Git Notes for professionals:

**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 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

**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 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

**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 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

**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