changes. This release branch is usually called master and is considered stable. Only tested and verified changes should go here. Again though, this command is mostly used for when you are in a group of people all actively making changes.

Git Revert

If you messed up with a commit and wish to get rid of it, you can use the command

git revert <hash>

This command will act as a new commit (which is visible in the git history) and you'll be able to make descriptive comments just like when you were making a commit earlier to explain why a commit needs to be reverted. When you are done, push your changes.



Git Ignore

If you have a folder of file which you most definitely never want to add (perhaps it's a bunch of binaries or shared libraries which are machine dependent and need to be changed for each of your teammates), you can create a file called a git ignore. For now we will just cover the most basic for right now we will just cover the most basic of all git ignores.

1. Create a file called ".gitignore" (note the . in the front again. Yet another invisible file)
2. Place it in your top directory (where your .git folder for this project exists)
3. Add the name of the file or directory you wish to ignore, one whole line for each.

Git clone

If a git project already exists and you wish to copy it. You will want to clone it from the remote repository (the website) to your local machine. This can be done through the command:
git clone <repository_URL> [<destination>]

Simply go to the website where you store your project and find the ssh or https git clone URL.

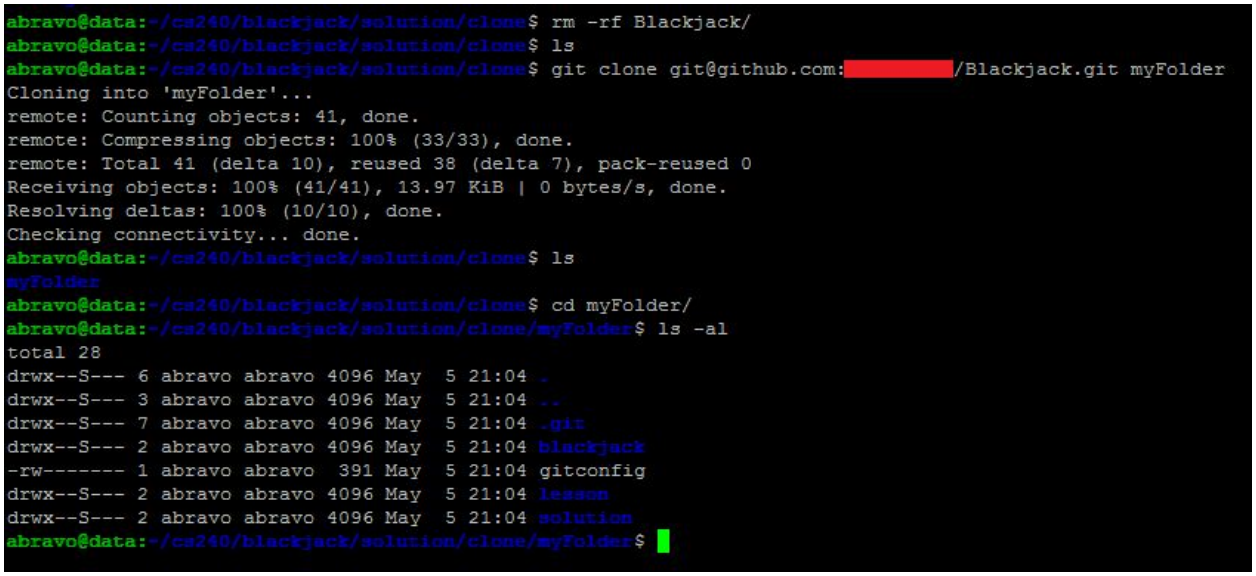
This is what the GitHub ssh git clone URL looks like:

If you do not want the URL but would prefer to download the project as a ZIP file. Hit “download ZIP”. Do this only if the project is finished or you do not intend to submit anything up to git anymore. The ZIP file will not come with the remote repository setup and none of the git commands will work.

Using the git clone command will create a brand new folder for you with the whole project as well as all of the git remote information set up for you. You will be able to git pull and git push right away on the latest changes.

```
abravo@data:~/cs240/blackjack/solution/clone$ git clone git@github.com:[redacted]/Blackjack.git
Cloning into 'Blackjack'...
remote: Counting objects: 41, done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 41 (delta 10), reused 38 (delta 7), pack-reused 0
Receiving objects: 100% (41/41), 13.97 KiB | 0 bytes/s, done.
Resolving deltas: 100% (10/10), done.
Checking connectivity... done.
abravo@data:~/cs240/blackjack/solution/clone$ ls -al
total 12
drwx--S--- 3 abravo abravo 4096 May  5 20:44 .
drwx--S--- 4 abravo abravo 4096 May  5 20:44 ..
drwx--S--- 6 abravo abravo 4096 May  5 20:44 Blackjack
abravo@data:~/cs240/blackjack/solution/clone$ cd Blackjack/
abravo@data:~/cs240/blackjack/solution/clone/Blackjack$ ls -al
total 28
drwx--S--- 6 abravo abravo 4096 May  5 20:44 .
drwx--S--- 3 abravo abravo 4096 May  5 20:44 ..
drwx--S--- 7 abravo abravo 4096 May  5 20:44 .git
drwx--S--- 2 abravo abravo 4096 May  5 20:44 blackjack
-rw------- 1 abravo abravo  391 May  5 20:44 gitconfig
drwx--S--- 2 abravo abravo 4096 May  5 20:44 lesson
drwx--S--- 2 abravo abravo 4096 May  5 20:44 solution
abravo@data:~/cs240/blackjack/solution/clone/Blackjack$ git remote --v
origin  git@github.com:[redacted]/Blackjack.git (fetch)
origin  git@github.com:[redacted]/Blackjack.git (push)
abravo@data:~/cs240/blackjack/solution/clone/Blackjack$
```

The folder created will have the same name as your project. In my case the project on GitHub is called “Blackjack” hence the newly created folder is also called the same. If you wish to create your own directory with a given name you can specify it at the end of the git clone command.



Other Commands

There are many other commands you might find useful sometime later such as git rebase, git reset, and many others. It’s all about practice and using what you need. For working on your own, you’ll most likely only need git remote, git pull, git add, git status, git commit, git push, and git checkout.

Whenever you run into problems, look up some git tutorials and examples. It’s a very useful tool and can be very powerful for group projects and working with teammates.