Some Basic Git commands:

git add JavaClass.java ←**use this to add a file named "JavaClass.java" to your git repository. Do this whenever you save a new version of your java program.**

git commit -m "Commit Message" ← **use this to add a message (replace Commit Message with the message you want) for the most recently added change, do this after each "git add". This commits your local changes to your local version of git.**

git push origin master ← **use this to "push" your change from your local computer to gitlab.**

If you have cloned the same project on multiple computers and are consistently pushing your local changes to gitlab (which you should be for practice and backups), when you go to another computer the "new" computer you are now using may have an old version of your program. Before you start coding, use the following command to get the most recent version from gitlab:
git pull

git clone pathnametogitrepo ← To get the repo from gitlab. You need to include the text you get from the clone dropdown box on gitlab after the word clone to replace "pathnametogitrepo".

Git Tips (in more detail)

This document includes the following discussions:

- Git Pull / Git Push
- Commiting Code
- Rebasing Branches
- Cherry-picking

Git Pull / Git Push

Pulling and Pushing is all about *remote* repositories. We **pull** from the remote repository when we want to update our local branches. We **push** to our remote repositories when we want to update the branches on the server. However, there is one intermediate layer. When you pull and push, your commits will also update an intermediate cache called *tracking branches*.

1. List your remote sources from inside your git repository by typing: **git remote** You will probably only have one. By default it is called origin.

    - To see remote details, use **git remote -v** This will list the remote name, the URL of the remote, and the command that defaults to it.

2. List your all of the local branches by typing: **git branch -a**. If you have pulled or pushed to a remote server, then you will probably have a *tracking branch* of the form /origin/branch. Keep these tracking branches in the back of your mind; if you are having problems, it can be useful.

3. To push your local branch to the server, using: **git push origin <branchname>**

    - You can use the shorthand **git push** If you have first set the upstream branch. You can do this by typing the full command and adding the **-u** flag. You only need to do this once. I prefer to always be explicit.

4. If git will not let you push your code, then there maybe a conflict on the remote source or your local branch may not align with the remote branch.

    - If you have just rebased, cherry-picked, or performed an operation that would change the hash of commits that are shared betwen the remote branch and the local branch, then you probably just need to force the push: **git push origin <branchname> -f**.

      However, *be cautious*. If you performed, say, a rebase, while someone else added a commit to the same remote branch (and you never pulled it down), force-pushing will cause that commit to be lost. (This is why its usually a bad idea to share branches.) Generally, however, rebasing, etc are standard git operations and force-pushing is something you will do quite frequently.

      To be safe, you can alway compare the commit history and make sure the messages match. But even then, if your rebase "skipped" commits or you "squashed" commits, the message history may not align either. Again, don't share branches. That way, at worst, the lost commit was something you wrote, so you can recreate it.

    - If there is a real merge conflict with the push, then you will need to resolve it before you can push. Generally, this means pulling the branch from remote and resolving the conflict locally. Then you will be able to push again.

5. To pull a branch from the remote server, type: **git pull origin <branchname>**

6. If you do not have a clean pull, you will need to resolve any merge conflicts.

    - Since pulling, unlike pushing, only affects your local repository, git allows pulls to complete even when there are problems.

    - If the branch you have pulled has been rebased, you will get a lot of merge conflicts. If you know that the branch has been rebased, it will probably be easier to create a temporary version of the branch: **git checkout -b <tempbranchname>**. Then delete the original branch: **git branch -D <originalbranch>**. Fetch the remote branch to move it to local a tracking branch: **git fetch origin <originalbranch>**. Finally checkout the original branch: **git checkout <originalbranch>**. Now you can just cherry-pick any commits you want to move from your temporary branch to your clean copy of the original branch.

Commiting code

When you have your repository is a state you'd like to persist, it is time to commit your code. There are two steps to this process - adding you code to the *staging area* and creating a *commit*.

1. Ensure you are currently at the top of your git repository.

2. Ensure you are on the correct branch.

3. Find out the status of your repo: **git status**

    - Identify the files that are already in your staging area and the files that have not yet been added.

4. Identify the files you want to include in your commit. You don't always want to include every file, but attempt to ensure your project compiles at each commit.

    - To remove files from your staging area, use **git reset -- <file>**. (Note the spaces *before and after* the double-dashes.) This does NOT change the file, it simply unstages it. If you want to return a file to its original state (the state at the last commit) you would use **git checkout -- <file>**.

5. Move the files you want to commit to your staging area by typing: **git add <file>**

    - Often, I will copy and paste the full file path listed by git status. Git status lists the file path *relative to your current directory*. This is good practice in case you are not at the top of your repo. The common commands like "**git add .**" or "**git add *.java**", it will only look in the current directory and below. When you are moving fast, it is easy to miss files in your commit.

- Remember that a commit only includes your staging area. You can continue to edit files, but only their state when you added them will be a part of the commit.

- Pay close attention to files that you have moved or renamed. Git is not case sensative, so renaming files for capitalization and cause wierd behavior. If files are not showing up, unstage the changes and force git to recogize the rename / move by using **git mv <sourcefile> <destinationfile>**. They should show up in the staging are as moves instead of deletions and creations.

6. Commit your files with **git commit -m "<feature>: <description>"**.
    - **-m** stands for message. If you exclude it, your session's default text editor will open (like vim or nano).
    - If you get stuck in vim, there are two basic things you'll need to do: To write to the file, hit **ESC** to enter command mode, then **i** to enter interactive mode. Then you can start editing the file. To exit, hit **ESC** to enter command mode then **:q** to quit or **:wq** to save and quit. If you get really stuck, hit **ESC** alot then type **:q!** to force quit.

Note: A clear concise commit message is important. A feature prefix can help identify a set of commits. At some point you will need to be looking a for a commit and will be thankful you provided a clear message!

- Changing your commit message can be painful if it is not the last commit in your branch (if it is, you can use **git commit --amend** to alter the last commit). The reason for this problem is that each commit's hash builds off of the previous commit's hash. It is also the reason why your commit hashes change when you rebase or cherry-pick commits.

Rebasing Branches

Generally, it is best to leave merging to your Git host (such as GitLab or GitHub). Pull requests are how we merge on our Git host. They allow your to track merges between branches, record the reason for the merges, and get feedback with code reviews. To prepare for a merge or pull request, we should clean up our commit history by basing our branch off of the latest commit on the branch we want to merge into.

Rebasing moves the base of our branch (and all subsequent commits) to the head of the branch we are rebasing against (usually the one we want to merge into). Rebase does *not* merge into this branch. Your branch will still be independent of the base branch. So feel free to rebase frequently.

1. Make sure the branch you want to *rebase against* is up to date.

- Do a **git fetch** to update your local copies of the remote branches on origin (the remove server). These branches usually look like: **/origin/branchname** when do you **git branch -a**.

- Checkout the branch you want to rebase against using: **git checkout <branchname>**

- Update the branch by doing: **git pull origin <branchname>**

    - If you get merge conflicts, you will need to resolve them, add the files, and create a new commit.

    - If you *know* that the code on the origin branch is what you want, cancel the merge (**git reset --hard**), checkout another branch (**git checkout <otherbranch>**), delete the branch you were trying to pull into (**git branch -D <branchname>**), then checkout the branch (**git checkout <branchname>**). Since the local copy of the branch is deleted, git will make a local copy of the local-remote (/origin/<branchname) branch.

2. Check out the branch you want to *rebase*: **git checkout <branchname>**
    - An important difference between rebase and merge - you rebase from the branch whose base you want to update, you merge from the branch you want to merge into.

3. Rebase by typing: **git rebase <branchname>**

    - Sometimes, before I rebase, I make a copy of the branch (**git checkout -b <copybranchname>**). If I really mess up, i can delete the branch I was rebasing and the copy branch will be the same as before. Just remember to delete these backups when you successfully rebase.

4. Rebase will move each of your commits, and if there are not any merge conflicts, it will proceed to the end. If there are merge conflicts, you will need to resolve them:

    - Identify the files that could not be staged by using **git status.**

    - Check each of these files for merge conflict notation and resolve the merge conflicts by choosing the code you want to remain. (If you are careful with your commits, hopefully you can build and run your code to ensure you correctly resolved the merge conflict).

    - As your resolve each files merge conflict, add it to the staging area using **git add <filename> .**

- When all of the changed files are staged, do **git rebase --continue**
    - *Do not commit the files yourself!* Let the rebase process handle it.
- Other Notes:
    - Sometimes, a rebase step will actually be empty - the branch you are rebasing against has already made the changes in the commit you are trying to add. In this case, git will inform you that you can do **git rebase --skip.**
    - You can cancel a rebase by doing **git rebase --abort**. Keep in mind that git does a *weird* thing by default - it tracks all of the "fixes" you've made during the rebase so that the next time you rebase, it will automatically apply those fixes. If you messed up your rebase by updating files incorrectly, these automatic patches can be a pain. To clear them, you can use **git rerere --clear**.

Cherry-pick for those Rebase-Averse

Rebasing is a single command that actually performs a set of operations you can do manually. Generally, rebasing creates a new branch off of the head of the branch you are rebasing against. Then it cherry-picks the commits on the branch you are rebasing, starting from the oldest. One at a time, it moves the commit to the new branch and gives you the opportunity to resolve any merge conflicts.

1. Ensure that the branch you want to rebase against is up to date. (See the rebase discussion above.)
2. Check your git log to identify the commits on the branch you want to rebase. Write down the commit hashes from oldest to newest.
3. Checkout the branch you want to rebase against: **git checkout <basebranch>**
4. Create a new branch: **git checkout -b <newbranch>**
5. One at a time, from oldest to newest, cherrypick the commits from your old branch onto your new branch: **git cherry-pick <commithash>**
    - If there is a merge conflict, then you must resolve it, add the conflicted files to your staging area, and use **git cherry-pick --continue**. *Do not create your own commit!*

6. When you've copied all of the commits from your old branch to your new branch, then you are finished and can delete your old banch.

Emergency!

- Prepare for mistakes. Before you make any big changes like rebasing or merging, create a temporary branch.

- Make a copy of your git repo. Why not?! Before you start diving into git trying to fix something you don't understand, just make a copy of your entire repository folder. (Make sure it includes the hidden .git folder).

- Read up on git logstash. This tracks *actions* on your git repository, not just it's state. It's a good way to undo a rebase (but you would have been safe if you had used a temporary branch!)

- Ask for help.