Git Cheatsheet — Jim Gagné — 3 March 2012

Set your identity and other global parameters

```
git config --global user.name "David Beckwith"
git config --global user.email "dbitsolutions@gmail.com"
git config --global color.ui "auto"
git config --global github.user <github_username>
git config --global github.token <github_token>
git config --global apply.whitespace nowarn #Ignore whitespace: Ruby is whitespace insensitive
```

Configure github wraparound: first install "hub" ("port install hub" or https://github.com/defunkt/hub)

```
hub alias <shell>  #Show instructions for aliasing git as hub in your shell. In .bashrc: "alias git=hub"
```

Show global settings (in ~/.git/config)

```
git config --global --list **OR** cat .git/config
```

Sample config file: github.com/matthewmccullough/git-workshop/configuration/dot-gitconfig

Set up aliases

```
git config --global alias.co checkout
git config --global alias.logp "log --graph --pretty=oneline"
```

Useful aliases

```
gb = branch; gba = "branch -a"; gc = "commit -v";
gd = git diff | mate; gl=git pull; gp=git push; gst=git status
```

Tell git to ignore files -- use ".gitignore" (git tracks the .gitignore file, just like any other)

Any files listed in /.gitignore (i.e., in root directory), each on separate line, eg, "*.log", " db/schema.rb", " db/schema.sql"
Filenames ending with "/" are treated as directories; "dir/" ignores contents and subfolders of the dir directory
Adding an empty ".gitignore" to a directory => git ignores ALL its files
Adding a nonempty ".gitignore" to a directory => ignore listed files, e.g., *.log or **/*.log (**/ = ignore in subdirectories)
Lines starting with "#" are comments; lines starting with "!" tell git NOT to ignore (eg, track an empty directory)
Sample Rails .gitignore: http://www.icoretech.org/2009/08/my-rails-gitignore/
Already committed files that you then .gitignore => file stays in repository unless deleted but cannot be modified

Git overview

Git is a *distributed* version control system (DVCS) with local and remote repositories, designed for multi-user projects
It has 3 or 4 buckets: the local project workspace, a staging area ("shopping cart" or "index") where you add and remove files before a commit, the local repository (usually located within the project itself), and (optionally) a remote repository.
There's only one repository directory (".git/", defaults to project root). Add/modify/delete files without confusing git.
Git tracks file *contents,* not files themselves. By default git ignores empty project directories.
Git is 10-100 times faster than subversion because it rarely uses the network; more efficient in many ways.
Every object is referenced by a 40-digit SHA1 hash that's unique in the universe; can refer to object with first few digits
Identical objects have identical SHA1 hashes; the slightest difference causes dramatic change in hash

Getting help

You have to have downloaded the git-doc files to access the EXTENSIVE help system; seems to be in default install

```
git --html-path                 #show path to local help/man files
git help (equivalent: --help)   #quick summary of common git commands
git help --all                  #short table of ALL 153 git commands
git help git --web              #display git man page in default web browser; MANY helpful links
git help <command> -w           #display <command> man page in default web browser; -w same as --web
```

Important terms and notes

"HEAD" = name of the most recent commit in current branch; "HEAD^" is the commit just before HEAD;
"HEAD^^^" or "HEAD~3" is the third previous commit; HEAD@{"1 week ago"} = whatever the HEAD was a week ago
"origin" = alias for the current remote repository. You can also use SHA1 ids (1st few digits suffice) in quotes as "names"
Note: commands that affect files accept the "-n" or "--dry-run" option to display results without carrying out command
Note: use " -- " to separate command options and branch names from file names in case of ambiguity

git clone—copy remote repository locally in a new directory (default directory name = same as repo)

```
git clone <repo>      #create directory and copy entire repository including branches (using ssh, git, or https)
git clone <repo> <dir>   #create dir/ here at pwd and copy entire repository (ssh, git, or https) into dir/
```

OR: start tracking an existing progject

```
git init          #run in existing project root directory: initializes git, creates .git/ directory within project
git init <dir>    #run in project directory: initializes; puts .git/ in dir/, creating dir/ if not there
```

Note: you can always run "git init", even on a project git is already tracking

Checkout—move to a new branch or create a new one; "-f" option discards current unsaved files

```
git checkout <branch>   #move to new branch and replace project files with <branch>; must first commit or "-f"
git checkout -b <branch>    #start a new branch and check it out using current project, creating new commit
```

Most important command: run frequently whenever you want to know what's happening

```
git status      #show branch name; show index status: new/changed files added, deleted, or not yet in index
```

git add—add new or modified files to the Index (also called "staging area", acts like a shopping cart)

```
git add .            #add all new or modified files to the index
git add . -A         #add ALL files to index: new, modified, deleted, renamed, etc.
git add <file1> <file2> ...     #add one or more new OR modified file(s) to the index
git rm <file1>       #remove <file1> from index AND delete from working tree; must run "git commit"
git rm <file1> --cached   #remove <file1> from index ONLY; content must match HEAD or workspace file
```

git commit—update local repository with new/modified/deleted/renamed files

```
git commit      #commit contents of index to the repository; opens default editor for commit message
```

Note: entering commit message and saving carries out commit; empty message aborts commit; "#" lines are ignored

```
git commit -m"<Description of changes>"   #commit contents of index to the repository
git commit -v   #commit index to the repository and show differnces as you commit
git commit -a   #commit ALL modified files (inside or outside index) to repository; ignores new files
git commit --amend   #redo last commit; adds new/modified files in index; you can modify commit message
```

git log—show commit logs

```
git log         #see log of commits; "-v" paginates log; "--stat" adds graph of changes; "-<n>" limits no. shown
git log --graph --pretty=oneline      #summary of commits with ASCII graph of repository tree
git log <file1>     #see only the commits that affect <file1>
git log -p      #see log of commits with list of changes at each step in diff format, i.e., create a "patch file"
git ls-files    #simple list of which files have been committed
gitk            #visual repository and index browser; "-n 3" shows last 3 commits; "--all" shows all branches
```

git diff—show changes between commits, commit and working tree, etc., in one overall list

```
git diff            #show changed/modified files in working tree not yet added to index
git diff --cached   #show files in index
```

```
git diff HEAD          #show differences between working tree and HEAD, ignoring index
git diff <branch>      #show files in working tree compared with tip of <branch> tree
git diff HEAD^ HEAD    #compare version before the last commit with the last commit
git diff <branch1> <branch2>   #compare tips of <branch1> and <branch2>
```

git branch—list, create, or delete branches (Note: branching is fast, cheap, uses few resources)

```
git branch             #list local branches; "*" denotes the active branch
git branch -r          #list remote tracking branches; "-a" = list ALL (local and remote tracking) branches
git branch <name>      #create a new branch <name>
git branch -d <name>   #delete branch <name>
```

Tagging: Two types of tags: "lightweight" (name and associated hash) and "heavyweight" (permanent)

Note: Lightweight tags are easy to move and rename. You can optionally specify the hash of the desired object

Note: Heavyweight tags are objects in the object tree and can be GPG signed, allowing secure tracking of objects

```
git tag                #list all existing tags
git tag -l <pattern>   #list existing tags that match <pattern>
git tag <name>         #create a lightweight tag linked to current commit (or other specified object)
git tag -a <name>      #create a heavyweight tag linked to current commit; object is annotated with message
git tag -F <file> <name>  #create a heavyweight tag linked to current commit; take annotate text from <file>
git tag -s <name>      #create a GPG signed heavyweight tag linked to current commit
git tag -v <name>      #verify GPG signed tag; requires public key of signer in your keyring
git tag -d <name>      #delete tag <name>; can list more than one <name>
git tag -f <name>      #move existing tag <name> to refer to a new object
git show <name>        #show information about tag and its associated object
```

Working with remote repositories; "origin" is name of default remote repository

```
git remote -v                  #list remotes currently being tracked; "-v" shows URLs
git remote add <name> <url>    #begin tracking repository at <url>, referring to it as <name>
git remote rename <old> <new>  #change local name from <old> to <new>
git remote rm <name>           #remove all local tracking branches and references to <name>
git remote set-url <name> <new-url>     #change the url where <name> points to <new-url>
git show <name>                #display information about remote <name>
git fetch <name>               #update local copy of remote branch, but don't merge
git fetch --all                #update local copies of all remote branches but don't merge
git merge <branch>             #merge <branch> into current local repository; requires commit message
```

Note: Be sure to commit local work before attempting a merge or pull

Note: Merge tools like KDiff3, tkdiff, vimdiff, araxis, etc., are invaluable in resolving conflicts. Choose with git mergetool

Note: If merging or pulling from a new HEAD within same branch, git will auto-update, ie, "fast forward", unless "--no-ff"

```
git merge --continue           #run after fixing first conflict to move onto the second, third, etc.
git merge --abort              #abandon merge and clean up
git pull <name>                #run fetch and then merge from <name>
git push <name>                #push changes to remote repository; --tag option includes tag names
```

Stash—"hide" recent changes in separate, local-only commit and revert to HEAD. See "git help stash"

```
git stash       #create a new commit in separate directory (refs/stash/); restore workspace to previous commit
git stash list  #list all stashes, e.g., stash@{0} <branch name> <commit info>
git stash pop   #reverse of "git stash": remove stash and move to workspace; can optionally specify stash name
git stash apply #move stash to workspace but perserve stash; good way to copy stash to multiple branches
```

Undo existing commits, etc. (First use "-n" option to show actions without carrying them out)

```
git revert <commit-name>        #create a new commit that reverts back to <commit-name>
git reset          #remove one or more commits; default is HEAD (one commit); "--hard" removes all traces
git clean          #remove all untracked files from workspace
```

Submodules: external repositories within existing project (Note: external updates are NOT tracked)

```
git submodule       #list existing submodules; "-" at start of hash = submodule not initialized; "+" = new changes
git submodule add <url> <name>  #add submodule at <url> to project under <name>; then detach from url
git submodule init <name>       #initialize submodule; must then commit to load ".gitmodules" and <name>
```

Cloning an external repository that itself contains submodules

```
git clone <url> <name>        #clone main repository
git submodule                 #list submodules; submodule directories will be empty and uninitialized
git submodule init <name>     #Initialize EACH submodule by name
git submodule update <name> #download/clone contents of each submodule; then run git commit
```

Updating a submodule to a newer version of an external repository

```
git checkout <branch>         #run from within submodule directory <name>
git add <name>                #run in main directory to put updated submodule in Index; run git commit
```

Other useful commands

```
git mv <file1> <file2>       #rename file1 as file2 in working directory and index; must commit changes
git mv file1 file2 dir       #MOVE one or more files to (preexisting directory) dir/,  then commit changes
git rebase master            #ensure changes in master since your last checkout appear in your branch
git blame <file>        #show who added what and when in a file (<file> could also be a tag-name, HEAD, etc.)
git show       #show various types of objects; very rich set of options; see "git help show -w" for details
git grep       #print lines in workspace, index, or repo matching a pattern; see "git help grep -w"
git notes      #add, remove, or show "sticky note" attached to an object; subcommands are add, list, show, etc.
git fsck       #check integrity and connectivity of objects in the database
git bisect     #tool to search for where a newly discovered bug was introduced; use "git bisect help" to start
git gui        #run the official Git GUI interface
```