Git Notes for professionals:

1.3 Sharing Code:

To share your code you create a repository on a remote server to which you will copy your local repository. To minimize the use of space on the remote server you create a bare repository:

**git init** --bare **/**path**/**to**/**repo.git

Now copy your local repository to the remote:

**git push** --set-upstream origin master

Adding --set-upstream (or -u) created an upstream (tracking) reference which is used by argument-less Gitcommands, e.g. **git pull**.

1.4 Settings

You need to **set who** you are **\***before**\*** creating any commit

To declare that identity for *all* repositories, use **git config** --global

This will store the setting in your user's .gitconfig file: **%**USERPROFILE**%**\.gitconfig for Windows

**git config** --global user.name "Your Name"

**git config** --global user.email mail**@**example.com

To declare an identity for a single repository, use **git config** inside a repo.

This will store the setting inside the individual repository, in the file **/**path**/**to**/**your**/**repo**/**.git**/**config

**git config** user.name "Your Login At Work"

**git config** user.email [mail\_at\_work**@**example.com](mailto:mail_at_work@example.com)

Settings stored in a repository's config file will take precedence over the global config when you use that repository.

Section 1.7 Setup SSH for git

Create a new public and private key pair with the following command

$ ssh-keygen

2.2 Prettier log

To see the log in a prettier graph-like structure use:

**git log** --decorate --oneline --graph

Since it's a pretty big command, you can assign an alias:

**git config** --global alias.lol "log --decorate --oneline --graph"

**git log** --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)

%C(yellow)<%an>%Creset'

The format option allows you to specify your own log output format:

**Parameter Details**

**%**C**(**color\_name**)** option colors the output that comes after it

%h or %H abbreviates commit hash (use %H for complete hash)

**%**Creset resets color to default terminal color

%d ref names

%s subject [commit message]

**%**cr committer date, relative to current date

**%**an author name

Searches for **addition** or **removal** of specific string or the string **matching** provided

git log -S"#define SAMPLES"

Searches for **changes** in **lines containing** specific string or the string **matching**

git log -G"#define SAMPLES"

**git shortlog** summarizes **git log** and groups by author

To simply see the number of commits and suppress the commit description, pass in the summary option

$ **git shortlog** –s

Searching commit string

**git log [**options**]** --grep "search\_string"

Log for a range of lines whithin a file.

$ **git log** -L 1,20:index.html

Filter logs

**git log** --after '3 days ago'

**git log** --after 2016-05-01

An alias to --after is --since.

Flags exist for the converse too: --before and --until.

You can also filter logs by author. e.g.

**git log** --author=author

log showing committed files

**git log** –stat

2.12 Show the contents of a single commit

**git show** 48c83b3

2.13 Git log between tow branches

**git log** master..foo

**git log**  --decorate --source --pretty=format:'%h %ar %an %d %s"' --all --graph

Section3.1 Deleting a remote branch

To delete a remote branch in Git:

**git push [**remote-name**]** --delete **[**branch-name**]**

**or**

**git push [**remote-name**]** :**[**branch-name**]**

Section3.2 Changing git remote url

**git remote** set-url origin https:**//**github.com**/**username**/**repo2.git

Section3.4 Removing local copies of deleted remote branches

To prune deleted branches from a specific remote:

**$ git remote** [remote-name] --prune

To prune deleted branches from all remotes

**$ git remote** --all --prune

The pull command combines a fetch and a merge

The pull with –rebase flag command combines a fetch and a rebase

Section 3.6 ls-remote

**$ git ls-remote**

Is one unique command allowing you to query a remote repo without having to clone/fetch it first. It will list refs/heads and refs/tags of said remote repo.

**$ git remote** --refs

**$ git remote** --tags

Section 3.8 Set Upstream on a new branch

You can create a new branch and switch to it using

**$ git checkout** –b mybranch

**$ git push** –u origin mybranch

Section 3.10 Renaming a remote

**$ git remote** –rename origin destination

Section 3.10 Show information about a specific remote

**$ git remote show**  origin

Section 3.12

**Section 3.10 Renaming a remote**

To rename remote, use command **git remote** rename

**$ git remote** –rename origin destination

Get existing remote name

**git remote**

Check existing remote with URL

**git remote** –v

**Section 3.11 Show information about a specific remote**

Output some information about a known remote: origin

**$ git remote show**  origin

Print just the remote's URL:

**git remote** get-url origin

You can obtain the url for an existing remote by using the command

**git remote** get-url **<**name**>**

**Section 3.12 Set the url for a specific remote**

You can change the url of an existing remote by the command

**git remote** set-url remote-name url

**Section 4.1 Staging all changes to files**

**git add** .

In version 2.x, **git add** . will stage all changes to files in the current directory and all its subdirectories. However, in 1.x it will only stage new and modified files, not deleted files.

**Section 4.2 Unstage a file that contains changes**

**git reset <**filePath>

**Section 4.3: Add changes by hunk**

You can see what "hunks" of work would be staged for commit using the patch flag:

**git add** -p

This opens an interactive prompt that allows you to look at the diffs and let you decide whether you want to include them or not. You can also open this via **git add** --interactive and selecting p.

**git add** -i (or --interactive) will give you an interactive interface where you can edit the index, to prepare what you want to have in the next commit. You can add and remove changes to whole files, add untracked files and remove files from being tracked, but also select subsection of changes to put in the index, by selecting chunks of changes to be added, splitting those chunks, or even editing the diff.

**Section 4.5: Show Staged Changes**

To display the hunks that are staged for commit:

**git diff** --cached

**Section 4.7: Stage deleted files**

**git rm** filename

To delete the file from git without removing it from disk, use the --cached flag

**git rm** --cached filename

**Chapter 5: Ignoring Files and Folders**

This topic illustrates how to avoid adding unwanted files (or file changes) in a Git repo. There are several ways

**Section 5.1: Ignoring files and directories with a .gitignore file**

You can make Git ignore certain files and directories — that is, exclude them from being tracked by Git — by creating one or more .gitignore files in your repository.

In software projects, .gitignore typically contains a listing of files and/or directories that are generated during the build process or at runtime. Entries in the .gitignore file may include names or paths pointing to:

1. temporary resources e.g. caches, log files, compiled code, etc.

2. local configuration files that should not be shared with other developers

3. files containing secret information, such as login passwords, keys and credentials

When created in the top level directory, the rules will apply recursively to all files and sub-directories throughout the entire repository. When created in a sub-directory, the rules will apply to that specific directory and its subdirectories.

When a file or directory is ignored, it will not be:

1. tracked by Git

2. reported by commands such as **git status** or **git diff**

3. staged with commands such as **git add** -A

*# Ignore files called 'file.ext'*

file.ext

*# Ignoring files with full path.*

*# This matches files in the root directory and subdirectories too.*

*# i.e. otherfile.ext will be ignored anywhere on the tree.*

dir**/**otherdir**/**file.ext

otherfile.ext

*# Ignoring directories*

*# Both the directory itself and its contents will be ignored.*

bin**/**

gen**/**

*# Glob pattern can also be used here to ignore paths with certain characters.*

*# For example, the below rule will match both build/ and Build/*

**[**bB**]**uild**/**

*# Without the trailing slash, the rule will match a file and/or*

*# a directory, so the following would ignore both a file named `gen`*

*# and a directory named `gen`, as well as any contents of that directory*

bin

gen

*# Ignoring files by extension*

*# All files with these extensions will be ignored in this directory and all its sub-directories.*

**\***.apk

**\***.class

*# It's possible to combine both forms to ignore files with certain*

*# extensions in certain directories. The following rules would be*

*# redundant with generic rules defined above.*

java**/\***.apk

gen**/\***.class

*# To ignore files only at the top level directory, but not in its*

*# subdirectories, prefix the rule with a `/`*

**/\***.apk

**/\***.class

*# To ignore any directories named DirectoryA*

*# in any depth use \*\* before DirectoryA*

*# Do not forget the last /,*

*# Otherwise it will ignore all files named DirectoryA, rather than directories*

**\*\*/**DirectoryA**/**

*# This would ignore*

*# DirectoryA/*

*# DirectoryB/DirectoryA/*

*# DirectoryC/DirectoryB/DirectoryA/*

*# It would not ignore a file named DirectoryA, at any level*

*# To ignore any directory named DirectoryB within a*

*# directory named DirectoryA with any number of*

*# directories in between, use \*\* between the directories*

DirectoryA**/\*\*/**DirectoryB**/**

*# This would ignore*

*# DirectoryA/DirectoryB/*

*# DirectoryA/DirectoryQ/DirectoryB/*

*# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/*

*# To ignore a set of files, wildcards can be used, as can be seen above.*

*# A sole '\*' will ignore everything in your folder, including your .gitignore file.*

*# To exclude specific files when using wildcards, negate them.*

*# So they are excluded from the ignore list:*

**!**.gitignore

*# Use the backslash as escape character to ignore files with a hash (#)*

*# (supported since 1.6.2.1)*

\*#\*#*

**Cleaning up ignored files**

You can use **git clean** -X to cleanup ignored files:

**git clean** -Xn *#display a list of ignored files*

**git clean** -Xf *#remove the previously displayed files*

Note: -X (caps) cleans up *only* ignored files. Use -x (no caps) to also remove untracked files.

The **git** check-ignore command reports on files ignored by Git. You can pass filenames on the command line, and **git** check-ignore will list the filenames that are ignored. For example:

$ **cat** .gitignore

**\***.o

$ **git** check-ignore example.o Readme.md

example.o

If you ignore files by using a pattern but have exceptions, prefix an exclamation mark(!) to the exception. For example:

**\***.txt

**!**important.txt

The above example instructs Git to ignore all files with the .txt extension except for files named important.txt.

===========================🡺 A completer avec doc maison

.gitignore ignores files locally, but it is intended to be committed to the repository and shared with other

contributors and users.

**Section 5.6: Ignore files locally without committing ignore rules**

Sometimes you want to have a file held in Git but ignore subsequent changes. Tell Git to ignore changes to a file or directory using update-index:

**git update-index** --assume-unchanged my-file.txt

The above command instructs Git to assume my-file.txt hasn't been changed, and not to check or report changes. The file is still present in the repository. This can be useful for providing defaults and allowing local environment overrides,

**Section 5.8: Ignoring a file in any directory**

To ignore a file foo.txt in **any** directory you should just write its name:

foo.txt *# matches all files 'foo.txt' in any directory*

If you want to ignore the file only in part of the tree, you can specify the subdirectories of a specific directory with \*\* pattern:

bar**/\*\*/**foo.txt *# matches all files 'foo.txt' in 'bar' and all subdirectories*

If you are unsure which rules to list in your .gitignore file, or you just want to add generally accepted exceptions to your project, you can choose or generate a .gitignore file:

https://www.gitignore.io/

https://github.com/github/gitignore

**Section 5.11: Create an Empty Folder**

is not possible to add and commit an empty folder in Git due to the fact that Git manages *files* and attaches their directory to them, which slims down commits and improves speed. To get around this, there are two methods:

One hack to get around this is to use a .gitkeep file to register the folder for Git. To do this, just create the required directory and add a .gitkeep file to the folder. This file is blank and doesn't serve any purpose other than to just register the folder. To do this in Windows (which has awkward file naming conventions) just open git bash in the directory and run the command:

$ touch .gitkeep

This command just makes a blank .gitkeep file in the current directory

**Section 5.12: Finding files ignored by .gitignore**

You can list all files ignored by git in current directory with command:

**git status** --ignored

If you want to list recursively ignored files in directories, you have to use additional parameter - --untrackedfiles=all

$ **git status** --ignored --untracked-files=all

**Section 6.1: Show differences in working branch**

**git diff**

This will show the *unstaged* changes on the current branch from the commit before it. It will only show changes relative to the index, meaning it shows what you *could* add to the next commit, but haven't.

To add (stage) these changes, you can use **git add**. If a file is staged, but was modified after it was staged, **git diff** will show the differences between the current file and the staged version.

**Section 6.2: Show changes between two commits**

**git diff** 1234abc..6789def *# old new*

Show the changes made in the last 3 commits:

**git diff @**~3..**@** *# HEAD -3 HEAD*

Note: the two dots (..) is optional, but adds clarity. This will show the textual difference between the commits, regardless of where they are in the tree.

**Section 6.3: Show differences for staged files**

**git diff** --staged

This will show the changes between the previous commit and the currently staged files.

**NOTE:** You can also use the following commands to accomplish the same thing:

**git diff** --cached

Which is just a synonym for --staged or

**git status** -v

Which will trigger the verbose settings of the status command.

**Section 6.4: Comparing branches**

Show the changes between the tip of **new** and the tip of **original**:

**git diff** original new *# equivalent to original..new*

Show all changes on **new** since it branched from **original**:

**git diff** original...new *# equivalent to $(git merge-base original new)..new*

Using only one parameter such as

git diff original

is equivalent to

git diff original..HEAD

**Section 6.5: Show both staged and unstaged changes**

To show all staged *and* unstaged changes, use:

**git diff** HEAD

**NOTE:** You can also use the following command:

**git status** -vv

The difference being that the output of the latter will actually tell you which changes are staged for commit and which are not.

**Section 6.6: Show differences for a specific file or directory**

**Section 6.6: Show dierences for a specific file or directory**

**git diff** myfile.txt

Shows the changes between the previous commit of the specified file (myfile.txt) and the locally-modified version that has not yet been staged.

This also works for directories:

**git diff** documentation

The above shows the changes between the previous commit of all files in the specified directory (documentation**/**) and the locally-modified versions of these files, that have not yet been staged. To show the difference between some version of a file in a given commit and the local HEAD version you can specify the commit you want to compare against:

**git diff** 27fa75e myfile.txt

Or if you want to see the version between two separate commits:

**git diff** 27fa75e ada9b57 myfile.txt

To show the difference between the version specified by the hash ada9b57 and the latest commit on the branch my\_branchname for only the relative directory called my\_changed\_directory**/** you can do this:

**git diff** ada9b57 my\_branchname my\_changed\_directory**/**

**git diff [**HEAD**|**--staged...**]** --word-diff

Rather than displaying lines changed, this will display differences within lines. For example, rather than:

-Hello world

+Hello world**!**

Where the whole line is marked as changed, word-diff alters the output to:

Hello **[**-world-**]{**+world**!**+**}**

You can omit the markers [-, -], {+, +} by specifying --word-diff=color or --color-words. This will only use color coding to mark the difference:

git diff @~1 @ --word-diff=color

**git diff** HEAD^ HEAD

This will show the changes between the previous commit and the current commit

To view difference between two branch or commit

**git diff <**branch1**/**commitId1**>..<**branch2**/**commitId2**>**

To view diff with current branch

**git diff <**branch**/**commitId**>**

To view summary of changes

**git diff** --stat **<**branch**/**commitId**>**

To view files that changed after a certain commit

**git diff** --name-only **<**commitId**>**

To view files that are different than a branch

**git diff** --name-only **<**branchName**>**

To view files that changed in a folder after a certain commit

**git diff** --name-only **<**commitId**> <**folder\_path>

**Section 6.11: Using meld to see all modifications in the working directory**

**git difftool** -t meld --dir-diff

will show the working directory changes. Alternatively,

**git difftool** -t meld --dir-diff **[**COMMIT\_A**] [**COMMIT\_B**]**

will show the differences between 2 specific commits.

**Chapter 7: Undoing**

**Section 7.1: Return to a previous commit**

To temporarily jump back to that commit, detach your head with:

**git checkout** 789abcd

This places you at commit 789abcd. You can now make new commits on top of this old commit without affecting the branch your head is on. Any changes can be made into a proper branch using either branch or checkout -b.

To roll back the ***last*** commit:

**git reset** --soft HEAD~

To permanently discard any changes made after the ***last*** commit:

**git reset** --hard HEAD~

**Section 7.2: Undoing changes**

Undo changes to a file or directory in the **working copy**.

**git checkout** -- file.txt

Used over all file paths, recursively from the current directory, it will undo all changes in the working copy.

**git checkout** -- .

To undo changes added to the **index**.

**git reset** --hard

Without the --hard flag this will do a soft reset

Assume you want to undo a dozen of commits and you want only some of them.

**git rebase** -i **<**earlier SHA**>**

**Section 8.4: Merge with a commit**

Default behaviour is when the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. Use --no-ff to resolve.

**git merge <**branch\_name**>** --no-ff -m "<commit message>"

**Section 8.5: Keep changes from only one side of a merge**

During a merge, you can pass --ours or --theirs to **git checkout** to take all changes for a file from one side or the other of a merge

$ **git checkout** --ours -- file1.txt *# Use our version of file1, delete all their changes*

$ **git checkout** --theirs -- file2.txt *# Use their version of file2, delete all our changes*

**Chapter 9: Submodules**

**Section 9.1: Cloning a Git repository having submodules**

When you clone a repository that uses submodules, you'll need to initialize and update them.

$ **git clone** --recursive https:**//**github.com**/**username**/**repo.git

This will clone the referenced submodules and place them in the appropriate folders (including submodules within submodules). This is equivalent to running **git submodule** update --init --recursive immediately after the clone is finished.

A submodule references a specific commit in another repository. To check out the exact state that is referenced for all submodules, run

**git submodule** update –recursive

To checkout the latest state of a specific submodule, you can use :

**git submodule** update --remote **<**submodule\_directory**>**

**Section 9.3: Adding a submodule**

You can include another Git repository as a folder within your project, tracked by Git:

$ **git submodule** add [https:**//**github.com**/**jquery**/**jquery.git](https://github.com/jquery/jquery.git)

To update that submodule to the latest commit of a branch of the submodule remote repo.

**git submodule** update --remote –recursive

**Section 9.5: Moving a submodule**

Run:

$ **git mv** /path/to/module *new/path/to/module*

**Section 9.6: Removing a submodule**

You can remove a submodule (e.g. the\_submodule) by calling:

$ **git submodule** deinit the\_submodule

$ **git rm** the\_submodule

**git submodule** deinit the\_submodule deletes the\_submodules' entry from .git/config. This excludes

the\_submodule from **git submodule** update, **git submodule sync** and **git submodule** foreach calls and

deletes its local content (source). Also, this will not be shown as change in your parent repository. **git**

**submodule** init and **git submodule** update will restore the submodule, again without commitable changes

in your parent repository.

**git rm** the\_submodule will remove the submodule from the work tree. The files will be gone as well as the

submodules' entry in the .gitmodules file (source). If only **git rm** the\_submodule (without prior **git**

**submodule** deinit the\_submodule is run, however, the submodules' entry in your .git/config file will remain.

**Chapter 10: Committing**

**Shortcuts**

If you have changed a lot of files in the directory, rather than listing each one of them, you could use:

**git add** --all *# equivalent to "git add -a"*

Or to add all changes, *not including files that have been deleted*, from the top-level directory and subdirectories:

**git add** .

Or to only add files which are currently tracked ("update"):

**git add** -u

If desired, review the staged changes:

**git status** *# display a list of changed files*

**git diff** --cached *# shows staged changes inside staged files*

Finally, commit the changes:

**git commit** -m "Commit message here"

Alternately, if you have only modified existing files or deleted files, and have not created any new ones, you can combine the actions of **git add** and **git commit** in a single command:

**git commit** -am "Commit message here"

Note that this will stage **all** modified files in the same way as **git add** –all

**Section 10.3: Amending a commit**

To specify the commit message inline:

**git commit** --amend -m "New commit message"

Or to use the previous commit message without changing it:

**git commit** --amend --no-edit

Amending updates the commit date but leaves the author date untouched. You can tell git to refresh the information.

**git commit** --amend --reset-author

You can also change the author of the commit with:

**git commit** --amend --author "New Author <email@address.com>"

You don't necessarily need to commit all files at once. Omit the -a or --all flag and specify which file you want to commit directly:

**git commit** path**/**to**/**a**/file** -m "your commit message goes here"

For directly committing more than one specific file, you can specify one or multiple files, directories and patterns as well:

**git commit** path**/**to**/**a**/file** path**/**to**/**a**/**folder**/\*** path**/**to**/**b**/file** -m "your commit message goes here"

Suppose you have many changes in one or more files but from each file you only want to commit some of the changes, you can select the desired changes using:

**git add** -p

or

**git add** -p **[file]**

Generally speaking, empty commits (or commits with state that is identical to the parent) is an error.

The --allow-empty commit will bypass the check.

**git commit** -m "This is a blank commit" --allow-empty

If someone else wrote the code you are committing, you can give them credit with the --author option:

**git commit** -m "msg" --author "John Smith <johnsmith@example.com>"

You can also provide a pattern, which Git will use to search for previous authors:

**git commit** -m "msg" --author "John"

**Section 11.1: Simple aliases**

There are two ways of creating aliases in Git:

with the ~**/**.gitconfig file:

**[alias]**

ci = commit

st = status

**co** = checkout

with the command line:

**git config** --global alias.ci "commit"

**git config** --global alias.st "status"

**git config** --global alias.co "checkout"

**Section 11.4: Temporarily ignore tracked files**

To temporarily mark a file as ignored (pass file as parameter to alias) - type:

unwatch = update-index --assume-unchanged

To start tracking file again - type:

**watch** = update-index --no-assume-unchanged

**git** unwatch my\_file.txt

**git watch** my\_file.txt

**Section 11.5: Show pretty log with branch graph**

**[alias]**

logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg = log --graph --date-order --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

lgb = log --graph --date-order --branches --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

lga = log --graph --date-order --all \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

Here an explanation of the options and placeholder used in the --pretty format (exhaustive list are available with **git help** log )

--graph - draw the commit tree

--date-order - use commit timestamp order when possible

--first-parent - follow only the first parent on merge node.

--branches - show all local branches (by default, only current branch is shown)

--all - show all local and remotes branches

%h - hash value for commit (abbreviated)

%ad - Date stamp (author)

%an - Author username

%an - Commit username

%C(auto) - to use colors defined in [color] section

%Creset - to reset color

%d - --decorate (branch & tag names)

%s - commit message

%ad - author date (will follow --date directive) (and not commiter date)

%an - author name (can be %cn for commiter name

**Rebasing** reapplies a series of commits on top of another commit. To rebase a branch, checkout the branch and then rebase it on top of another branch.

**git checkout** topic

**git rebase** master

After the rebase, the applied commits will have a different hash. You should not rebase commits you

have already pushed to a remote host. A consequence my be an inability to **git push** your local rebased branch to a remote host, leaving your only option to **git push** --force.

**Section 12.2: Rebase: ours and theirs, local and remote**

A rebase switches the meaning of "ours" and "theirs":

**git checkout** topic

**git rebase** master *# rebase topic branch on top of master branch*

The first thing a rebase does is resetting the HEAD to master; before cherry-picking commits from the old branch topic to a new one (every commit in the former topic branch will be rewritten and will be identified by a different hash).

With respect to terminologies used by merge tools (not to be confused with local ref or remote ref)

=**> local** is master **(**"ours"**)**,

=**>** remote is topic **(**"theirs"**)**

That means a merge/diff tool will present the upstream branch as **local** (master: the branch on top of which you are rebasing), and the working branch as remote (topic: the branch being rebased)

**On a merge:**

c--c--x--x--x(\*) <- current branch topic ('\*'=HEAD)

\\

\--y--y--y <- other branch to merge

We don't change the current branch topic, so what we have is still what we were working on (and we merge from another branch)

c--c--x--x--x---------o(\*) MERGE, still on branch topic

\ ^ /

\ ours /

\ /

--y--y--y--/

^

Theirs

**On a rebase:**

But **on a rebase** we switch sides because the first thing a rebase does is to checkout the upstream branch to replay the current commits on top of it!

c--c--x--x--x(\*) <- current branch topic ('\*'=HEAD)

\

\

\--y--y--y <- upstream branch

A **git rebase upstream** will first set HEAD to the upstream branch, hence the switch of 'ours' and 'theirs' compared to the previous "current" working branch.

c--c--x--x--x <- former "current" branch, new "theirs"

\

\

\--y--y--y(\*) <- set HEAD to this commit, to replay x's on it .this will be the new "ours"

^

|

upstream

The rebase will then replay 'their' commits on the new 'our' topic branch:

The rebase will then replay 'their' commits on the new 'our' topic branch:

c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"

\

\

\--y--y--y--x'--x'--x'(\*) <- topic once all x's are replayed, point branch topic to this commit

^

|

upstream branch

**Section 12.3: Interactive Rebase**

This example aims to describe how one can utilize **git rebase** in interactive mode. It is expected that one has a basic understanding of what **git rebase** is and what it does.

Interactive rebase is initiated using following command:

**git rebase** -i

The -i option refers to *interactive mode*. Using interactive rebase, the user can change commit messages, as well as reorder, split, and/or squash (combine to one) commits.

Say you want to rearrange your last three commits. To do this you can run:

**git rebase** -i HEAD~3

- **Rewording commit messages**

**- Changing the content of a commit**

- **Squashing multiple commits into one**

You have started an interactive rebase. In the editor where you pick your commits, you decide that something is going wrong (for example a commit is missing, or you chose the wrong rebase destination), and you want to abort the rebase.

To do this, simply delete all commits and actions (i.e. all lines not starting with the # sign) and the rebase will be aborted!

Sometimes you need rewrite history with a rebase, but **git push** complains about doing so because you rewrote history.

**Section 12.10: Pushing after a rebase**

Sometimes you need rewrite history with a rebase, but **git push** complains about doing so because you rewrote history.

This can be solved with a **git push** --force, but consider **git push** --force-with-lease, indicating that you want the push to fail if the local remote-tracking branch differs from the branch on the remote, e.g., someone else pushed to the remote after the last fetch. This avoids inadvertently overwriting someone else's recent push.

**Note**: **git push** --force - and even --force-with-lease for that matter - can be a dangerous command because it rewrites the history of the branch. If another person had pulled the branch before the forced push, his/her **git pull** or **git fetch** will have errors because the local history and the remote history are diverged. This may cause the person to have unexpected errors. With enough looking at the reflogs the other user's work can be recovered, but it can lead to a lot of wasted time. If you must do a forced push to a branch with other contributors, try to coordinate with them so that they do not have to deal with errors.

**Chapter 13: Configuration**

Change the core.editor configuration setting.

$ **git config** --global core.editor **notepad**

**Section 13.2: Auto correct typos**

**git config** --global help.autocorrect 17

This enables autocorrect in git and will forgive you for your minor mistakes (e.g. **git** stats instead of **git status**).

The parameter you supply to help.autocorrect determines how long the system should wait, in tenths of a second, before automatically applying the autocorrected command. In the command above, 17 means that git should wait 1.7 seconds before applying the autocorrected command.

Git config allows you to customize how git works. It is commonly used to set your name and email or favorite editor or how merges should be done.

To see the current configuration.

$ **git config** --list

...

core.editor=**vim**

credential.helper=osxkeychain

...

To edit the config:

$ **git config <**key**> <**value**>**

$ **git config** core.ignorecase **true**

If you intend the change to be true for all your repositories, use --global

$ **git config** --global user.name "Your Name"

$ **git config** --global user.email "Your Email"

$ **git config** --global core.editor **vi**

You can list again to see your changes.

**Section 13.4: Username and email address**

Right after you install Git, the first thing you should do is set your username and email address. From a shell, type:

**git config** --global user.name "Mr. Bean"

**git config** --global user.email mrbean**@**example.com

You have 3 sources for git configuration:

* (system) **<git>/etc/gitconfig**, with **<git>** being the git installation path.(on Windows, it is **<git>\mingw64\etc\gitconfig**)
* (global) ~**/**.gitconfig (Windows: **%**USERPROFILE**%**\.gitconfig)
* (local) .git**/**config (within a git repo $GIT\_DIR)

The order is important: any config set in one source can be overridden by a source listed below it. **git config** --system**/**global**/local** is the command to list 3 of those sources, but only git config -l would list *all resolved* configs.

Since git 2.8, if you want to see which config comes from which file, you type:

**git config** --list --show-origin

**Section 13.8: configuration for one command only**

you can use -c **<**name**>**=**<**value**>** to add a configuration only for one command.

To commit as an other user without having to change your settings in .gitconfig :

**git** -c user.email = mail**@**example commit -m "some message"

Note: for that example you don't need to precise both user.name and user.email, git will complete the missing information from the previous commits.

**Section 14.1: Creating and checking out new branches**

To create a new branch, while staying on the current branch, use:

**git branch <**name**>**

Generally, the branch name must not contain spaces and is subject to other specifications listed here. To switch to an existing branch :

**git checkout <**name**>**

To create a new branch and switch to it:

**git checkout** -b **<**name**>**

To create a branch at a point other than the last commit of the current branch (also known as HEAD), use either of these commands:

**git branch <**name**> [<**start-point**>]**

**git checkout** -b **<**name**> [<**start-point**>]**

The **<start-point>** can be any revision known to git (e.g. another branch name, commit SHA, or a symbolic reference such as HEAD or a tag name):

**git checkout** -b **<**name**>** some\_other\_branch

**git checkout** -b **<**name**>** af295

**git checkout** -b **<**name**>** HEAD~5

**git checkout** -b **<**name**>** v1.0.5

To create a branch from a remote branch (the default **<remote\_name>** is origin):

**git branch <**name**> <**remote\_name**>/<**branch\_name**>**

**git checkout** -b **<**name**> <**remote\_name**>/<**branch\_name**>**

If a given branch name is only found on one remote, you can simply use

**git checkout** -b **<**branch\_name**>**

which is equivalent to

**git checkout** -b **<**branch\_name**> <**remote\_name**>/<**branch\_name**>**

Sometimes you may need to move several of your recent commits to a new branch. This can be achieved by branching and "rolling back", like so:

**git branch <**new\_name**>**

**git reset** --hard HEAD~2 *# Go back 2 commits, you will lose uncommitted work.*

**git checkout <**new\_name**>**



You can quickly switch to the previous branch using

**git checkout** -

**Section 14.2: Listing branches**

Git provides multiple commands for listing branches. All commands use the function of **git branch**, which will provide a list of a certain branches, depending on which options are put on the command line.

List local branches **git branch**

List local branches verbose **git branch** -v

List remote and local branches **git branch** -a OR **git branch** --all

List remote and local branches (verbose) **git branch** -av

List remote branches **git branch** -r

List remote branches with latest commit **git branch** -rv

List merged branches **git branch** --merged

List unmerged branches **git branch** --no-merged

List branches containing commit **git branch** --contains **[<**commit**>]**

* Adding an additional v to -v e.g. $ **git branch** -avv or $ **git branch** -vv will print the name of the upstream branch as well.
* Branches shown in red color are remote branches

**Section 14.3: Delete a remote branch**

To delete a branch on the origin remote repository, you can use for Git version 1.5.0 and newer

**git push** origin :**<**branchName**>**

and as of Git version 1.7.0, you can delete a remote branch using

**git push** origin --delete **<**branchName**>**

To delete a local remote-tracking branch:

**git branch** --delete --remotes **<**remote**>/<**branch**>**

**git branch** -dr **<**remote**>/<**branch**>** *# Shorter*

**git fetch <**remote**>** --prune *# Delete multiple obsolete tracking branches*

**git fetch <**remote**>** -p *# Shorter*

To delete a branch locally. Note that this will not delete the branch if it has any unmerged changes:

**git branch** -d **<**branchName**>**

To delete a branch, even if it has unmerged changes:

**git branch** -D **<**branchName**>**

$ **git branch** -d dev

Deletes the branch named dev *if* its changes are merged with another branch and will not be lost. If the dev branch does contain changes that have not yet been merged that would be lost, **git branch** -d will fail:

$ **git branch** -d dev

Per the warning message, you can force delete the branch (and lose any unmerged changes in that branch) by using the -D flag:

$ **git branch** -D dev

**Section 14.5: Check out a new branch tracking a remote branch**

There are three ways of creating a new branch feature which tracks the remote branch origin**/**feature:

**git checkout** --track -b feature origin**/**feature,

**git checkout** -t origin**/**feature,

**git checkout** feature - assuming that there is no local feature branch and there is only one remote with

the feature branch.

To set upstream to track the remote branch - type:

**git branch** --set-upstream-to=**<**remote**>/<**branch**> <**branch**>**

**git branch** -u **<**remote**>/<**branch**> <**branch**>**

where:

**<remote>** can be: origin, develop or the one created by user,

**<branch>** is user's branch to track on remote.

To verify which remote branches your local branches are tracking:

**git branch** –vv

**Section 14.7: Create an orphan branch (i.e. branch with no parent commit)**

**git checkout** --orphan new-orphan-branch

**Section 14.8: Rename a branch**

Rename the branch you have checked out:

**git branch** -m new\_branch\_name

Rename another branch:

**git branch** -m branch\_you\_want\_to\_rename new\_branch\_name

**Section 14.9: Searching in branches**

To list local branches that contain a specific commit or tag

**git branch** --contains **<**commit**>**

To list local and remote branches that contain a specific commit or tag

**git branch** -a --contains **<**commit**>**

**Section 14.10: Push branch to remote**

Use to push commits made on your local branch to a remote repository.

The **git push** command takes two arguments:

A remote name, for example, origin

A branch name, for example, master

For example:

**git push <**REMOTENAME**> <**BRANCHNAME**>**

As an example, you usually run **git push** origin master to push your local changes to your online repository.

Using -u (short for --set-upstream) will set up the tracking information during the push.

**git push** -u **<**REMOTENAME**> <**BRANCHNAME**>**

By default, **git** pushes the local branch to a remote branch with the same name. For example, if you have a local called new-feature, if you push the local branch it will create a remote branch new-feature as well. If you want to use a different name for the remote branch, append the remote name after the local branch name, separated by ::

**git push <**REMOTENAME**> <**LOCALBRANCHNAME**>**:**<**REMOTEBRANCHNAME**>**

A branch is just a pointer to a commit, so you can freely move it around. To make it so that the branch is referring to the commit aabbcc, issue the command

**git reset** --hard aabbcc

Please note that this will overwrite your branch's current commit, and as so, its entire history. You might loose some work by issuing this command. If that's the case, you can use the reflog to recover the lost commits. It can be advised to perform this command on a new branch instead of your current one. However, this command can be particularly useful when rebasing or doing such other large history modifications.

**Section 15.1: List Commits in master but not in origin/master**

**git rev-list** --oneline master ^origin**/**master

Git rev-list will list commits in one branch that are not in another branch. It is a great tool when you're trying to

figure out if code has been merged into a branch or not.

Using the --oneline option will display the title of each commit.

The ^ operator excludes commits in the specified branch from the list.

You can pass more than two branches if you want. For example, **git rev-list** foo bar ^baz lists commits

in foo and bar, but not baz.

**Chapter 16: Squashing**

**Section 16.1: Squash Recent Commits Without Rebasing**

If you want to squash the previous x commits into a single one, you can use the following commands:

**git reset** --soft HEAD~x

**git commit**

Replacing x with the number of previous commits you want to be included in the squashed commit.

Mind that this will create a *new* commit, essentially forgetting information about the previous x commits including their author, message and date. You probably want to *first* copy-paste an existing commit message.

You can use **git merge** --squash to squash changes introduced by a branch into a single commit. No actual commit will be created.

**git merge** --squash **<**branch**>**

**git commit**

Commits can be squashed during a **git rebase**. It is recommended that you understand rebasing before attempting to squash commits in this fashion.

1. Determine which commit you would like to rebase from, and note its commit hash.

2. Run **git rebase** -i **[**commit **hash]**.

Alternatively, you can type HEAD~4 instead of a commit hash, to view the latest commit and 4 more commits before the latest one.

**Chapter 17: Cherry Picking**

A cherry-pick takes the patch that was introduced in a commit and tries to reapply it on the branch you’re currently on.

**git cherry-pick <**commit-hash**>** will apply the changes made in an existing commit to another branch, while recording a new commit. Essentially, you can copy commits from branch to branch

**git checkout** master

**git cherry-pick** b886a0

Where the new commit a66b23 has the same content (source diff, commit message) as b886a0 (but a different parent). Note that cherry-picking will only pick up changes on that commit (b886a0 in this case) not all the changes in feature branch (for this you will have to either use rebasing or merging).

**git cherry-pick <**commit-A**>**..**<**commit-B**>** will place every commit *after* A and up to and including B on top of the currently checked-out branch

**git cherry-pick <**commit-A**>**^..**<**commit-B**>** will place commit A and every commit up to and including B on top of the currently checked-out branch.

Before you start the cherry-pick process, you can check if the commit you want to cherry-pick already exists in the

target branch, in which case you don't have to do anything.

**git branch** --contains **<**commit**>** lists local branches that contain the specified commit.

**git branch** -r --contains **<**commit**>** also includes remote tracking branches in the list.

Command **git cherry** shows the changes which haven't yet been cherry-picked.

**git checkout** master

**git cherry** development

The commits that being with + will be the ones that haven't yet cherry-picked into development.

**Chapter 18: Recovering**

**Section 18.1: Recovering from a reset**

$ **git reset @**~3 *# go back 3 commits*

$ **git reflog**

**………..**

$ **git reset** 2c52489

**Section 18.2: Recover from git stash**

To get your most recent stash after running git stash, use

**git stash** apply

To see a list of your stashes, use

**git stash** list

You will get a list that looks something like this

stash**@{**0**}**: WIP on master: 67a4e01 Merge tests into develop

stash**@{**1**}**: WIP on master: 70f0d95 Add user role to localStorage on user **login**

Choose a different git stash to restore with the number that shows up for the stash you want

**git stash** apply stash**@{**2**}**

You can also choose 'git stash pop', it works same as 'git stash apply' like..

**git stash** pop

or

**git stash** pop stash**@{**2**}**

Difference in git stash apply and git stash pop...

**git stash pop**: stash data will be remove from stack of stash list.

**git stash** list

You will get a list that looks something like this

stash**@{**0**}**: WIP on master: 67a4e01 Merge tests into develop

stash**@{**1**}**: WIP on master: 70f0d95 Add user role to localStorage on user **login**

Now pop stash data using command

**git stash** pop

Again Check for stash list

**git stash** list

You will get a list that looks something like this

stash**@{**0**}**: WIP on master: 70f0d95 Add user role to localStorage on user **login**

You can see one stash data is removed (popped) from stash list and stash@{1} became stash@{0}.

**Section 18.3: Recovering from a lost commit**

In case you have reverted back to a past commit and lost a newer commit you can recover the lost commit by running

**git reflog**

Then find your lost commit, and reset back to it by doing

**git reset** HEAD --hard **<**sha1-of-commit**>**

**Section 18.4: Restore a deleted file after a commit**

In case you have accidentally commited a delete on a file and later realized that you need it back.

First find the commit id of the commit that deleted your file.

**git log** --diff-filter=D --summary

Will give you a sorted summary of commits which deleted files.

Then proceed to restore the file by

**git checkout** 81eeccf~1 **<**your-lost-file-name**>**

(Replace 81eeccf with your own commit id)

**Section 18.5: Restore file to a previous version**

To restore a file to a previous version you can use reset

**git reset <**sha1-of-commit**> <**file-name**>**

If you have already made local changes to the file (that you do not require!) you can also use the --hard option [Bizarre!!!!!!!!!!!]

**Section 18.6: Recover a deleted branch**

To recover a deleted branch you need to find the commit which was the head of your deleted branch by running

**git reflog**

You can then recreate the branch by running

**git checkout** -b **<**branch-name**> <**sha1-of-commit**>**

You will not be able to recover deleted branches if git's garbage collector deleted dangling commits - those without refs. Always have a backup of your repository, especially when you work in a small team / proprietary project

**Section 19.1: Clean Interactively**

**git clean** –i

Will print out items to be removed and ask for a confirmation via commands like the follow

**Forcefully remove untracked files**

**git clean** -f

Will remove all untracked files

**git clean** -fX

Will remove only ignored files from the current directory and all subdirectories.

**git clean** -fx

Will remove ignored files too from the current directory and all subdirectories.

**git clean** -Xn

Will preview all files that will be cleaned.

**Clean All Untracked Directories**

**git clean** -fd

Will remove all untracked directories and the files within them. It will start at the current working directory and will iterate through all subdirectories

**git clean** -dn

Will preview all directories that will be cleaned.

**Section 23.2: Updating with local changes**

When local changes are present, the **git pull** command aborts reporting :

error: Your local changes to the following files would be overwritten by merge

In order to update (like svn update did with subversion), you can run :

**git stash**

**git pull** --rebase

**git stash** pop

A convenient way could be to define an alias using :

Version < 2.9

**git config** --global alias.up '!git stash && git pull --rebase && git stash pop'

Version ≥ 2.9

**git config** --global alias.up 'pull --rebase --autostash'

Next you can simply use :

**git** up

If you are pulling in fresh commits from the remote repository and you have local changes on the current branch then git will automatically merge the remote version and your version. If you would like to reduce the number of merges on your branch you can tell git to rebase your commits on the remote version of the branch.

**git pull** –rebase

To only allow fast forwarding the local branch, you can use:

**git pull** --ff-only

This will display an error when the local branch is not fast-forwardable, and needs to be either rebased or merged with upstream.

Save the current state of working directory and the index (also known as the staging area) in a stack of stashes.

**git stash**

To include all untracked files in the stash use the --include-untracked or -u flags.

**git stash** --include-untracked

To include a message with your stash to make it more easily identifiable later

**git stash** save "<whatever message>"

To leave the staging area in current state after stash use the --keep-index or -k flags.

**git stash** --keep-index

To apply the last stash and remove it from the stack - type:

**git stash** pop

To apply specific stash and remove it from the stack - type:

**git stash** pop stash**@{**n**}**

Applies the last stash without removing it from the stack

**git stash** apply

Or a specific stash

**git stash** apply stash**@{**n**}**

Shows the changes saved in the last stash

**git stash** show

Or a specific stash

**git stash** show stash**@{**n**}**

To show content of the changes saved for the specific stash

**git stash** show -p stash**@{**n**}**

**git stash** list

This will list all stashes in the stack in reverse chronological order.

You will get a list that looks something like this:

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

**Section 26.8: Move your work in progress to another branch**

If while working you realize you're on wrong branch and you haven't created any commits yet, you can easily move

your work to correct branch using stashing:

**git stash**

**git checkout** correct-branch

**git stash** pop

Remove all stash

**git stash clear**

Removes the last stash

**git stash** drop

Or a specific stash

**git stash** drop stash**@{**n**}**

**Section 26.10: Apply part of a stash with checkout**

You've made a stash and wish to checkout only some of the files in that stash.

**git checkout** stash**@{**0**}** -- myfile.txt

To get your most recent stash after running git stash, use

**git stash** apply

To see a list of your stashes, use

**git stash** list

You will get a list that looks something like this

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

Choose a different git stash to restore with the number that shows up for the stash you want

**git stash** apply stash**@{**2**}**

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes and saves it on a stack of unfinished changes that you can reapply at any time.

**Stashing only modified files:**

Suppose you don't want to stash the staged files and only stash the modified files so you can use:

**git stash** --keep-index

Which will stash only the modified files.

**Stashing untracked files:**

Stash never saves the untracked files it only stashes the modified and staged files. So suppose if you need to stash the untracked files too then you can use this:

**git stash** -u

this will track the untracked, staged and modified files.

**Stash some particular changes only:**

Suppose you need to stash only some part of code from the file or only some files only from all the modified and

stashed files then you can do it like this:

**git stash** --patch

Git will not stash everything that is modified but will instead prompt you interactively which of the changes you

would like to stash and which you would like to keep in your working directory.

**Chapter 28: Renaming**

**Rename Folders**

**git mv** directoryToFolder**/**oldName directoryToFolder**/**newName

Followed by **git commit** and/or **git push**

**Section 28.2: rename a local and the remote branch**

the easiest way is to have the local branch checked out:

**git checkout** old\_branch

then rename the local branch, delete the old remote and set the new renamed branch as upstream:

**git branch** -m new\_branch

**git push** origin :old\_branch

**git push** --set-upstream origin new\_branch

**Section 28.3: Renaming a local branch**

You can rename branch in local repository using this command:

**git branch** -m old\_name new\_name

**Section 29.1: Push a specific object to a remote branch**

**General syntax**

**git push <**remotename**> <**object**>**:**<**remotebranchname**>**

**Example**

**git push** origin master:wip-yourname

Will push your master branch to the wip-yourname branch of origin (most of the time, the repository you cloned from).

**Delete remote branch**

Deleting the remote branch is the equivalent of pushing an empty object to it.

**git push <**remotename**>** :**<**remotebranchname**>**

**Example**

**git push** origin :wip-yourname

Will delete the remote branch wip-yourname

Instead of using the colon, you can also use the --delete flag, which is better readable in some cases.

**Example**

**git push** origin --delete wip-yourname

**Section 29.2: Push**

**git push**

will push your code to your existing upstream. Depending on the push configuration, it will either push code from you current branch (default in Git 2.x) or from all branches (default in Git 1.x).

**Specify remote repository**

When working with git, it can be handy to have multiple remote repositories. To specify a remote repository to push to, just append its name to the command.

**git push** origin

**Specify Branch**

To push to a specific branch, say feature\_x:

**git push** origin feature\_x

**Set the remote tracking branch**

Unless the branch you are working on originally comes from a remote repository, simply using **git push** won't work

the first time. You must perform the following command to tell git to push the current branch to a specific

remote/branch combination

**git push** --set-upstream origin master

Here, master is the branch name on the remote origin. You can use -u as a shorthand for --set-upstream.

**Section 29.3: Force Pushing**

Sometimes, when you have local changes incompatible with remote changes (ie, when you cannot fast-forward the remote branch, or the remote branch is not a direct ancestor of your local branch), the only way to push your changes is a force push.

**git push** -f

or

**git push** --force

This will **overwrite** any remote changes and your remote will match your local.

Attention: Using this command may cause the remote repository to **lose commits**. Moreover, it is strongly advised against doing a force push if you are sharing this remote repository with others, since their history will retain every overwritten commit, thus rending their work out of sync with the remote repository.

**Section 29.4: Push tags**

**git push** --tags

Pushes all of the **git** tags in the local repository that are not in the remote one

**Chapter 30: Internals**

**Section 30.1: Repo**

A **git** repository is an on-disk data structure which stores metadata for a set of files and directories.

It lives in your project's .git**/** folder. Every time you commit data to git, it gets stored here.

Inversely, .git**/** contains every single commit.

It's basic structure is like this:

.git**/**

objects**/**

refs**/**

**Section 30.2: Objects**

**git** is fundamentally a key-value store. When you add data to **git**, it builds an object and uses the SHA-1 hash of the object's contents as a key.

Therefore, any content in **git** can be looked up by it's hash:

**git cat-file** -p 4bb6f98

There are 4 types of Object:

* Blob
* **Tree**
* Commit
* Tag

HEAD is a special ref. It always points to the current object. You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

This is what's known as a "detached head" - because HEAD is not attached to (pointing at) any ref, but rather points directly to an object.

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a410e...

They are stored in `.git/refs/heads/ in plain text files.

$ **cat** .git**/**refs**/**heads**/**mainline

4bb6f98a223abc9345a0cef9200562333

Now, it's possible to navigate **git** purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask **git** to go to a specific place by name rather than by hash.

A commit is probably the object type most familiar to **git** users, as it's what they are used to creating with the **git commit** commands.

However, the commit does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other objects which contain the actual contents of the commit.

A commit contains a few things:

* hash of a **tree**
* hash of a parent commit
* author name/email, commiter name/email
* commit message

You can see the contents of any commit like this:

$ **git cat-file** commit 5bac93

**Tree**

A very important note is that the **tree** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each commit contains a snapshot of the entire project\*.

\**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a commit should be considered as containing a complete copy of the project*.

**Parent**

The parent line contains a hash of another commit object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders.

A **tree** contains:

* 0 or more blob objects
* 0 or more **tree** objects

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article. But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects

2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects

3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

This results in a new snapshot of your project being added to **git** that is connected to the previous state.

**Section 30.9: Moving HEAD**

When you run **git checkout** on a commit (specified by hash or ref) you're telling **git** to make your working directory look like how it did when the snapshot was taken.

1. Update the files in the working directory to match the **tree** inside the commit

2. Update HEAD to point to the specified hash or ref

**Section 30.10: Moving refs around**

Running **git reset** --hard moves refs to the specified hash/ref.

Moving MyBranch to b8dc53:

$ **git checkout** MyBranch *# moves HEAD to MyBranch*

$ **git reset** --hard b8dc53 *# makes MyBranch point to b8dc53*

**Section 30.11: Creating new Refs**

Running **git checkout** -b **<**refname**>** will create a new ref that points to the current commit.

$ **cat** .git**/head**

1f324a

$ **git checkout** -b TestBranch

$ **cat** .git**/**refs**/**heads**/**TestBranch

1f324a

**Section 33.1: Cloning the SVN repository**

You need to create a new local copy of the repository with the command

**git svn** clone SVN\_REPO\_ROOT\_URL **[**DEST\_FOLDER\_PATH**]** -T TRUNK\_REPO\_PATH -t TAGS\_REPO\_PATH -b

BRANCHES\_REPO\_PATH

If your SVN repository follows the standard layout (trunk, branches, tags folders) you can save some typing:

**git svn** clone -s SVN\_REPO\_ROOT\_URL **[**DEST\_FOLDER\_PATH**]**

**git svn** clone checks out each SVN revision, one by one, and makes a git commit in your local repository in order to recreate the history. If the SVN repository has a lot of commits this will take a while.

When the command is finished you will have a full fledged git repository with a local branch called master that tracks the trunk branch in the SVN repository.

**Section 34.1: Create an archive of git repository**

With **git archive** it is possible to create compressed archives of a repository, for example for distributing releases.

With **git archive** it is possible to create compressed archives of a repository, for example for distributing releases.

Create a tar archive of current HEAD revision:

**git archive** --format **tar** HEAD **| cat >** archive-HEAD.tar

Create a tar archive of current HEAD revision with gzip compression:

**git archive** --format **tar** HEAD **| gzip >** archive-HEAD.tar.gz

This can also be done with (which will use the in-built tar.gz handling):

**git archive** --format tar.gz HEAD **>** archive-HEAD.tar.gz

Create a zip archive of current HEAD revision:

**git archive** --format **zip** HEAD **>** archive-HEAD.zip

Alternatively it is possible to just specify an output file with valid extension and the format and compression type will be inferred from it:

**git archive** --output=archive-HEAD.tar.gz HEAD

**Section 34.2: Create an archive of git repository with**

**directory prefix**

It is considered good practice to use a prefix when creating git archives, so that extraction will place all files inside a directory. To create an archive of HEAD with a directory prefix:

**git archive** --output=archive-HEAD.zip --prefix=src-directory-name HEAD

When extracted all the files will be extracted inside a directory named src-directory-name in the current directory

**Section 34.3: Create archive of git repository based on**

**specific branch, revision, tag or directory**

It is also possible to create archives of other items than HEAD, such as branches, commits, tags, and directories.

To create an archive of a local branch dev:

**git archive** --output=archive-dev.zip --prefix=src-directory-name dev

To create an archive of a remote branch origin**/**dev:

**git archive** --output=archive-dev.zip --prefix=src-directory-name origin**/**dev

To create an archive of a tag v.01:

**git archive** --output=archive-v.01.zip --prefix=src-directory-name v.01

**Section 37.1: Overview**

**git show** shows various Git objects.

**For commits:**

Shows the commit message and a diff of the changes introduced.

**Command Description**

**git show** shows the previous commit

**git show @**~3 shows the 3rd-from-last commit

**For trees and blobs:**

Shows the tree or blob.

**Command Description**

**git show @**~3: shows the project root directory as it was 3 commits ago (a tree)

**git show @**~3:src**/**program.js shows src**/**program.js as it was 3 commits ago (a blob)

**git show @**:a.txt **@**:b.txt shows a.txt concatenated with b.txt from current commit

**For tags:**

Shows the tag message and the referenced object.

**Section 42.1: Only show certain lines**

Output can be restricted by specifying line ranges as

**git blame** -L **<start>**,**<**end**>**

Where **<start>** and **<end>** can be:

line number

**git blame** -L 10,30

/regex/

**git blame** -L **/**void main**/**, **git blame** -L 46,**/**void foo**/**

+offset, -offset (only for **<end>**)

**git blame** -L 108,+30, **git blame** -L 215,-15

**//** Shows the author and commit per line of specified **file**

**git blame** test.c

**//** Shows the author email and commit per line of specified

**git blame** -e test.c **file**

**//** Limits the selection of lines by specified range

**git blame** -L 1,10 test.c

**Chapter 43: Git revisions syntax**

**Section 43.3: The default revision: HEAD**

$ **git show** *# equivalent to 'git show HEAD'*

'HEAD' names the commit on which you based the changes in the working tree, and is usually the symbolic name for the current branch. Many (but not all) commands that take revision parameter defaults to 'HEAD' if it is missing.

**Section 43.4: Reflog references: <refname>@{<n>}**

$ **git show @{**1**}** *# uses reflog for current branch*

$ **git show** master**@{**1**}** *# uses reflog for branch 'master'*

$ **git show** HEAD**@{**1**}** *# uses 'HEAD' reflog*

A ref, usually a branch or HEAD, followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g.**{**1**}**, **{**15**}**) specifies the n-th prior value of that ref *in your* ***local*** *repository*. You can check recent reflog entries with

**git reflog** command, or --walk-reflogs / -g option to **git log**.

**Section 43.5: Reflog references:<refname>@{<date>}**

$ **git show** master**@{**yesterday**}**

$ **git show** HEAD**@{**5 minutes ago**}** *# or HEAD@{5.minutes.ago}*

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. **{**yesterday**}**, **{**1 month 2 weeks

3 days 1 hour 1 second ago**}** or **{**1979-02-26 18:30:00**}**) specifies the value of the ref at a prior point in time (or closest point to it). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *'master'* branch last week.

You can use **git reflog** with a date specifier to look up exact time where you did something to given ref in the local repository.

$ **git reflog** HEAD**@{**now**}**

08bb350 HEAD**@{**Sat Jul 23 19:48:13 2016 +0200**}**: reset: moving to HEAD^

4ebf58d HEAD**@{**Sat Jul 23 19:39:20 2016 +0200**}**: commit: gitweb**(**1**)**: Document query parameters

08bb350 HEAD**@{**Sat Jul 23 19:26:43 2016 +0200**}**: pull: Fast-forward

**Section 43.6: Tracked / upstream branch:**

**<branchname>@{upstream}**

$ **git log @{**upstream**}**.. *# what was done locally and not yet published, current branch*

$ **git show** master**@{**upstream**}** *# show upstream of branch 'master'*

The suffix **@{**upstream**}** appended to a branchname (short form **<branchname>**@{u}) refers to the branch that the branch specified by branchname is set to build on top of (configured with branch.**<**name**>**.remote and branch.**<**name**>**.merge, or with **git branch** --set-upstream-to=**<**branch**>**). A missing branchname defaults to the current one.

Together with syntax for revision ranges it is very useful to see the commits your branch is ahead of upstream (commits in your local repository not yet present upstream), and what commits you are behind (commits in upstream not merged into local branch), or both:

$ **git log** --oneline **@{**u**}**..

$ **git log** --oneline ..**@{**u**}**

$ **git log** --oneline --left-right **@{**u**}**... *# same as* [*...@{u}*](mailto:...@%7bu%7d)

**Section 43.7: Commit ancestry chain: <rev>^, <rev>~<n>, etc**

$ **git reset** --hard HEAD^ *# discard last commit*

$ **git rebase** --interactive HEAD~5 *# rebase last 4 commits*

A suffix ^ to a revision parameter means the first parent of that commit object. ^**<**n**>** means the <n>-th parent (i.e. **<rev>**^ is equivalent to **<rev>**^1).

A suffix ~**<**n**>** to a revision parameter means the commit object that is the <n>-th generation ancestor of the named commit object, following only the first parents. This means that for example **<rev>**~3 is equivalent to **<rev>**^^^. As a shortcut, **<rev>**~ means **<rev>**~1, and is equivalent to **<rev>**^1, or **<rev>**^ in short.

To find such symbolic names you can use the **git name-rev** command:

$ **git name-rev** 33db5f4d9027a10e477ccf054b2c1ab94f74c85a

33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags**/**v0.99~940

Note that --pretty=oneline and not --oneline must be used in the following example

$ **git log** --pretty=oneline **| git name-rev** --stdin --name-only

master Sixth batch of topics **for** 2.10

master~1 Merge branch 'ls/p4-tmp-refs'

master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'

**Section 43.8: Dereferencing branches and tags: <rev>^0,<rev>^{<type>}**

In some cases the behavior of a command depends on whether it is given branch name, tag name, or an arbitrary revision. You can use "de-referencing" syntax if you need the latter.

A suffix ^ followed by an object type name (tag, commit, **tree**, blob) enclosed in brace pair (for example

v0.99.8^**{**commit**}**) means dereference the object at **<rev>** recursively until an object of type **<type>** is found or the object cannot be dereferenced anymore. **<rev>**^0 is a short-hand for **<rev>**^{commit}.

$ **git checkout** HEAD^0 *# equivalent to 'git checkout --detach' in modern Git*

A suffix ^ followed by an empty brace pair (for example v0.99.8^**{}**) means to dereference the tag recursively until a non-tag object is found.

Compare

$ **git show** v1.0

$ **git cat-file** -p v1.0

$ **git replace** --edit v1.0

with

$ **git show** v1.0^**{}**

$ **git cat-file** -p v1.0^**{}**

$ **git replace** --edit v1.0^**{}**

**Section 43.9: Youngest matching commit: <rev>^{/<text>},:/<text>**

$ **git show** HEAD^**{/**fix nasty bug**}** *# find starting from HEAD*

$ **git show** ':/fix nasty bug' *# find starting from any branch*

A colon (':'), followed by a slash ('/'), followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from *any* ref.

The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. :**/**^foo. The special sequence :**/!** is reserved for modifiers to what is matched. :**/!**-foo performs a negative match, while :**/!!**foo matches a literal ! character, followed by foo.

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the :**/<**text**>** syntax below that it returns the youngest matching commit which is reachable from the **<rev>** before ^.

Section45.1

$ git remote list all configured remote repositories. It shows the short names

To show more detailed information

$ git remote –v

Section 45.2 Change remote url

It takes 2 arguments: an existing remote name (origin, upstream) and the url.

$ git remote set-url name url

**$ git remote** set-url origin https:**//**localserver**/**develop**/**myrepo.git

$ git remote rm dev remote the remote depository dev

Remove the remote named **<name>**. All remote-tracking branches and configuration settings for the remote are removed.

To add a remote, use **git remote** add in the root of your local repository.

$ git remote add name url

You can view more information about a remote repository by **git remote** show **<**remote repository **alias>**

$ git remote show name

Rename the remote named **<old>** to **<new>**. All remote-tracking branches and configuration settings for the remote are updated.

To rename a remote branch name dev to dev1 :

**git remote** rename dev dev1

**Chapter 48: Git statistics**

Section 48.3 Git statistics

$ git shortlog –sn

$ git shortlog -sne

Git shortlog is used to summarize the git log ouput and group the commits by author

-s skips the summary

-n changes the ordering from alphabetical to number of commits descending

-e shows the email

**git shortlog** -sne *#Names along with their email ids and the Number of commits*

**Section 48.8: Show the total number of commits per author**

In order to get the total number of commits that each developer or contributor has made on a repository, you can simply use the **git shortlog**:

**git shortlog** -s

which provides the author names and number of commits by each one.

Additionally, if you want to have the results calculated on all branches, add --all flag to the command:

**git shortlog** -s --all

**Section 48.4: Commits per date**

**git log** --pretty=format:"%ai" **| awk** '{print " : "$1}' **| sort** -r **| uniq** -c

**Section 48.4: Commits per date**

**git log** --pretty=format:"%ai" **| awk** '{print " : "$1}' **| sort** -r **| uniq** -c

**Section 48.5: Total number of commits in a branch**

**git log** --pretty=oneline **|wc** -l

**Section 48.6: List all commits in pretty format**

**git log** --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"

This will give a nice overview of all commits (1 per line) with date, user and commit message.

Section50.1 gitk git gui

Gitk is a graphical history viewer

$ gitk [git log options]

$gitk –all

$ git gui is a primarly a tool for crafting commits

Suppose that you had started an interactive rebase:

**git rebase** --interactive HEAD~20

and by mistake, you squashed or dropped some commits that you didn't want to lose, but then completed the rebase. To recover, do **git reflog**

**Section 53.5: Setting up Beyond Compare**

You can set the path to bcomp.exe

**git config** --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'

and configure bc3 as default

**git config** --global diff.tool bc3

**Chapter 58: Change git repository name**

If you change repository name on the remote side, such as your github or bitbucket, when you push your exisiting code, you will see error: Fatal error, repository not found\*\*.

**Section 58.1: Change local setting**

Go to terminal,

**cd** projectFolder

**git remote** -v **(**it will show previous **git** url**)**

**git remote** set-url origin https:**//**username**@**bitbucket.org**/**username**/**newName.git

**git remote** -v **(**double check, it will show new **git** url**)**

**git push (do** whatever you want.**)**

**Chapter 59: Git Tagging**

Like most Version Control Systems (VCSs), Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

**Section 59.1: Listing all available tags**

Using the command **git tag** lists out all available tags:

$ **git tag**

**<**output follows**>**

v0.1

v1.3

**Note**: the tags are output in an **alphabetical** order.

One may also search for available tags:

$ **git tag** -l "v1.8.5\*"

**Section 59.2: Create and push tag(s) in GIT**

**Create a tag:**

To create a tag on your current branch:

**git tag <** tagname **>**

This will create a local tag with the current state of the branch you are on.

To create a tag with some commit:

**git tag** tag-name commit-identifier

This will create a local tag with the commit-identifier of the branch you are on.

**Push a commit in GIT:**

Push an individual tag:

**git push** origin tag-name

Push all the tags at once

**git push** origin –tags

**Section 60.1: Delete local branches that have been deleted on**

**the remote**

To remote tracking between local and deleted remote branches use

**git fetch** -p

you can then use

**git branch** -vv

to see which branches are no longer being tracked.

Branches that are no longer being tracked will be in the form below, containing 'gone'

branch 12345e6 **[**origin**/**branch: gone**]** Fixed bug

you can then use a combination of the above commands, looking for where 'git branch -vv' returns 'gone' then using '-d' to delete the branches

**git fetch** -p **&& git branch** -vv **| awk** '/: gone]/{print $1}' **| xargs git branch** –d

**Chapter 61: diff-tree**

Compares the content and mode of blobs found via two tree objects.

**Section 61.1: See the files changed in a specific commit**

git diff-tree --no-commit-id --name-only -r COMMIT\_ID