VERSION CONTROL: GETTING STARTED WITH GIT

Git is one of the most popular **version control** tools available but is not the only one. For instance, it is possible to exercise some rudimentary version control of files with Dropbox, Drive, SVN or even manually with the command-line tools `FC` (which stands for File Control and is part of the Windows toolset) and `diff` (which stands for Difference and is part of the Mac toolset) as presented below.

Please create the `FileV1.html` file with your preferred plain text editing tool (I recommend Sublime which can be installed on all three reference operating systems).

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

Then save this code as `FileV1.html` into a local repository called `myrepo` (create it if it doesn't exist).

Now, create a new version of your file (save it in the same folder as `FileV2.html`) and (1) change the title of your web page and (2) insert a body. For instance:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>My page</title>
```

```
</head>
<body>
Hello everyone!
</body>
</html>
```

Then, please open the terminal and go inside the folder called `myrepo.`
In Windows you can compare both versions of your file with the command:

```
FC FileV1.html FileV2.html
```

In Mac you can compare both versions of your file with the command:

```
diff FileV1.html FileV2.html
```

In contrast to these applications, Git automatically manages the versions of your files and enables users to easily collaborate with others whilst creating large applications with multi-dependencies among the files of your applications. In order to better understand what Git is and what it does, a quote from the official book (available for free here: Git Book) perfectly illustrates this: *"Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots."*

In order to begin using Git, one must first download and install it. Below are the instructions for each operating system. The remainder of the guide assumes you will be using the Bash shell.

Note: Unless one is constrained to working only with editors such as **vim, nano, or emacs,** it might be worthwhile to check the source control capabilities of your IDE of choice, as many IDEs natively support Git (or have extensions that add support) and allow one to do many of the operations described below automatically. In any case, it is absolutely worthwhile to understand how Git works, and how it interacts with your code.

Windows:

1. Navigate to https://git-scm.com/downloads with your favorite browser.
2. Click Windows and wait for the download to start.
3. Once the file has been downloaded, open it and follow the install procedure.
4. Git will now be available in the machine along with Git Bash, allowing one to make use of git just as one would on any unix machine.
5. To get colored Git output, run `git config --global color.ui auto`

OSX:

Note: If you have already installed the Xcode command line tools, you should already have Git available in your system. The Git book recommends this as the ideal option. [source]

Option 1 (Official Git .dmg)

1. Navigate to https://git-scm.com/downloads with your favorite browser
2. Click Mac OS X and wait for the download to start
3. Once the file has been downloaded, open it and follow the prompts
4. To get colored Git output, run `git config --global color.ui auto`

Option 2 (With Homebrew already installed. To install Homebrew, copy and paste this command in the macOS terminal prompt: */usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"***)**

1. Run the command `brew install git`
2. To get colored Git output, run `git config --global color.ui auto`

Linux

Given the wide variety of Linux distributions available, it is best to check the recommended command on the website: Linux Install Instructions. **It is almost certain that git will be available in your distribution's package manager.** Then, to get colored Git output, run `git config --global color.ui auto`

Once Git is in place, one may now create repositories out of any folder and easily manage the versioning of any project one may develop in place. In order to have similar results as before with *diff* and *FC* we must run through a somewhat similar process. However, Git has some quite significant advantages that make the process of managing a codebase far easier in real use-cases. Let us now run through the previous example with Git so that these differences make themselves manifest.

First **create** a folder where you will store your codebase. This will now become your local repository and will be where Git will be set up. You must now **initialize** your repository with Git, so that any files or folders contained within can be tracked. This can be done as follows:

```
$ cd /some/parent/directory/
$ mkdir myrepo
$ cd myrepo
$ git init
```

Now, with the repository ready, proceed to create a file just as before (however, just name it ***file.html*** this time). Fill it with the first set of content mentioned on page 1 of this document. You may do this directly from the command-line (with the text editor of your choice, be it **vim, nano, emacs, sublime, etc...**) as follows:

```
$ nano file.html
```

Now check the status of the repository and you will see this output:

```
$ git status
On branch master


No commits yet


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

      file.html

nothing added to commit but untracked files present (use "git add"
to track)
```

This means that Git is aware that new elements are present within your repository but it has not yet been asked to keep track of them. With that in mind, proceed to add the file to the staging area (Git's cache saved in the binary **index** file) and then **commit** the changes, essentially telling Git to create a snapshot of the state of the repository (local repository).

Run the following commands:

```
$ git add file.html
$ git commit -m "added file.html"
```

Once this has been done, Git is now conscious of both the existence of ***file.html*** and of the state it was in during this **commit**. Now, open ***file.html*** with nano (or your preferred editor) and write "*Hello world!*" between the ***<body></body>*** tags. Once done, please query Git about the status of your repository.

```
$ 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:   file.html
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

We can see that Git is very much aware of the changes made to the file (comparing it with its last commit) and, crucially, there is no need to duplicate the file as Git will enable on to keep a continuous stream of changes to a single file as its progression through the commits where it is changed. This becomes evident when we ask Git to show us what changes it has identified.

```
$ git diff file.html
diff --git a/file.html b/file.html
index 1441a85..b76435f 100644
--- a/file.html
+++ b/file.html
@@ -5,6 +5,6 @@
 <title>Insert title here</title>
 </head>
 <body>
-
+Hello world!
 </body>
 </html>
```

If we then add the file to our next commit and then do the commit the state of Git's snapshot of the directory will now be brought up to date. We can do this just as before (with one simple difference that is worth mentioning: if we give **.** as the argument to **git add** it will add every file in the main directory's underlying folder structure, thus removing the need to list out every file we want to commit):

```
$ git add .
$ git commit -m "added a body to the file"
```

Finally we can now inspect how Git is tracking the changes made to the repository:

```
$ git log
commit a9b50797ee1c0f9847c47e659e990a57409abba4 (HEAD -> master)
Author: user <user@email.com>
Date:   Sun Jul 28 12:07:59 2019 +0200

    added a body to the file

commit d3e90205d172f1505286bbe6fbd49f7eff1b8590
Author: user <user@email.com>
Date:   Sun Jul 28 11:45:19 2019 +0200

    added file.html
```

And you can even query Git for the changes in your repository from two different versions (notice that each version has an identifier and **git log** gives the versions in order, from newest to oldest):

```
$ git diff d3e90205d172f1505286bbe6fbd49f7eff1b8590 a9b50797ee1c0f9847c47e659e990a57409abba4
```

Now, suppose you would like to add an experimental feature to your program, but would rather test it out first before adding it to your codebase (for example, you could have two possible solutions to a problem and would like to test them both out before committing to one). This can be accomplished by making use of several of Git's features. The first of these is its ability to create *branches*[1]. To create a new branch you must run the following command (please note that you may use any name, except that of other branches that already exist):

[1] The best way to conceive of branches is that they allow the codebase to evolve separately. They represent their own sequence of commits that do not affect those within other branches. In essence, a branch allows commits to be added without necessarily having them be part of the main branch's evolution until one sees fit to merge them. One useful example would be the following: A typical workflow would have a main ***master*** branch that would only be committed to when the software reaches a certain milestone or stable configuration and a ***development*** branch that would have the current and most up to date version that is being worked on.

```
$ git branch development
```

You can now check that this new branch exists by running the **`git branch`** command. The output should look like this:

```
$ git branch
  development
* master
```

This would indicate that there are two available branches in the local repository and shows a *star* **(*)** besides the currently active branch. It is worth notice that branches are created in such a way that they start out *at the active branch's currently active commit* and as such allow you to diverge from the current state of the directory while allowing the other branch to evolve separately if need be. With all this said and done, in order to switch to the newly created branch run the following command:

```
$ git checkout development
```

This command will, thus, change the currently active branch and make it so that any new commits that are added will now belong to the newly selected branch. Evidently one may switch back and forth between branches at will by changing the parameter of the *git checkout* command. It is also worth noting that in practice one usually desires to *both* create a branch and immediately switch to it, and, fortunately, git provides a convenient command that does exactly that:

```
$ git checkout -b <new_branch_name>
```

We can now proceed to make any changes desired to the software. In this case, let's assume you would like to test out the bold change of bolding the phrase that is to be displayed. Now proceed to change the file so that it looks as follows with your favorite editor.

```
<!DOCTYPE html>
```

```
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<b>
Hello world!
</b>
</body>
</html>
```

With this done you may now commit this change just as you did before, but this time this change will belong to the **development** branch. Run the following commands to do so:

```
$ git add .
$ git commit -m "added bold tags"
```

Now, assuming you have tested these changes and have decided that they are worthy of adding to your main (**master**) branch, you will need to do a *merge* of your changes. First change back to your **master** branch by running the following command:

```
$ git checkout master
```

You may notice that the changes you have made to your files are no longer present. However, there is no need to worry, these changes are stored as the commits that you have made on your **development** branch. Since these changes were not made on the master branch, you will not see them reflected on the files. In order to *bring* these changes to your **master** branch, you will need then to *merge* your **development** branch *into* your **master** branch. In other words, you will take all the commits that have been made to the development branch since it was created and add them to the master branch so that it now reflects the changes that were made on the other branch. To do this run the following command:

```
$ git merge development
```

You should see output similar to this:

```
$ git merge development
Updating 3372db0..b1a764f
Fast-forward
 file.html | 2 ++
 1 file changed, 2 insertions(+)
```

This indicates that Git was able to successfully bring your **master** branch forward to where the development branch was. It will not always be that simple and on occasion conflicts will emerge such that Git will be unable to easily merge your two branches and you will need to manually make the changes and create a new commit that reflects the changes made so that the two branches are now up to date. However, in this case, given the nature of the changes and the branches, it conformed to the simplest of cases.

Using GitHub

This is all fantastic for working alone, however, Git was designed with collaboration and teamwork in mind; thus, it is no wonder that something more is necessary to be able to collaborate with other's repositories. **GitHub** is one such option (though there are several others, such as **Gitlab, Bitbucket** among others), as it, essentially, enables one to create cloud-based Git repositories. Services like **GitHub** allow seamless synchronization among the many copies that might exist of a project, and, additionally, allow one to quickly merge all changes made to an application.

In order to use GitHub, it suffices to sign-up on their website and create repositories. These repositories will act as cloud-based copies of the local repositories, or, rather, as a centralized single source of truth of the stable state of the codebase to which all modifications are pushed and then spread to others. To connect a local repository to one in the cloud, it is necessary to add a “**remote**” to the local Git repository. This is accomplished as follows (per this guide) (You do not need to do steps 2 through 4 if you followed the first --Git-- section of this tutorial):

1. Create a new GitHub repository and copy its URL (make sure it is empty) and continue with step 3.
2. If you are not the owner of the project, but you have the repository URL of someone else, please create a new fork with the URL of the GitHub project (your own copy of someone else’s repository in your GitHub account) and go to the step 3 (and avoid step 4).
3. Create your local repository

```
$ cd /some/parent/directory/
$ mkdir myrepo
$ cd myrepo
```

And if you are working on a fork (you did step 2), please clone this fork in you local repository with the following command

```
$ git clone <your fork URL>
```

4. Initialize your Git local repository

```
$ git init
```

5. Create at least one file

```
$ nano file.html
```

6. Add at least one commit to be pushed.

```
$ git add .
$ git commit -m "file.html added"
```

7. Add your repository’s URL to Git with the following command. This will make Git interpret the `origin` keyword as a reference to the specified URL (copy it from the “Clone or download” button of the GitHub interface).

```
$ git remote add origin <your URL>
```

8. Push the changes of your local repository to the Web one (called `origin` in the previous command) with the following command:

```
$ git push -u origin master
```

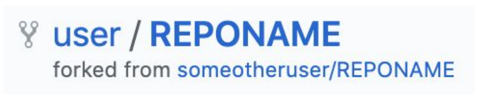
It is worth mentioning that the **-u** flag in the previous command links the current branch with the one on GitHub. It links the current local branch to the corresponding one in the remote repository. It only needs to be used during the first push operation. It has the added benefit of removing the need to re-state the remote destination and branch when doing *push* commands. As such, one may now simply[2] do the following command when pushing commits to the repository:

```
$ git push
```

This covers the procedure to be followed in order to make your commits appear in the remote repository; however, when one works as part of a team, it is also essential to have the commits of others available in the local repository.
If you are the owner of the project, you just execute the following command:

```
$ git fetch
```

However, if you are working in a forked[3] repository, you should say to Git which is the upstream of your project (go to GitHub and then, click on “forked from” and get the URL from the “Clone or download” button).



[2] It is possible that you may encounter an error that states: “you are not currently on a branch”. If this is the case, and you are sure that the changes you currently have are the correct version of your codebase (provided you have done a commit) you must do the following:
$ git log --oneline -n1 # this will give you the SHA signature of your current commit
$ git checkout <the_branch_you_should_be_on>
$ git merge ${commit-sha-signature}
If however you have not committed any changes you can simply run:
$ git stash
$ git checkout <the_branch_you_should_be_on>
$ git stash pop
Please see this answer on stackoverflow for more information: https://stackoverflow.com/a/4735584
The appearance of this error, however, should be unlikely.
[3] Please see the Pull-Request Section further down.

```
$ git remote add upstream <the url of the original project>
%then we verify that the upstream was correctly defined
$ git remote -v
$ git fetch upstream
```

The *fetch* command, in essence, brings updates of all the information from the remote repositories to your local repository. That is to say, you will now have the latest changes from the remote repository available locally. This command however does ***not*** affect in any way your local directory or files[4], it merely brings over the latest changes for each remote branch. Should one have more than one remote repository linked to the current repository, it might be worthwhile to call *fetch* with the ***--all*** flag, as this will update *all* the remote repositories' information.

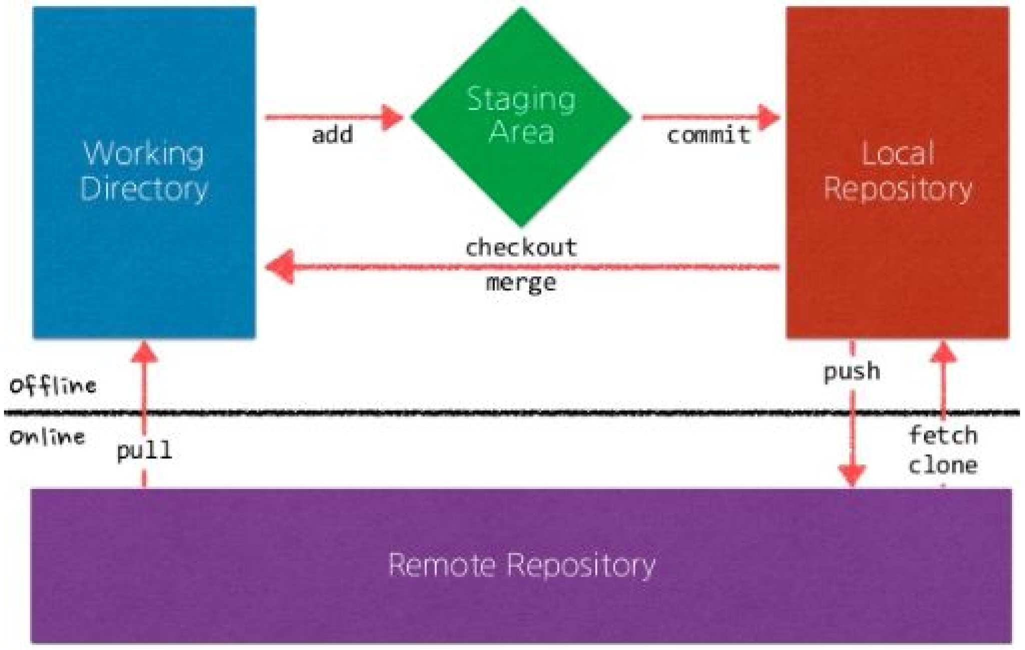
After you run the fetch command, you would usually *merge* the changes into the local branch. For the sake of example, let us imagine that you are currently working on the ***development*** branch of your local repository. Then you hear that a colleague has just uploaded his latest changes to the same branch in the remote repository[5]. After running the *git fetch* command, you would now have the aforementioned changes available to be integrated into your local copy of the repository. To do so, you would then run the following command to integrate these changes, which would, in turn, have the effect of "adding" all the commits that only existed in the remote repository onto the end[6] of your current line of commits, which would in turn modify all your files so that they are consistent[7] with the current *latest* commit.

```
$ git merge origin/development
```

[4] Strictly speaking, it will modify the content of the "**.git**" folder, however in practical terms, it will not make changes to actual source code you are working on.
[5] That would mean that, in essence, if both of you are working on the same repository (and merely have local copies of it), and there is a push made to the remote repository, then your fetch commands will automatically obtain the updates made, as the "origin" reference that is configured when you clone the repository will be the same.
[6] Or wherever the algorithm determines is correct to make sure the commit "history" is consistent.
[7] This, contrary to what one might expect, cannot always be made automatically. Should the automatic merge fail, one will need to manually resolve the conflicts in each file and create a new commit with all issues resolved. Most modern IDEs allow one to do this somewhat easily.

Evidently you must take care to avoid merge conflicts...

Now, in terms of the other common-use command, the *git pull* command acts in a very similar fashion to what has just been said. However, one key difference separates it from *fetch:* when you run a pull command, it has the effect of doing a *git fetch* command first and then immediately running a merge command. This creates thus the potential issue of unexpected merge conflicts (as mentioned in the above footnote) that must be resolved before work can resume. It is worth noting however, that, in general, pull commands can be used in place of manually merging any changes that were added to the remote repo.



Having nearly covered most the general use-cases of Git, we can now proceed to examine the image shown above that resumes all of the above into a simple diagram:

- **git add:** This command adds the files to the current commit, so that their change of state is tracked by Git.
- **git commit:** This command makes the changes part of your repository and brings forward the state of the repository. In a sense, with this command you "submit" your changes to git so that it now tracks their new state and any further changes are calculated from this new state.
- **git checkout:** This command allows you to switch branches, it makes it so the

files in your current working directory reflect the state of the latest commit of the branch that has been switched to.

- **git merge:** This command "adds" the changes from another branch to the currently active one, thus modifying the current state.
- **git push:** This command allows you to send your changes to a remote repository so that its state matches the local one.
- **git clone:** This command "downloads" a remote repository to your local system and creates a folder that contains its copy.
- **git fetch:** Brings the latest changes from the remote repository to the local repository without changing the actual current files.
- **git pull:** Does a *fetch* command and then *merges* the latest changes from the *current branch's corresponding remote branch.*

Pull Requests and Forks (Specific to Git-Hub)

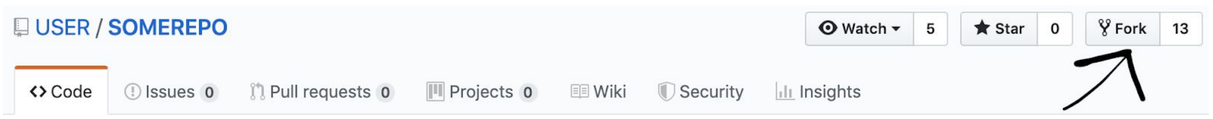
"Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch." [source] Pull requests allow one to have one final check, and even request changes, before merging work from a branch into another branch. Similar mechanisms exist with other providers. The key takeaway, though, is that when making large changes to a shared codebase, it is important that these changes are made with both the consent and approval of all other members of the team. They are done through the repository's page on GitHub with the "new pull request" button[8].

Forks, within the GitHub ecosystem, exist as copies of a repository that evolve as their own projects while allowing the "forked" repository to submit commits to be integrated into the main repository's code, which can then be given to all others that are working with said code. This is usually the method that is preferred when the

[8] Please see these two guides to better understand the process: How to create a pull request and How to merge a pull request.

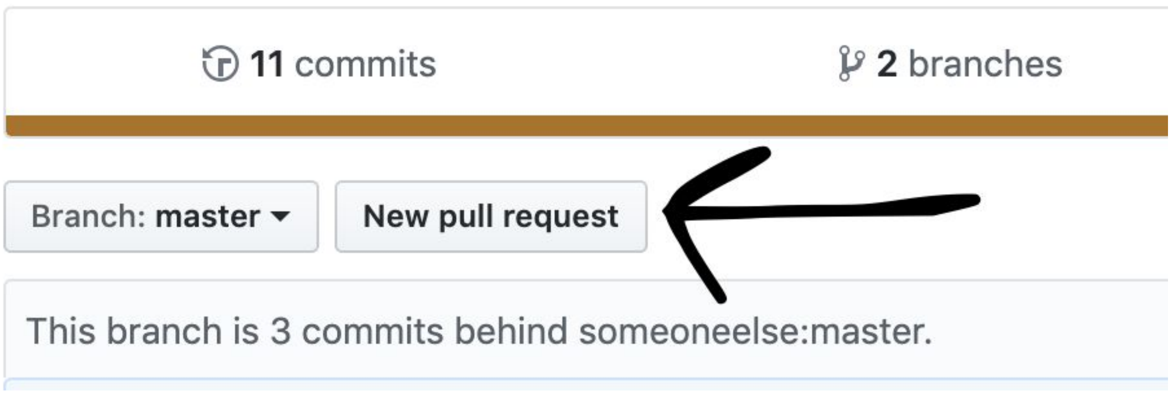
users that will be submitting code do not have ownership of the main repository and are, instead, taking it as a starting point for their own additions. In a real project, the way this would work is that only the project's maintainer has push access on the main project repository, and he reviews, tests and integrates the code sent by others (through the aforementioned pull-requests), in order to grow the codebase. Once these changes have been added to the branches of the main project, they can be "fetched" (as was mentioned before) by the other users. The advantage of this system is that it necessitates the use of a review mechanism before introducing any changes to a stable codebase, and that is a traceable record of the "who" and "what" of every update.

The creation of these forks requires little effort: all that is needed is to press the fork icon in the project's page and the fork is then created.



From that point onwards a copy of this project will be available within your own profile.

Having now completed some work, one may send these changes to the owner of the project by creating a pull request by using the "new pull request" button on one's fork of a project.



This branch is 3 commits behind someoneelse:master.

This will allow you to compare the changes that have been made, whether the branches can be automatically merged. If all goes well, you may now submit a pull request and, once accepted, it will become part of the new project[9].

Collaboration

When working as a team, it is important to have a clear collaboration strategy, in order to avoid having difficulties and incompatibilities. The choices made will impact the ease with which the project is developed. For instance, each developer should have a clearly delimited area of responsibility in the development of the application. For it might be problematic to deal with simultaneous or contradictory changes to a single part of the application. With that said, the following strategy may prove useful for a small team: A repository will be configured in such a way as to allow push access to all members of the team[10]. This repository will then have a development branch created. A choice must then be made: if the team is following the principle of continuous integration[11], all developers work on the same branch whilst constantly and consistently pushing their changes and pulling any changes from the repository. With good communication, no integration problems should arise. However, if they prefer to not interact constantly with the repository, they may create their own branches, do their work there, and finally merge the results to the development branch once done (perhaps making use of pull-requests, which are explained further below). Once the codebase has reached a milestone and has been tested, it may be merged into the master branch.

Note: It is important to understand that the perfect collaboration strategy will depend on the team and the project, and there really is no one-size-fits-all solution. However the key to any successful project lies in effective and constant communication, no

[9] Though evidently not the only way to accomplish this, using this technique leverages some of the existing infrastructure offered by git to make working in teams easier.
[10] Please see https://help.github.com/en/articles/inviting-collaborators-to-a-personal-repository for more details.
[11] See https://en.wikipedia.org/wiki/Continuous_integration. In essence, work is committed and pulled constantly, in order to avoid any inconsistency in the codebase and to ease the integration of the work of others.

matter how it is developed.

Common command sequences and procedures

Clone someone else's repository from GitHub

```
$ git clone https://github.com/somegithubuser/somerepo.git (1)
$ cd somerepo
```

Command **(1)** creates a local copy of the repository set in the url. If it is your own repository you will be able to push any local commits to the cloud, otherwise you will not be able to, unless you've been given access by the owner of the repository.

Initialize a git repository in an existing folder

```
$ cd /path/to/my/codebase
$ git init (1)
$ git add . (2)
$ git commit -m "first commit" (3)
```

These commands will allow you to begin tracking your codebase with **git**, by initializing the ***.git*** folder **(1)**, then adding all files and sub-folders to the repository **(2)**, and finally committing the result so that the state of the repository is on par with the files **(3)**.

Review the status of the repository (to check for changes that have not been added to a commit or untracked files)

```
$ cd /path/to/my/codebase
$ git status (1)
```

This command **(1)** will display the current status of the codebase, and will alert you to anything that differs from the latest snapshot and the actual files.

Configure git with your information

```
$ git config --global user.name "My Name" (1)
$ git config --global user.email "my@email.com" (2)
```

These commands **(1) and (2)** will setup your identifying information so that when you make a commit, this information (your name) will be visible to your collaborators and will allow you to identify yourself with your cloud base repository (your email). This only needs to be done once, as it is set globally.

Add a new file to an existing repository

```
$ cd /path/to/my/codebase
$ git status (1)
$ touch myfile.java (2)
$ vim myfile.java (3)
$ git add myfile.java (4)
$ git commit -m "added myfile.java" (5)
```

When adding new files to a repository it is a good idea to first check the status of the repository **(1).** Afterwards, one would create the file and modify it at will **(2) and (3)** (this could be done in many different ways, git makes no distinction based on what editor or IDE one uses, or how the file came to be in the folder). Finally one would add the file so that **git** tracks it **(4)**, and then one would commit these changes to update the current snapshot of the repository **(5).**

Review changes made to files

```
$ cd /path/to/my/codebase
$ vim myfile.txt (1)
$ vim mysecondfile.txt (1)
... (1)
```

```
$ git diff (2)
```

When one makes changes to one or more files **(1)** one can quickly see what additions or deletions have been made to those files by using the ***diff*** command **(2)**. This will output a specially formatted recap of what has been changed so that one may then either commit or undo said changes. **Note:** Most IDEs allow you to do this process interactively within them, and enables one to review said changes without needing to resort to the command line.

Create a new branch and switch to it

```
$ cd /path/to/my/codebase
$ git branch mybranch (1)
$ git checkout mybranch (2)
```

These commands first create a new branch that begins at the current state of the repository **(1)** and then switches to that branch **(2)** so that any further commit will be applied to that branch.
This can also be done with only one command:

```
$ git checkout -b mybranch
```

Check current and available branches

```
$ cd /path/to/my/codebase
$ git branch (1)
```

This command **(1)** will list all available branches and will show an asterisk **(*)** beside the current branch.

Remove a branch (assuming you are on branch *mybranch*)

```
$ cd /path/to/my/codebase
$ git branch (1)
$ git checkout master (2)
$ git branch -D mybranch (3)
```

These commands remove a branch locally. In order to do so, one should (though not must) first check the current branches **(1)**, switch to another branch **(2)** and finally remove the branch **(3)**.

Merge a branch into another (assuming you are on branch *mybranch* and would like to bring those changes over to master)

```
$ cd /path/to/my/codebase
$ git checkout master (1)
$ git merge mybranch (2)
```

With these commands one would first switch over to the branch one wishes to bring up to date **(1)** and then one would merge the changes into the branch **(2)**.

Fetch the latest changes from a remote repository

```
$ cd /path/to/my/codebase
$ git fetch (1)
```

This command **(1)** retrieves the latest information from the repository (new branches, new commits, etc.). Should one desire to merge any changes that were made to the current branch immediately one would run:

```
$ git pull (2)
```

This command **(2)** is equivalent to running **git fetch** and then ***merging*** the changes into the repository.
It is worth mentioning that one may merge the changes from the remote repository as follows:

```
$ git merge origin/somebranch (3)
```

This command **(3)** will merge the current commit from the remote repository's *somebranch* ***branch*** into the current branch.

Push your committed changes to your online repository

```
$ cd /path/to/my/codebase
$ git push origin master (1)
```

This command **(1)** allows you to push any commits that do not exist in the remote repository to it.

Add a new remote repository

```
$ cd /path/to/my/codebase
$ git remote add upstream https://github.com/adev/upstream.git (1)
```

With this command **(1)** one may add a new repository from which one may fetch changes. This is particularly useful if one is working on one's own copy of a repository.

Remove a file from the repository *and* the filesystem

```
$ git rm myfile.html (1)
```

This command will erase the file from your repository and will make it so git stops tracking it as well. If you wish to remove an entire folder add the **-r** flag as follows:

```
$ git rm -r folder
```

Remove a file from git only (and not the filesystem)

```
$ git rm --cached myfile.html (1)
```

This command will remove the file from git and it will no longer be tracked, but it will remain in your filesystem. This might be useful if you previously tracked a file but now wish to ignore it (for instance you might want to add it to .gitignore or move it to another project). As before if you want to remove a folder add the **-r** flag as follows:

```
$ git rm --cached -r folder
```

Remove a local branch

```
$ git branch -d mybranch (1)
```

This command **(1)** removes the local branch named *mybranch* but only if it is up to date and does not have pending changes. You must also be on another branch when you delete it. If, however, you would like to remove the branch irrespective of its status, use the following command:

```
$ git branch -D mybranch (2)
```

This command **(2)** will force the deletion of the branch.

Remove a remote branch

```
$ git push origin :mybranch (1)
```

This command **(1)** will delete the branch after the colon in the *origin* remote repository.

Other important git features to keep in mind

.gitignore

This file can (and should) be added to any git repository. It defines all the files that git should not track and that will not be committed to the remote repository. Generally one would avoid uploading files such as executables and log files, among others.

SSH Keys

In order to avoid having to identify oneself every time one interacts with the remote server, it might be worthwhile to set up public/private key-pairs and add them to the remote repository's configuration[12].

[12] https://help.github.com/en/articles/connecting-to-github-with-ssh This guide contains how to do this for GitHub, though the procedure is quite similar with other remote repository providers. In general, what one would do is generate a public key and add it to the repository, and, aftwards, everytime a connection is made to the repository it will make use of this key to authenticate you. This does create its own security risks, but is nonetheless a quite common procedure.