Git

# Git

* Git is a version control system
* Manages source code history. Used to manage a project.
* Local
* You run git with the terminal (command prompt or integrated terminal)
* Manages the different versions of a project in an efficient way (every time you commit)

# GitHub

* Hosting and collaboration provider
* Git repository hosting
* Can push a git repository to GitHub

# Key definitions

* Repository: The location where your code history is stored.
* Branch: “Folder” within the repository containing commits (not really a folder). It contains all the versions of the code which have been committed.
* Commit: Different **stages** of your code inside a branch.
* Untracked file: git knows they are files but git needs to be told which files it should manage/track. Need to explicelty tell git which files to include in the git repository (using git add)
* Staged/unstaged. Git is now aware of the file and tracking the changes to that file
* Branch head. This is the current version of the code you are using

# Key commands

### cd “insert path”

* Change directory to whatever git repo you want to be in.

### git init

* Initialise a git repository that git will manage.
* Initially says empty even if there are no files as there is untracked files (try, “git status” on newly initialised repository).

### cd “insert path”

* Change directory to whatever git repo you want to be in.

### git add

* Use this to tell git to track files (add to git repo). “git add .” tracks all files in the repo folder.

### Git commit -m “insert name”

* Make a commit.
* “-m” is the shortcut to give the commit a name. Need to see what changed in the commit

### Git branch

* See all the branches and what branch you are on.

### Git checkout

* Can be used to obtain a version of the code at a certain commit.
* Can also be used to switch branch
* Can be used to create a branch using “git checkout -b “branch name””
  + Note you require “-“ as spaces.
* You can also revert all unstaged changes by using “git checkout -- .” which will revert back to the most recent commit.

# Setting up a git repository and usage example

* Use the “git init “file path”” command to initialise an empty repository or reinitialise another repository.
* The repo is initially “empty” as there is no files being tracked/staged by git, therefore you require “git add “file name” or “.” for all to begin tracking.
* “git commit -m ‘commit message’” is used to commit the code and add to the repository. A master branch is automatically created. So the commit is saved in the master branch which is saved within the git repository.
* After changing the code, you can commit again (save a snapshot of the version of the code), by using “git add .” (example) and then “git commit…” again. The new code is added to the **git repository** and saved the two snapshots.
* **Note, git does not create copies of the code, it simply tracks changes. When we use “git add .”, git looks at all the files and checks if we changed anything. If we commit it these changes are saved. (highly inefficient to keep saving copies of the code)**
* **Also note that “head” of the branch is the last commit (usually) and is the current version of the branch.**
* If you want to go back to a previous version of the code, you find the ID of the commit and use the command “git checkout {commit ID}” and it updates the code with the commit version of the code. This commit is now the new branch head.
* With this, we now have the previous version of the code. However it is important to note that this commit is now no longer part of the master branch which is good for experimenting what went wrong with the code/can make changes etc (called detached head state). You cant commit this back to master branch as it is not part of the branch.
* If you wanted to delete the last commit, you use the command (while on the branch) “git reset –hard {commit ID}” where the commit ID is the commit previous to the last. This sets the branch head to that commit ID.
  + This is used to jump back to previous commits permanently (and not detached head state).
* If you want to go back to a previous commit after any **unstaged changes** (changes which you did not add or commit). This is reverting all unstaged changes and you use command “git checkout -- .”
* You can create another branch for a specific feature while having the main still functioning as it should. After the feature is finished you can merge the branch with master.
* Use git checkout -b <branch name> to create a new branch. This branch will have the same branch head as the master (unless specified otherwise) meaning it is the same version. (seen as (HEAD -> branch name, master). It also has all previous commits as the master.
* Can commit to that branch as normal and the new head becomes the latest commit to the branch while the head of the master remains the same.
* Now, if you checkout master then the changes to the actual files revert back to master and any new file (from the new branch) will be removed. If you checkout the new branch again then any new files will be put back and the version of the code updated again!
* To merge the two branches, “git merge {branch name}” which will combine the two branches. This also **adds any commits made in the branch to the master branch**.
* However, if both branches change the same element/line of code, there will be a conflict. To resolve the conflict manually, delete any code and >>>> <<<< that is not the code you want and save and recommit it to the master branch.
* To delete the branch, “git branch -D {branch name}

# Git merge vs git rebase

* git merge preserves all the commits in the branch being merged into the master. Note, this applies commits in chronological order. Meaning after the merge, its very hard to go back to the original version before it if the commits are mixed together. Use git merge --squash to put all commits on top of the master (in one commit)
* git merge ---squash "branchname" doesnt transfer branch commits to master, it merges and then stops before committing. You must manually add a commit and message.

### Git rebase example: Updating a sub branch with new master commits

* Looks at both branches and finds the common commit (this is the base of the sub branch).
* Git analyses both branches and finds the difference
* It looks at the current branch and saves the changes (difference between the sub branch and the master branch AT THE COMMON COMMIT) internally.
* It goes into the master and checks what changed in there (any new commits).
* If any new commits it uses the latest commit and applies the changes of the internally saved info (changes in the sub branch) to the latest version of master/latest commit made.
* It aligns both branches by rebasing the current branch with latest commit of the branch it is rebasing for then applies the changes made on top of the newly aligned branch.
* Rule is to: merge into master, rebase into feature/subbranch
  + I think rebasing master with a subbranch can potentially overwrite master branch commits with old commits.
    - e.g. if you have master branch: m1,m2,m3,m4
    - sub branch m2,m3o,f1,f2
    - then m4 is applied on top and main is: m2,m3o,f1,f2, m4

### Git push

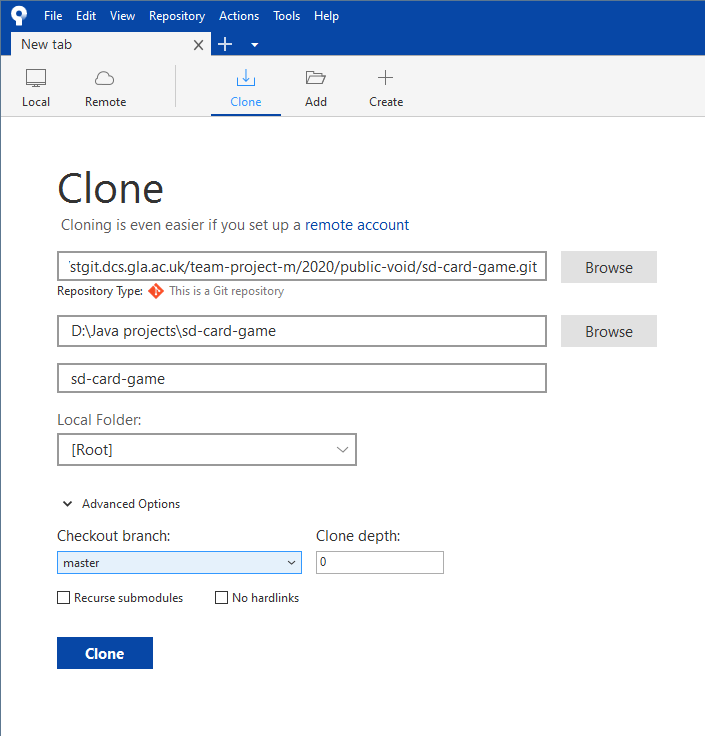
* Push commits onto the remote branch (origin?). Note, need to be up to date with the code before so need to pull before. You don’t want to overwrite code.
  + You cannot push to code which has commits ahead of you

### Git stashing and cleaning

* <https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning>

# Setting up the MSC project using Git

1. Download Git (<https://git-scm.com/downloads>)
2. Download source tree (<https://www.sourcetreeapp.com/>). This is just a Git GUI with an inbuilt command prompt that Simon recommended. I think its really nice cause you can see a visualisation of the branches.
3. Clone the project from GitLab either through:
   1. **Source tree**



* Where you specify a directory (use the Java workspace which is basically where all your projects are and name the file “sd-card-game”. You can also checkout specific branches too in the “Checkout branch” section.
  1. **Directly through the command line.**
* First set the directory you want to clone the project to. (e.g. D:\Java projects\) using the commands:
  + cd “directory”
  + Note this should be in your java workspace as before.
* Clone the remote repository on Gitlab using:
  + git clone <https://stgit.dcs.gla.ac.uk/team-project-m/2020/public-void/sd-card-game.git>
    - Note, this by default **clones the master branch** (or all branches. Mine only does master though)
  + **To clone a specific branch**:
    - git clone -b <branchname> <https://stgit.dcs.gla.ac.uk/team-project-m/2020/public-void/sd-card-game.git>

1. With the project cloned, we need to correct the tracked files (the files which git checks if there are any changes when we write code etc)
   1. Open the command line.
   2. Change directory to the git directory (your java repository and project name, e.g. mine is: D:\Java projects\sd-card-game) (note, this step isn’t needed if you use Sourcetree and open the command prompt after cloning through source tree)
   3. Check which files are being tracked:
      1. “git ls-tree --full-tree --name-only -r HEAD”
      2. Instead of HEAD, you can specify the branch name.
      3. Details: <https://stackoverflow.com/questions/15606955/how-can-i-make-git-show-a-list-of-the-files-that-are-being-tracked>
   4. Remove .classpath and .project using:
      1. “git rm --cached ".classpath"”
      2. “git rm --cached ".project"”
      3. Note, you might not have .project.
      4. Details: <https://stackoverflow.com/questions/1274057/how-to-make-git-forget-about-a-file-that-was-tracked-but-is-now-in-gitignore>
   5. Commit these changes with:
      1. git commit -m “Cleaning tracked files”
2. Copy any project files from the template folder into their correct place (all files and folders except “app”. Copy “assets” and place inside your “app” folder.)
3. The project should be ready to start using sbt.
   1. Open powershell.
   2. Change current directory to the project directory where “app”, sbt files etc are.
   3. Run “sbt eclipse” to set up the project for eclipse.
   4. Etc.
4. To use the project on eclipse, follow his steps to import the project as normal. However, now you can push and pull, switch branch etc as much as you like within the one project (no more copying over files to another project!)
   1. You can switch branch by e.g. “git checkout -b Unit-Classes-Sprint-2 origin/Unit-Classes-Sprint-2”
   2. This fetches the branch from gitlab and swaps your files with the version of this branch’s files.
   3. You can swap back by using:
      1. git checkout ”your branch name”
   4. You can pull from a branch using:
      1. git pull
   5. Or push your commits to the branch using
      1. git push
      2. Note, it will ask you to pull first to resolve any conflicts locally.

# Merge Procedure

I asked about the best procedure of working with multiple branches and merging. You can have multiple branches and merge them one after the other, if conflicts occur that cannot be auto fixed, its up to us to decide who should resolve it. The general procedure is:

1. Merge all our branches into one patch (this is just a branch which is a copy of master)
2. When wanting to merge to master:
   1. **First merge master into this patch**
   2. **Solve any conflicts**
   3. **Merge patch (with fixed conflicts) into master**.

* We do not want to solve any conflicts in the master branch. You should never merge directly into master, only through this patch apparently.
* You can merge your branch at any time into the patch and merging into master is usually a very critical moment.

For example, when looking to apply updates and finish the sprint, if we have branches a,b,c,d,e, it goes like:

* Merge branches a,b,c,d,e into patch-abcde (can merge at any time)
* Merge master into patch-abcde
* Solve conflicts in patch-abcde
* Merge patch-abcde into master which should have no issues.