The Basics (aka the 80% of what you will ever do with Git in your life)

In this set of work you will be asked to do some exercises on your own to understand the basic functionality of Git. Before starting with the exercises, you need to make sure that you have Git installed and configured on your machine. Instructions follow.

Installing Git

If you're on Linux or Mac, then you're all set (jump to section Configuration).

If you're on Windows, most probably you will not have Git installed. Click start and write Git-Bash.

- If Git-Bash is installed you will see it in the results. Click the Git-Bash Terminal and you're ready to start (jump to section Configuration).
- If not, then you need to install it. The link for download is at the Official Git Website. Pick the Windows Operating system, download and follow the installation instructions.

Configuration

Once you have Git installed, open up your terminal. On Linux, this is whatever terminal you happen to use, on Windows this means launching 'Git Bash' from the Start Menu and or Mac the Terminal app.

A big part of any version control system is enabling collaboration. For this reason Git needs to know your name and email, this is so that the people you are collaborating with know who made what changes, and how to contact that person. So let's do that.

If you're on Windows, open Git Bash otherwise whatever you use as terminal.

In your terminal:

- Register who you are with Git:

```
git config --global user.name "Your Name"
```

Example: git config --global user.name "John Doe"

- Register your email with Git:

```
git config --global user.email "your.username@some.domain.com"
```

Example: git config --global user.email "john.doe@hull.ac.uk"

That's it, you're done installing and configuring Git.

Exercises

1) Versioning

i] Create a directory: This is your project directory. As part of this practical let's say that this is a folder named 'LearnGit'. Navigate to the desired place in your filesystem (e.g. Dekstop) and create the 'LearnGit' folder. This can be done by either using the file manager of your OS or by running in the terminal the following commands (make sure that you have navigated to the appropriate location - e.g. Desktop - before running the mkdir command).

```
mkdir LearnGit cd LearnGit
```

ii] As every good project has a README, let's create that first. Again you can do this by using your favourite text editor (save the .txt file inside the folder you created before - e.g. Desktop/LearnGit/Readme.txt) or by running the following command in the terminal.

```
echo "This project is for learning the git VCS tool" > Readme.txt
```

iii] Now that you have a file, let's track it with git! First we have to initialise this directory as a repository

```
git init
```

This creates all the files needed for Git. If you have hidden files enabled in your machine you will be able to see a semi-transparent folder created in the LearnGit folder named ".git".

iv] Now we have a repo, but git does *not* track all files automatically. This is useful because we only need Git to track the files we need to version. For example, versioning binary files it's a waste of space. Of course we can also prompt git to ignore files that could possibly be tracked, by initializing a .gitignore file but more on that will follow.

In order to ask git to track a file we need to *stage* it. This is done by using the following command:

```
git add Readme.txt
```

(Readme.txt is the name of the file we need Git to track.)

v] Now if we run the command `git status` we should see something very much like the following.

```
git status \# On branch master \# \# Initial commit \# \# Changes to be committed: \#   (use "git rm --cached <file>..." to unstage) \# \#   new file:   Readme.txt \#
```

What this is telling us is that we have no history so far ('Initial commit') and a list of changes that will be committed *if* we commit.

vi] The file is now tracked, but we need to commit it to the repository. This is done by the following command:

```
git commit -m "Creation of the Readme file"
```

Where "Creation of the Readme file" is a short description of what are the changes that we are about to commit to the repository. In this case just the fact that we created the Readme file.

You can now check the `git log` to see the history of your repository.

vii] Let's create a new file in the directory but **do not add** it to git. Do this by either using the file manager of your operating system or by typing the following command to the terminal.

```
touch newFile.txt
```

viii] Check the status of the repo with

```
git status
```

Can you understand what the status describes?

ix) Let's update the **Readme.txt** file we created at the beginning. Open the file with your favourite text editor, add a new line and save the file. For those you prefer the terminal you can quickly add a new line of text by typing the following command:

```
echo "This is a new line into the Readme file..." >> Readme.txt
```

Now check the status of the repository. It should look like this:

```
git status /# On branch master /# Changes not staged for commit: /#   (use "git add <file>..." to update what will be committed) /#    (use "git checkout -- <file>..." to discard changes in working directory) /# /# modified:   Readme.txt /# /# Untracked files: /#   (use "git add <file>..." to include in what will be committed) /# /#   newFile no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that for both the new file and for the changed file, git is telling us that we need to run `git add` in order to save the changes. However, because README is being tracked by git already, git is able to *discard* the changes! That's already useful. No matter how many changes we make to a file we can always go back to a previous version.
As an exercise, stage (hint: add) the Readme.txt file and commit the changes to the repository. Don't do anything with the newFile.

x) Say that the newFile we created is a file we don't need to version or share. However, it annoyingly appears every time you type `git status`. In order to tell git to ignore this file you need to add its filename to a special file that is called .gitignore . Let's do it.
Type the following to the terminal:

```
echo "newFile.txt" > .gitignore
```

This command creates a new file named .gitignore and adds the line "newFile.txt" into it. The same could be done by using your favourite text editor (i.e. create a new text file, write newFile in it, close and save it as ".gitignore". NB.: In some cases the operating system will automatically add the .txt extension to your .gitignore file renaming it to .gitignore.txt. This will not work as git won't see the file. Make sure that your file has an empty name and the .gitignore extension. If you don't know how to make it work please ask one of us :))

If you now run `git status` you'll see that git is telling you that .gitignore is not being tracked, but you'll also notice that there is no mention of newFile.txt!

xi) Tell git to track .gitignore and commit this file to the repository.

2) Branching: The first example/exercise introduced you to the basic versioning functionality. Sometimes it is needed to fork one repository and work on a different **branch** than the **master**. In this set of exercises we will create a branch, add and commit some file to it and then merge it back to the master branch.

i] Continue to where you left at the previous step. Type the following command in your terminal:

```
git checkout -b "myNewBranch"
```

You should receive a message like the following:

```
Switched to a new branch 'myNewBranch'
```

This is it you just created and switched to your new branch. Everything you add and commit now will be part of this branch.

ii] Check the branches available on the repository by typing the command:

```
git branch
```

Try to understand what the output suggests.

iii] Now create one (or more) new document(s), **add** and **commit** it (them) to the new branch.

iv] If you now type `git status` you will find out that the first line reveals the branch you're working on and if you did everything correctly that there's no file needed to be staged or committed.
Make the following experiment to understand what a branch is. Type the following command that switches you back to the master branch:

```
git checkout master
```

You will receive a message like this: `Switched to branch 'master'`. Now check your folder. The files you created as part of the myNewBranch branch are not there! Don't panic because they are not lost. They are still there but right now you asked git to bring everything that is part of the **master** branch. The files of the **myNewBranch** are not part of it so they were removed. Let's bring them back. Type:

```
git checkout myNewBranch
```

Now you're back to your new branch and the previously missing files are back there. Phew...

v] Let's say that you finished working on the new fantastic feature of your project that you've implemented in this new branch, you tested it, it works and you are now ready to merge it with everything else (i.e. the files that are in the master branch).

First of all switch to the master branch. You now know how to do it (hint: git checkout master) Type the following:

```
git merge myNewBranch
```

You will receive the following output (not exactly the same as the commit id will be different)

```
Updating fecba8e..0b63991 Fast-forward  branchTest.txt | 1 +  1 file changed, 1 insertion(+)  create mode 100644 branchTest.txt
```

This is it. You created a branch, you implemented code in that branch without annoying the people working on other branches (e.g. the master branch) and when you finished you merged your code to the master.