Git:

Overview:

- Git itself can be imagined as something that sits on top of your file system and manipulates files. Even better, you can imagine Git as a tree structure where each commit creates a new node in that tree.
- Nearly all Git commands actually serve to navigate on this tree and to manipulate it accordingly.
- Git is just as legitimate in the enterprise as any other VCS.
- Installation instructions can be found here:

  https://linode.com/docs/development/version-control/how-to-install-git-on-linux-mac-and-windows/

The full documentation can be found at:

https://git-scm.com/docs

Terminology:

- **master -** the repository's main branch. Depending on the work flow it is the one people work on or the one where the integration happens
- **clone -** copies an existing git repository, normally from some remote location to your local environment.
- **commit -** submitting files to the repository (the local one); in other VCS it is often referred to as "checkin"
- **fetch or pull -** is like "update" or "get latest" in other VCS. The difference between fetch and pull is that pull combines both, fetching the latest code from a remote repo as well as performs the merging.
- **push -** is used to submit the code to a remote repository
- **remote -** these are "remote" locations of your repository, normally on some central server.
- **SHA -** every commit or node in the Git tree is identified by a unique SHA key. You can use them in various commands in order to manipulate a specific node.
- **head -** is a reference to the node to which our working space of the repository currently points.
- **branch -** is just like in other VCS with the difference that a branch in Git is actually nothing more special than a particular label on a given node. It is not a physical copy of the files as in other popular VCS.
- **Merge** - git-merge - Join two or more development histories together
- **Checkout** - Switch branches or restore working tree files
- **Revert** - Revert some existing commits

Git config:

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating.

To do so, just add your name and email using use the following commands:

```
$ git config user.name "John Doe"

$ git config user.email johndoe@example.com
```

Git actions:

Action	Command
Create Git	Git init
Check Git status	Git status
Add a specific file to Git	Git add <hello.txt>
Add all files in folder	Git add * / git add .
Commit	Git commit –m <"your_info">
Add & commit at once	Git commit –a –m "message"

Branches:

- Branching and merging is what makes Git so powerful and for what it has been optimized, being a distributed version control system (VCS).
- Indeed, feature branches are quite popular to be used with Git.
- Feature branches are created for every new kind of functionality you're going to add to your system and they are normally deleted afterwards once the feature is merged back into the main integration branch (normally the master branch).
- The advantage is that you can experiment with new functionality in a separated, isolated "playground" and quickly switch back and forth to the original "master" branch when needed. Moreover, it can be easily discarded again (in case it is not needed) by simply dropping the feature branch.

Action	**Command**
Creating a branch	Git branch <my_bramch>
Switch branches (checkout)	Git checkout <branch>
Merging	Git merge <branch>
Get commit history	Git log
Checkout specific commit	Git checkout <commit_id>
Reset to a specific commit	Git reset --hard <commit_id>
Revert to a specific commit	Git revert <commit_id>

Conflicts

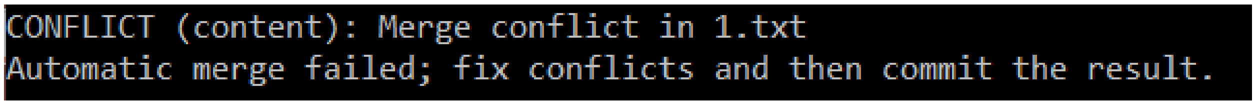
Merging and conflicts are a common part of the Git experience. Conflicts in other version control tools like SVN can be costly and time-consuming. Git makes merging super easy. Most of the time, Git will figure out how to automatically integrate new changes.

Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

Let’s consider the following scenario:
While we are inside **Master**, we create a file named 1.txt containing the word **“hello”**

Later we move into a **second branch**, **delete** the word **“hello”** from the file, and writing the word **“world”**.

When we will try merging from one branch to another, we will receive a conflict and the reason is simple.. git “doesn’t know“ if we want to keep the word “hello” or “world” inside 1.txt



```
CONFLICT (content): Merge conflict in 1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The solution is easy, we will need to modify the file manually, with the desired contents.

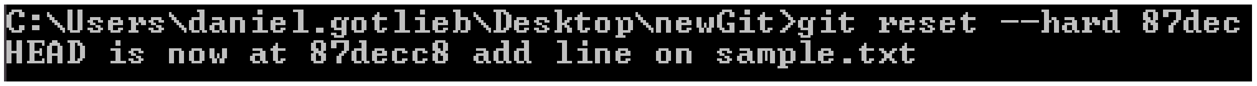
Detached head

- As we learned before each commit has a unique SHA1 hash.
- This unique id can be used to switch to commits.
- When a specific *commit* is checked out instead of a *branch* - is what's called a "detached HEAD".
- The consequence is that when you make changes and commit them, these **changes do NOT belong to any branch**, And this means they can easily get lost.
- To fix it, just checkout to Master and commit what you want to be committed.

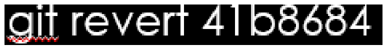
Action	Command
Add remote	git remote add <origin> <git@address.git>
Push to remote	git push -u <origin> <master>
Clone	git clone <git@address.git>

Reset and revert

- Reset is used when we want to "go back" to a specific commit, and will often be used with the flag –hard which will make everything match the commit you've reset to.



```
C:\Users\daniel.gotlieb\Desktop\newGit>git reset --hard 87dec
HEAD is now at 87decc8 add line on sample.txt
```

- Revert is used when we want to "undo a commit" The commit (sha) you pass to `git revert' is the commit we want to undo, not the state that you want to roll back to. 

```
git revert 41b8684
```

- For example, we committed a change in a .py file, and now we want to "cancel it", we can find the sha of this commit and use revert.
- An easy way to look at the difference between reset and revert is Reset will change an historical commit while Revert will create a new commit with the changes.