Git Cheat Sheet

Configuration

User Information

Git uses a set of user name and user email to sign the commit messages developed by the developer, as such these are required fields that are better configured globally when working from a personal or dedicated computer.

```
$ git config --global user.name "[your_name]"
$ git config --global user.email [your email]
```

Important to notice that if using your full name when setting the user name the command should not forget to use the "" to enclose a string that accepts spaces on it.

Text Editor

You can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your systems default editor.

```
$ git config --global core.editor <editor>
```

Since I run the git commands directly from a terminal I strongly recommend using a terminal base editor as vim, nano or emacs.

Merge Tool

The right merging tool can save a lot of time and errors when having to merge non fast-forward codes. There is a plethora of merge tools available, however I strongly recommend `meld` for its intuitive use and for its simple but effective graphical user interface.

```
$ git config --global merge.tool meld
```

Notice that you could change `meld` for the name of any other merging tool available.

Save on Typing

If you are using HTTPS URL to push your changes to a given repository the Git server will ask for confirmation through your user-name and password for every command accessing the repository. However you can store the credentials in memory for a few minutes and perform validation automatically, think of it as a timed log in session with the remote repository.

```
$ git config --global credential.helper cache
```

Working with Remotes

Tracking

Checking out a local branch from a remote-tracking branch automatically creates what is called a tracking branch (and the branch it tracks is called an upstream branch). Tracking branches are local branches that have a direct relationship to a remote branch.

```
$ git checkout -b [l_branch] <remote>/<branch>
```

Notice how the `l_branch` refers to the name of the newly created tracking branch and its upstream being `branch`. For example, assume that a new bug as been reported through issue 110, then we can easily checkout the code to fix it as:

```
$ git checkout -b issue110 origin/master
```

Cloning Remotes

Cloning is a way of start code development with respect an existing repository, it basically allow us to:

- Clones a repository into a newly created directory
- Creates remote-tracking branches for each branch in the cloned repository
- Creates and checks out an initial branch that is forked from the cloned repositorys currently active branch

```
$ git clone [-b <name>]
https://github.com/<user>/<repo>.git   [dir]
```

Notice that `-b <name>` may refer to a specific branch that wish to be cloned and `dir` is the name of the local folder to clone into. The command above shows a cloning using a git url via HTTPS, however git is able to manage other protocols such as SSH and its own git protocol.

Adding Remotes

Git repositories can be hosted locally or remotely. Git makes an special distinction for repositories hosted remotely in its interface. When a repository is created locally and it is desired to hosted remotely after its creation, a remote addition is required with:

```
$ git remote add <remote_name> <url>
```

Notice that traditionally the remote name is defined as `origin`, but this is only and identifier. Nothing stops you from using another name for your remote repositories.

Inspecting Remotes

The command is ideal when trying to get some information about the remote `<remote_name>`. The command prints a relationship of the remote branches that are being tracked and its local origins.

```
$ git remote show  <remote_name>
```

Remote Branches

Checking Out

Checking out a local branch from a remote-tracking branch automatically creates what is called a *"tracking branch* (and the branch it tracks is called an *"upstream branch"*). Tracking branches are local branches that have a direct relationship to a remote branch.

```
$ git checkout -b [branch] [remote]/[branch]
```

This command is ideal when for example checking out a branch to perform a bug fix, because it will set the upstream state automatically.

Deleting Remote Branches

After work on a given branch has been completed, and the user wants to delete that deprecated branch, the git push command can be used for deletions as in:

```
$ git push origin --delete <branch>
```

Notice that `branch_name` refers to the remote branch that is targeted in the deletion.

Additional Tools

Stashing

Stashing lets to record the current state of the working directory and the index locally without affecting the HEAD pointer. This is very useful when you want to save the code (you do not consider it ready to commit) but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.

```
$ git stash save
```

The previous command will save your work locally with a stash number (for later referencing)so that you can easily change your effort to other branches if needed.
Once work has been completed, the following command may be run to show the current stash numbers available

```
$ git stash list
```

The stash number will be used to revert your work to its latest state after as

```
$ git stash apply stash@{[stash_number]}
```

Renaming, copying and removing

```
$ git log <options>
```

```
$ git mv [old_file_name] [new_file_name]
```

Git is not very good in renaming automatically things, in order to do so a move operation is performed

```
$ git rm [options] <file_name>
```

Of particular interest is the option `--cached`, which will remove the file from the git repository but keep it in your file system.

Logs

The command display a list of the commits in the repository. By default it shows the SHA-1 number , the commit message and the author of the commit. Of relevant use is the option `--oneline` which display a brief version of the above message description that consist of the SHA-1 number and the commit message

```
$ git log <options>
```

The command is ideal when trying to go back to previous states as its shows the SHA-1 number required for the checkout action.

Gitk

Even though is not officially part of the Git software, the gitk tool is an excellent visualizer of a repository history

```
$ gitk [<options>] [<revision range>] [--] [<path>...]
```

Of particular interest is the combination of revision ranges which will basically plot the differences between the given revision and the path