Git Command Cheat Sheet

Recommended Editor	SublimeText	Package Control: GitSavvy, GitGutter
Recommended Diff & Merge Tools	p4merge Meld	
Recommended Git GUI Tool	GitHub Desktop SourceTree SmartGit GitKraken	

Artifact versioning FLOW	UNASSIGNED	assigned with status... M,D,etc	STAGED	COMMITTED

Section	Command	Description	Special example	Comments
HELP	git help	Displays help	git help -a	Displays all git subcommands and help
			git help -g	Displays all git help guides
			git help <cmd or guide>	Displays help for <cmd> or <guide>
			git help <cmd> <sub-cmd>	Displays help for <cmd> and <sub-cmd>
SETUP	git config	Displays the set of either local or global configuration parameters. We can replace global for local for local config settings. Review: https://gist.github.com/trey/2722934 https://github.com/mathiasbynens/dotfiles	git config --lcoal user.name "my name"	Sets user name for local repository
			git config --global user.name "my name"	Sets user name for the entire machine
			git config --local user.email "my@email.com"	Sets user name for local repository
			git config --global user.email "my@email.com"	Sets user name for the entire machine
			git config --global core.editor "subl -w"	Sets global editor. In my case sublime text 3 (inside linux)
			git config --local color.ui "auto"	Sets user name for local repository
			git config --global color.ui "auto"	Sets user name for the entire machine
			git config --local --list	Lists current local git config settings.
			git config --global --list	Lists current global git config settings.
			git config --list	List current settings applying for the current repository
			git config --global --edit	Editing the global configuration options from ~/.gitconfig
	git init	Initialize repository	git init	Initiates a git repository
			git init "project name"	Creates directory "project name"
	git clone ...	Clone an existing repository (URL)	git clone https://	We can create a repo cloning another

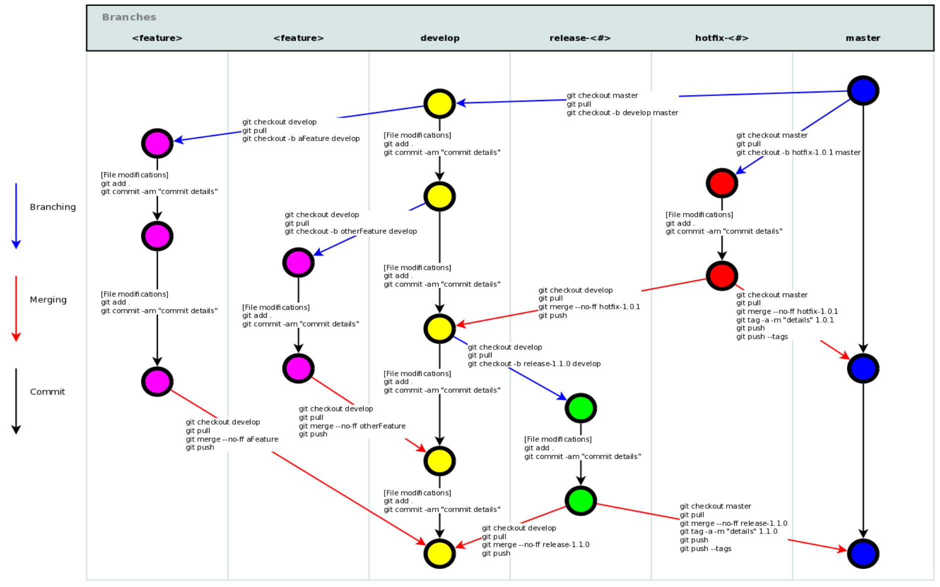
	If you wish to change your name/email from commit history after you have already commited... run the following command		git filter-branch --commit-filter 'if ["$GIT_AUTHOR_NAME" = "Old Name"];
			then
			export GIT_AUTHOR_NAME="New Name";
			export GIT_AUTHOR_EMAIL="new@email.com";
			export GIT_COMMITTER_NAME="New Name";
			export GIT_COMMITTER_EMAIL="new@email.com"
			fi; git commit-tree "$@"'	

OPERATIONAL	git status	Report of current repository status	git status --long	
			git status --short	
			git status -s	Equal to git status --short
	git add	Adds file from modified/delete/created area to stage status	git add .	Stages all files/changes
			git add newfile.txt	Assigns + stages newfile.txt
			git add file.txt	Stages file with name: file.txt
	git rm	Delete file	git rm file1	Deletes file and stages to git
			git rm --cached file1	Deletes file only from git cache and into unassigned status (i.e. untracks artifacts
	git mv	Move file	git mv file1 file2	Moves file1 to file2 and stages to git
	git commit	Commits staged files to the corresponding branch	git commit -m "<message>"	It is always a good practice to have a descriptive **tagged** message. E.g. local: changed... etc - file.txt [3]
			git commit -am "<message>"	Express commit: commits all files by first adding everything to the staging area
	git diff	Shows differences in the artifacs	git diff	Shows all differences in the repository (diff format)
			git diff HEAD	Compare the working directory with the LAST commit
			git diff --staged HEAD	Compare the staging area and the LAST commit
			git diff --cached HEAD	Same as git diff --staged HEAD
			git diff -- <file or path>	Shows differences in <file or path> that have not been staged
			git diff --staged -- <file or path>	Shows differences in <file or path> that have been staged
			git diff <commit id1> <commit id2>	Shows the differences between ALL files changed between <commit id1> and <commit id2> going from <commit id1> to <commit id2>. Interesting: One can use HEAD~<#> where <#> is the 0-th to #-th last commit. You can also use HEAD^ for the commit prior to HEAD commit.
			git diff <commit id1> <commit id2> -- <file or folder>	Same as before only for artifact <file or folder>
			git diff --name-only <commit id1> <commit id2>	Shows only the files that changed between both commits

			git difftool	Use the default tool to see differences! (Recommends: Meld or P4merge [also for merge!])
				Before this, we must set difftool defaults in .gitconfig via the git config command as follows: (e.g. in OSX)
				$ git config --global diff.tool p4merge
				$ git config --global difftool.p4merge.path /Applications/p4merge.app/Contents/MacOS/p4merge
				$ git config --global difftool.prompt false

	git ls-files	List files	git ls-files --other --ignored --exclude-standard	List ignored artifacts

Section	Command	Description	Special example	Comments
H I S T O R Y	git log	Displays the commit history and details which can be either long or short descriptions	git log --oneline	Online log for each commit
			git log <commit id>	Displays commit history up to identifier <commit id>
			git log <file or folder>	Displays commits related to the particular artifact
			git log <commit id1>..<commit id2> --oneline	Displays commit history from <commit id1> to <commit id2> excluding <commit id1>
			git log -n 3	Displays the last 3 commits
			git log --follow <file>	List version history for a file including renames
		Advanced Log	git log --oneline --decorate --graph --all	See what happens! "ALL" Branches
			git log --oneline --decorate --graph	See what happens! Only current branch
			git log --date=relative --oneline --pretty=format:"%C(ul yellow)%h%Creset [%Cblue%<(14)%ad%Creset] %C(bold green)%<(15)%an%Creset %C(blink red)%d %Creset%s" -3	
			git log --stat	Details of the information of each commit!
			git log --stat --oneline	Abbreviated
			git log --stat --oneline -n <#>	Same as before only for the last <#> commits
			git log -p	The patch represented on each commit. Shows the most detailed view of the project's history
			git log -p --oneline -n <#>	Patches in abbreviated format for only the last <#> commits
			git log --online --grep="<string or REGEXP>"	Searches plain text data sets for lines matching <string or REGEXP>
B R A N C H I N G	git branch	Create or list branch	git branch -a	Displays all branches. Master is always present
			git branch <new branch>	Create <new branch> (inherits the commit history from its parent branch): It's not active & inherits the commits from it's parent branch
			gi branch -m <old branch> <new branch>	Rename branch from <old branch> to <new branch>
			git branch -d <branch>	Deletes safely the specified <branch>
			git branch <new branch> <commit id>	Creates <new branch> from <commit id>
	git checkout	Enter branch or the state of any identifier	git checkout <existing branch>	Enter the <existing branch> to it's most recent commit
			git checkout <commit id>	Updates all files in the working directory to match the specified <commit id> in the current branch. This will put you into a DETACHED HEAD STATE. Any modifications are impermanent. You need to create a branch to retain changes
			git checkout -b <new branch>	You can create a new branch <new branch> based on the current branch (inheriting the commit history from its parent branch) and switches to the new branch immediately
			git checkout -b <new branch> <from branch>	Creates <new branch> from branch <from branch> and switches to the <new branch>
			git checkout	Sends you to the HEAD state of the current branch: the LAST commit
			git checkout -- <file>	It will undo the changes for the artifact <file>
			git checkout --detach <branch>	Detaches either a branch or a commit id to a HEADLESS state
			git checkout --detach <commit id>	
		Checkout Files: VERY USEFUL	git checkout <commit> file.txt	This command reverts "permanently" file.txt to match it's counterpart in the given <commit> and stages file.txt Commit is not executed yet
		Checkout to HEAD	git checkout HEAD	Goes to the head. This can be done when you want to revert any checkout that has gone to a certain commit id for a particular file or files.
		Checkout to Orphan State	git checkout --orphan NEWBRANCH	This will create an orphan (parent-less) branch. Useful when inserting propietary code or encumbered bits
	git diff	Show differences between two branches	git diff <branch1> <branch2>	This shows differences between <branch1> and <branch2>



```
git checkout master
git pull
git checkout -b develop master

git checkout develop
git pull
git checkout -b aFeature develop

[File modifications]
git add .
git commit -am "commit details"

git checkout master
git pull
git checkout -b hotfix-1.0.1 master

git checkout develop
git pull
git checkout -b otherFeature develop

git checkout develop
git pull
git merge --no-ff hotfix-1.0.1
git push

git checkout master
git pull
git merge --no-ff hotfix-1.0.1
git tag -a -m "details" 1.0.1
git push
git push --tags

git checkout develop
git pull
git checkout -b release-1.1.0 develop

git checkout develop
git pull
git merge --no-ff otherFeature
git push

git checkout develop
git pull
git merge --no-ff aFeature
git push

git checkout develop
git pull
git merge --no-ff release-1.1.0
git push

git checkout master
git pull
git merge --no-ff release-1.1.0
git tag -a -m "details" 1.1.0
git push
git push --tags
```

Image taken from geekgumbo: http://www.geekgumbo.com/wp-content/uploads/2011/08/nvie-git-workflow-commands.png

Section	Command	Description	Special example	Comments
UNDO OPERs	git revert	Reverts commit stage to a previous one. It appends a new commit with the resulting content. That way it does not destroy the history. Very helpful for finding bugs…	git revert HEAD	After an undesireable change, revert to the previous change
			git revert <commit>	Revert to update all files to <commit> state
	git reset	Resets any changes from the staging area back… HARD form resets and removes the changes. It reinstates the working directory to match the last commit	git reset <file>	If the <file> was modified and staged it removes it from the staging area and leaves the working directory unchanged
			git reset	Resets changes to the staging area for ALL files
			git reset --hard	This is a hard reset. Undos all changes and restores all files to the last snapshot. CAREFUL
			git reset <commit id>	Removes the commits after <commit id> from the history but leaves all files in the working directory unchanged and unstaged. Very USEFUL: when we made several changes to multiple files and wish to create commit state for each modified file.
			git reset <commit id> --hard	Does as previously but reinstates all files to match the exactly <commit id> snapshot. ALL CHANGES ARE LOST!
			git reset <commit id> <file>	Restores <file> to how it was at the commit <commit id> but leaves the working directory unchanged. This also works with the --hard option and its further consequences
	git clean	Cleans working directory similar to git reset. Handle with care. EXCEPT: those in .gitignore	git clean <path>	Cleans the <path> from all untracked files. Useful when we are compiling and the git clean will remove the compiled libraries or executables. Much like a make clean! Behavior can be changed by the config clean force flag. <path> could be empty
			git clean -n <path>	Shows what files are going to be cleaned in <path> without removing them. <path> could be empty
			git clean -f <path>	Forces the cleaning on the <path>. <path> could be empty
			git clean -df <path>	Will remove untracked files and directories from <path>. <path> could be empty
			git clean -xf <path>	Removes even ignored files mentioned in .gitignore <path> could be empty
MERGE	git merge	Merge two branches. It is always a good idea to use diff prior to merge: git difftool <branch 1> <branch 2>	git checkout <branch 1>; git merge <branch 2>	Fast forward merge of two branches. Standing on <branch 1>, merges <branch 2> into the current branch. Now both branches point to the same commit id! Only possible if the intersection of commit history on both branches is equal to <branch 1>. Does not preserve the branched-off of <branch 2> from <branch 1>
			git checkout <branch 1>; git merge <branch 2> --no-ff	NO fast forward merge of two branches. Preserving branched off. This will resolve into a merge commit with a message from the editor. This preserves commit history and branching. Now we can delete <branch 2> but it also preserves the commit history and how the development branched off!
	git merge	3-way MERGE. This scenario is useful when the commit history of both branches shows different commits from the branched out moment on both routes.	git checkout <branch 1>; git merge <branch 2> -m "<comment>"	Via 3-way merge, git does a merge via a 'recursive' strategy. This is not simple when both branches have modified the same file on different commits on their individual paths
	When there are conflicts? Which version to use?			
	Try 2 Merge!	You will get a MERGING State with modified working directory and Staging area. After merging run the mergetool!	git mergetool	It will show a panel with a 3-way merge scenario. The middle panel represents te original commit from where both branches branched out. In order to solve conflicts you go to each conflict and select each accordingly on the application. Save & Exit
	After resolving…	Commit!	git commit -am "<comment>"	Important: Git retains the original file from which we resolved the merge 3-way conflict. There is a way to avoid the creation of this <file>.orig files. 1. Ignore *.orig files in the .gitignore file 2. Local: git config --local mergetool.keepbackup false Global: git config --global mergetool.keepbackup false
AMEND	git commit --amend	Use with care as it implies that commit history will be lost	git commit --amend --no-edit	The --no-edit flag will use the same last commit message. It replaces the last commit with an entirely different commit! Use it with care as if you are developing on a further commit id and someone ammends on a previous commit all work will be lost!
			git commit --amend -m "<new message>"	Ammends the last staging area with the last commit and sets a new message for the commit.
REBASING	git rebase	Moving a branch from one base to anocher. Convenient when you have a feature development and the base from which you were based needed a bug fix or correction!	git rebase <new base>	Standing on <branch> you can set a <new base> that can be a tag to HEAD, commit id or another branch.
		In the case of conflicts…	git rebase <new base>	Will set the working and staging area into a rebase state
			git rebase --abort	Dedice to undo the rebasing
			git rebase --skip	Skip patches
			git mergetool; git rebase --continue	This will resolve the conflicts using the mergetool and proceed!

Section	Command	Description	Special example	Comments
G I T H U B	git clone	Clone remote repository	git clone <url>	Git creates a local copy of the remote repository to origin/master. But the origin refers to an "external source" (in this case, Github)
		REVIEW CLONED	git log --oneline origin/master	Shows the log of the copy of the remote repository
		When cloned… git branch -a shows all branches + remotes/origin/etc	git branch -r	Shows only remote branches
			git remote show origin	Displays information about remote origin
			git ls-remote --heads origin	Displays the references or tags of branches to remote origin
	git pull	Synchronize repository with remote and merges the contents: remote -> Local. AFTER CLONING as cloning saves the external source	git pull origin master	origin == Remote in case cloning was done first master == master branch of the remote repository
	git push	Push changes to remote repository	git push origin master	master in this case refers to the master branch in the local repository
			git push origin master --force	It will force to modify the master in Github
			git push	Pushes ALL the changes
	git remote	Remote repository settings	git remote -v	Displays remote git settings (from where to fetch and where to push)
			git remove add origin https://github.com/<repo info>	Connects a local repository to a remote one on Github in case you have not done it so far
			git remote set-url origin git@github.com:<repo info>	Change origin URL's to use SSH. Prior, we must set ssh-key. Follow the steps described ahead: 1. Create the rsa key with 4096bit encryption with the following command: ID_RSA_FILE="$HOME/.ssh/id_rsa.pub"; if [! -e $ID_RSA_FILE]; then ssh-keygen -t rsa -b 4096 else echo "id_dsa key exists" fi 2. Upload $HOME/.ssh/id_rsa.pub to Github 3. Test your remote SSH connection settings 4. Remember <repo info>==user/project.git 5. If you wish to make the ssh-key authentication with a password, submit the following command from the terminal: ssh-keygen -p
			git remote set-url origin https://github.com/<repo info>	Fallback to HTTPS protocol for fetching and pulling
			git push -u origin master	This scenario is interesting. Appears when we are creating the repository locally and want to upload it to Github for example. 1. We create the repo in Github with the name accordingly. 2. We set the remote origin: git remote add origin https://etc… 3. Execute the aforementioned push command. This command is only required once.
	git diff	Differences between local and remote repositories	git diff master origin/master	Show differences between the master local branch and the master remote branch
	git fetch	Updates local repository with the "remote" pointed by origin/<branch>	git fetch origin <branch>	Fetches branch <branch> from the "origin" remote reference
			git fetch	Can be used when we only have master branch
	What is the diference then between fetch and pull?		FETCH DOES NOT MERGE, ONLY BRINGS COMMITS AND DOESN'T CHANGE THE WORKING DIRECTORY. DOES NOT REMOVE LOCAL-ONLY BRANCHES. It is not a disruptive command and only updates the local copy of the remote repository	git pull, pulls data and merges such that your working directory reflects the remote repository. It will first fetch and merge. Therefore, it is always a good idea to do a **git fetch** as frequently as possible!
	git fetch	What to do after a fetch? A Merge!	git fetch; git merge origin/master	Since this is a fast forward merge, in case the remote repository is ahead in development and there are no updates not pushed to the remote, we sit on master and merge origin/master. Other scenarios will bid you to do a 3-way merge
		Alternatives? Use wisely!	git pull origin master	What does it do? Fetch + Emerge by creating a branch in the case where the remote repo and the local repo have gone on with commits separately and merges it recursively if there are no merge conflicts.
			git pull --rebase <remote-name> <branch name>	Sometimes it is easier not to merge in the upstream content and it's better to reapply your work into your current changes. This keeps your local changes AHEAD of the remote changes. This, obviously, only possible if there are conflicts.

L O G G I N G	git reflog	Check out the reference log of all movements of the repository. BEWARE, reflog is only available if you have worked on the repository. If not, for example if you clone another repo, you will not be able to extract the log of that particular repo. This is PURELY LOCAL. Reflog can assist to restore previous states like an undo history! BEWARE, reflog is only accessible for a certain period of time!	git reflog	Describes events and activity done inside the repository. It also details the particular action taken, respectively tagged, to arrive to each <commid id> and, further, it references also to the action with a syntax HEAD@{<#>}
	git show	Git shows more detailed information corresponding to the particular HEAD	git show HEAD@{<#>}	Shows the reflog for a particular "HEAD". Very useful to document changes in a timeframe with a corresponding time format. It can also allows to go back in time to for example an hour ago!
			git show master@{5.days.ago}	
	git log	All logs of a branch!	git log -g <branch>	Reference log details for a particular <branch>

Section	Command	Description	Special example	Comments
TAGGING	git tag	Tags particular commits in two ways: lightweight and annotated. First… lightweight	git tag	Shows all tags. Tags will also appear in git log!
			git tag <tag>	Tags the current committed state with tag <tag>. E.g. v1.0-rc1
			git show <tag>	Will show the particular history for the tag <tag>
	git tag -a	Annotated: They are stored as full objects in the git database. They are checksummed, they contain the tagger name, email and a message. They can be signed and verified with GPG (GNU Privacy Guard)	git tag -a <tag> -m "<message>	-a stands for annotated (--annotated) tag
			git show <tag>	Shows info for the annotated tag. This shows also the tagger info, date and the message for the tag
		How to determine if it is annotated or lightweight?	git cat-file -t <tag>	cat-file provides size or type of content, the -t corresponds to tag object. Now case: tag: the tag is annotated commit: the tag is lightweight
		Can we search tags? YES	git tag -l "<pattern>"	The pattern matches any Regular Expression. (or --list)
	git diff	Compare tags	git diff <tag 1> <tag 2>	
			git difftool <tag 1> <tag 2>	
	git tag	Update tags. Suppose you wish to move the tag to an earlier commit as you have made some changes to your development beta version for example	git tag -a <tag> -f <commit id>	Or --force, sets the annotated tag to the <commit id> and deletes the tag from the previous commit
		Delete tags	git tag <tag> --delete	Deletes tag <tag>
	git checkout	What if we checkout a tag>	git checkout <tag>	This will set put us in a detached HEAD state to the <commit id> pointed by <tag>
			git checkout -b <new branch> <tag>	This will create a new branch <new branch> from the tag <tag>
TAGS & GITHUB	git push	Tags in Github!	git push origin :<tag 1> :<tag 2> :<tag 3> …	Deletes tag 1, tag 2, … in Github
			git push origin --tags	Pushes all tags
			git push origin --follow-tags	Pushes only Annotated tags. Avoid pushing lightweight tags to the remote repository. It is a good practice to keep lightweight tags for development only
	git config	Configuring Push with TAGS	git config --global push.followtags true	Configures push to automatically push annotated tags
			git push origin master	Now it pushes the annotated tags as well

STASHING	git stash	Temporarily store current state without commiting. Useful for when you want to stop development and leave to finish later. This saves off work	git stash	Saves the current state - WIP (Work In Progress). BEWARE, it only saves the staged copy of the code - it only keeps tracked files. After stashing, the working directory is restored back to the last commit.
			git stash --keep-index	This keeps the changes in the staging area and moves all other changes to the stash
			git stash save "<message>"	Saves stash with a message
			git stash -u save "<message>"	It will save unfinished work for all artifacts in the repo
		List stashes	git stash list	Shows all stashes as stash@{<#>} Notice that stash displays in reverse order
		Show stash details	git stash show stash@{<#>}	Shows defails of artifacts changed between last commit to the saved stash
		Apply a stash	git stash apply	Applies the first stash or stash@{0}
			git stash apply --index	Applies the stash exactly as you left it: exact restoration. When you do not use --index it will not retain the changes that were already submitted to the staging area.
			git stash apply stash@{<#>}	Applies the corresponding stash
		Drop a stash	git stash drop	Drops the first stash or stash@{0}
			git stash drop stash@{<#>}	Drops the corresponding stash
		Pop a stash	git stash pop	This applies and drops the first stash
			git stash pop stash@{<#>}	Applies and drops the corresponding stash
		Clear the stash	git stash clear	Drops all the stashes in one command
		Creates a branch from a stash	git stash branch <new branch>	Creates branch <new branch> from the stash. It not only creates the branch, it moves into the branch, applies the branch and drops it as well. A one fit for all. Very convenient