Git : Detailed Lab Guide

Part 1: Getting started with Git

Section 1.1: Create your first repository, then add and commit files

At the command line, first verify that you have Git installed:

On all operating systems:

```
git --version
```

On UNIX-like operating systems:

```
which git
```

If nothing is returned, or the command is not recognized, you may have to install Git on your system by downloading and running the installer. See the Git homepage for exceptionally clear and easy installation instructions.

After installing Git, configure your username and email address. Do this *before* making a commit.

Once Git is installed, navigate to the directory you want to place under version control and create an empty Git repository:

```
git init
```

This creates a hidden folder, .git, which contains the plumbing needed for Git to work.

Next, check what files Git will add to your new repository; this step is worth special care:

```
git status
```

Review the resulting list of files; you can tell Git which of the files to place into version control (avoid adding files with confidential information such as passwords, or files that just clutter the repo):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

If all files in the list should be shared with everyone who has access to the repository, a single command will add everything in your current directory and its subdirectories:

```
git add .
```

This will "stage" all files to be added to version control, preparing them to be committed in your first commit.

For files that you want never under version control, create and populate a file named .gitignore before running the add command.

Commit all the files that have been added, along with a commit message:

```
git commit -m "Initial commit"
```

This creates a new commit with the given message. A commit is like a save or snapshot of your entire project. You can now push, or upload, it to a remote repository, and later you can jump back to it if necessary.

If you omit the -m parameter, your default editor will open and you can edit and save the commit message there.

Adding a remote

To add a new remote, use the **git remote** add command on the terminal, in the directory your repository is stored at.

The **git remote** add command takes two arguments:

1. A remote name, for example, origin
2. A remote URL, for example, https://<your-git-service-address>/user/repo.git

```
git remote add origin https://<your-git-service-address>/owner/repository.git
```

NOTE: Before adding the remote you have to create the required repository in your git service, You'll be able to push/pull commits after adding your remote.

Section 1.2: Clone a repository

The **git clone** command is used to copy an existing Git 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: one which has only the .git

objects and doesn't create a working copy in the filesystem. As a bonus you set this remote as an upstream server to easily share updates with other programmers.
On the remote server:

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

On the local machine:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Note that ssh: is just one possible way of accessing the remote repository.)
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 Git commands, e.g. git pull.

Section 1.4: Setting your user name and email

You need to **set** who you are *before* creating any commit. That will allow commits to have the right author name and email associated to them.

It has nothing to do with authentication when pushing to a remote repository (e.g. when pushing to a remote repository using your GitHub, BitBucket, or GitLab account)

To declare that identity for *all* repositories, use git config --global
This will store the setting in your user's .gitconfig file: e.g. $HOME/.gitconfig or for Windows, %USERPROFILE%\.gitconfig.

```
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 $GIT_DIR/config. e.g. /path/to/your/repo/.git/config.

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email 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.

Tips: if you have different identities (one for open-source project, one at work, one for private repos, ...), and you don't want to forget to set the right one for each different repos you are working on:

Remove a global identity

```
git config --global --remove-section user.name
git config --global --remove-section user.email
```

Version ≥ 2.8

To force git to look for your identity only within a repository's settings, not in the global config:

```
git config --global user.useConfigOnly true
```

That way, if you forget to set your user.name and user.email for a given repository and try to make a commit, you will see:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Section 1.5: Setting up the upstream remote

If you have cloned a fork (e.g. an open source project on Github) you may not have push access to the upstream repository, so you need both your fork but be able to fetch the upstream repository.

First check the remote names:

```
$ git remote -v
origin https://github.com/myusername/repo.git (fetch)
origin https://github.com/myusername/repo.git (push)
upstream # this line may or may not be here
```

If upstream is there already (it is on *some* Git versions) you need to set the URL (currently it's empty):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

If the upstream is **not** there, or if you also want to add a friend/colleague's fork (currently they do not exist):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Section 1.6: Learning about a command

To get more information about any git command – i.e. details about what the command does, available options and
other documentation – use the --help option or the **help** command.
For example, to get all available information about the **git diff** command, use:

```
git diff --help
git help diff
```

Similarly, to get all available information about the status command, use:

```
git status --help
git help status
```

If you only want a quick help showing you the meaning of the most used command line flags, use -h:

```
git checkout -h
```

Section 1.7: Set up SSH for Git

If you are using **Windows** open Git Bash. If you are using **Mac** or **Linux** open your Terminal.
Before you generate an SSH key, you can check to see if you have any existing SSH keys.
List the contents of your ~/.ssh directory:

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

7

Check the directory listing to see if you already have a public SSH key. By default the filenames of the public keys are one of the following:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

If you see an existing public and private key pair listed that you would like to use on your Bitbucket, GitHub (or similar) account you can copy the contents of the id_*.pub file.
If not, you can create a new public and private key pair with the following command:

```
$ ssh-keygen
```

Press the Enter or Return key to accept the default location. Enter and re-enter a passphrase when prompted, or leave it empty.
Ensure your SSH key is added to the ssh-agent. Start the ssh-agent in the background if it's not already running:

```
$ eval "$(ssh-agent -s)"
```

Add you SSH key to the ssh-agent. Notice that you'll need te replace id_rsa in the command with the name of your **private key file**:

```
$ ssh-add ~/.ssh/id_rsa
```

If you want to change the upstream of an existing repository over SSH you can run the following command:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Section 1.8: Git Installation

Let's get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major
ones are to install it in reverse chronological
order – that is, the most recent commits show up first. As you can see, this command lists each commit
with its SHA-1 checksum, the author's name and email, the date written, and the commit message. -
source

Example

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian
Date: Thu Mar 24 15:52:07 2016 -0700
Merge pull request #7724 from BKinahan/fix/where-art-thou
Fix 'its' typo in Where Art Thou description
commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
```

```
Author: BKinahan
Date: Thu Mar 24 21:11:36 2016 +0000
Fix 'its' typo in Where Art Thou description
commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra
Date: Thu Mar 24 14:26:04 2016 +0530
Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
Remove unnecessary comma from CONTRIBUTING.md
```

If you wish to limit your command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

```
git log -2
```

Section 2.2: Prettier log

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

```
git log --decorate --oneline --graph
```

sample output :

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
* 9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* | f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

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

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

To use the alias version:

```
# history of current branch :
git lol
# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master
# combined history of everything in your repo :
git lol --all
```

Section 2.3: Colorize Logs

```
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

Section 2.4: Oneline log

```
git log --oneline
```

will show all of your commits with only the first part of the hash and the commit message. Each commit will be in a single line, as the oneline flag suggests.

The oneline option prints each commit on a single line, which is useful if you’ re looking at a lot of commits. - **source**

Example(repository, with the same section of code from the other example):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

If you wish to limit you command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

```
git log -2 --oneline
```

Section 2.5: Log search

```
git log -S"#define SAMPLES"
```

Searches for **addition** or **removal** of specific string or the string **matching** provided REGEXP. In this case we're looking for addition/removal of the string #define SAMPLES. For example:

```
+#define SAMPLES 100000
```

or

```
-#define SAMPLES 100000
```

```
git log -G"#define SAMPLES"
```

Searches for **changes** in **lines containing** specific string or the string **matching** provided REGEXP. For example:

```
-#define SAMPLES 100000
+#define SAMPLES 100000000
```

Section 2.6: List all contributions grouped by author name

git shortlog summarizes git log and groups by author

If no parameters are given, a list of all commits made per committer will be shown in chronological order.

```
$ git shortlog
Committer 1 (<number_of_commits>):
Commit Message 1
Commit Message 2
```

```
...
Committer 2 (<number_of_commits>):
Commit Message 1
Commit Message 2
...
```

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

```
-s
--summary
```

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

To sort the output by number of commits instead of alphabetically by committer name, pass in the numbered option:

```
-n
--numbered
```

To add the email of a committer, add the email option:

```
-e
--email
```

A custom format option can also be provided if you want to display information other than the commit subject:

```
--format
```

This can be any string accepted by the --format option of git log.

See **Colorizing Logs** above for more information on this.

Section 2.7: Searching commit string in git log

Searching git log using some string in log:

```
git log [options] --grep "search_string"
```

Example:

```
git log --all --grep "removed file"
```

Will search for removed **file** string in **all logs** in **all branches**.

Starting from git 2.4+, the search can be inverted using the --invert-grep option.

Example:

```
git log --grep="add file" --invert-grep
```

Will show all commits that do not contain add **file**.

Section 2.8: Log for a range of lines within a file

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date: Tue Mar 22 16:33:42 2016 -0500
commit message
diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
<!DOCTYPE HTML>
<html>
- <head>
- <meta charset="utf-8">
+
+<head>
+ <meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Section 2.9: Filter logs

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

Specific dates work too:

```
git log --after 2016-05-01
```

As with other commands and flags that accept a date parameter, the allowed date format is as supported by GNU date (highly flexible).

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
```

Section 2.10: Log with changes inline

To see the log with changes inline, use the -p or --patch options.

```
git log --patch
```

Example ( )

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date: Wed Mar 2 10:35:25 2016 -0800
fix readme error link
```

```
diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
the logging. The criteria for matching errors is based on the constructor and
message.
-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).
```

```
## Asynchronous behaviors
commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Section 2.11: Log showing commited files

```
git log --stat
```

Example:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date: Mon Jun 6 21:32:30 2016 -0300
MercadoLibre java-sdk dependency
mltracking-poc/.gitignore | 1 +
mltracking-poc/pom.xml | 14 ++++++++++++--
2 files changed, 13 insertions(+), 2 deletions(-)
commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date: Sat Jun 4 12:35:16 2016 -0300
[manasses] generated by SpringBoot initializr
.gitignore | 42
++++++++++++
mltracking-poc/mvnw | 233
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
mltracking-poc/mvnw.cmd | 145
++++++++++++++++++++++++++++++++++++++++
mltracking-poc/pom.xml | 74
++++++++++++++++++++
mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java | 12 ++++
mltracking-poc/src/main/resources/application.properties | 0
mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java | 18 +++++
7 files changed, 524 insertions(+)
```

Section 2.12: Show the contents of a single commit

Using git show we can view a single commit

```
git show 48c83b3
```

```
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Example

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date: Wed May 4 18:26:40 2016 -0400
The commit message will be shown here.
diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {
colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
- colorMap.put(BuildStatus.SUCCESS, Color.decode( "#55FF55" ));
+ colorMap.put(BuildStatus.SUCCESS, Color.decode( "#33CC33" ));
colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Section 2.13: Git Log Between Two Branches

git log master..foo will show the commits that are on foo and not on master. Helpful for seeing what commits you've added since branching!

Section 2.14: One line showing commiter name and time since commit

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
```

```
%C(yellow) %d %Creset %s"' --all --graph
```
example
```
* 40554ac 3 months ago Alexander Zolotov Merge pull request #95 from
gmandnepr/external_plugins
|\
| * e509f61 3 months ago Ievgen Degtiarenko Documenting new property
| * 46d4cb6 3 months ago Ievgen Degtiarenko Running idea with external plugins
| * 6253da4 3 months ago Ievgen Degtiarenko Resolve external plugin classes
| * 9fdb4e7 3 months ago Ievgen Degtiarenko Keep original artifact name as this may be
important for intellij
| * 22e82e4 3 months ago Ievgen Degtiarenko Declaring external plugin in intellij section
|/
* bc3d2cb 3 months ago Alexander Zolotov Ignore DTD in plugin.xml
```

Part 3: Working with Remotes

Section 3.1: 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]
```

Section 3.2: Changing Git Remote URL

Check existing remote
```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```
Changing repository URL
```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```
Verify new remote URL
```
git remote -v
# origin https://github.com/username/repo2.git (fetch)
# origin https://github.com/username/repo2.git (push)
```

Section 3.3: List Existing Remotes

List all the existing remotes associated with this repository:
```
git remote
```
List all the existing remotes associated with this repository in detail including the fetch and push URLs:
```
git remote --verbose
```
or simply
```
git remote -v
```

Section 3.4: Removing Local Copies of Deleted Remote Branches

If a remote branch has been deleted, your local repository has to be told to prune the reference to it.
To prune deleted branches ")
```
git fetch remote-name
git merge remote-name/branch-name
```
The pull command combines a fetch and a merge.
```
git pull
```
The pull with --rebase flag command combines a fetch and a rebase instead of merge.
```
git pull --rebase remote-name branch-name
```

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.
You will see sometimes refs/tags/v0.1.6 *and* refs/tags/v0.1.6^{}: the ^{} to list the dereferenced annotated tag (ie the commit that tag is pointing to)
Since git 2.8 (March 2016), you can avoid that double entry for a tag, and list directly those dereferenced tags with:

```
git ls-remote --ref
```

It can also help resolve the actual url used by a remote repo when you have "url.<base>.insteadOf" config setting.
If `git remote --get-url <aremotename>` returns https://server.com/user/repo, and you have set `git config url.ssh://git@server.com:.insteadOf https://server.com/:`

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Section 3.7: Adding a New Remote Repository

```
git remote add upstream git-repository-url
```

Adds remote git repository represented by git-repository-url as new remote named upstream to the git repository

Section 3.8: Set Upstream on a New Branch

You can create a new branch and switch to it using

```
git checkout -b AP-57
```

19

After you use git checkout to create a new branch, you will need to set that upstream origin to push to using

```
git push --set-upstream origin AP-57
```

After that, you can use git push while you are on that branch.

Section 3.9: Getting Started

Syntax for pushing to a remote branch

```
git push <remote_name> <branch_name>
```

Example

```
git push origin master
```

Section 3.10: Renaming a Remote

To rename remote, use command `git remote rename`
The `git remote rename` command takes two arguments:
An existing remote name, for example : **origin**
A new name for the remote, for example : **destination**
Get existing remote name

```
git remote
# origin
```

Check existing remote with URL

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Rename remote

```
git remote rename origin destination
# Change remote name
```

you want your remote to point to, you can use the set-url option, like so:

```
git remote set-url <remote_name> <remote_repository_url>
```

Example:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

21

Part 4: Staging

Section 4.1: Staging All Changes to Files

```
git add -A
```

Version ≥ 2.0

```
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.
Use git add -A, or its equivalent command git add --all, to stage all changes to files in any version of git.

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
or
git add --patch
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.
Stage this hunk [y,n,q,a,d,/,s,e,?]?
y stage this hunk for the next commit
n do not stage this hunk for the next commit
q quit; do not stage this hunk or any of the remaining hunks
a stage this hunk and all later hunks in the file
d do not stage this hunk or any of the later hunks in the file
g select a hunk to go to
/ search for a hunk matching the given regex
j leave this hunk undecided, see next undecided hunk
J leave this hunk undecided, see next hunk
k leave this hunk undecided, see previous undecided hunk
K leave this hunk undecided, see previous hunk
s split the current hunk into smaller hunks
e manually edit the current hunk
? print hunk help
This makes it easy to catch changes which you do not want to commit.
You can also open this via git add --interactive and selecting p.

Section 4.4: Interactive add

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 .
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
In the unusual case that you need to ignore tracked files, special care should be taken. See: Ignore files that have already been committed to a Git repository.
Examples
Here are some generic examples of rules in a .gitignore file, based on glob file patterns:

```
# Lines starting with `#` are comments.
# Ignore files called 'file.ext'
file.ext
# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment
# 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/
   25
# 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
```

. If you want to ignore certain files without committing the ignore rules, here are some options:

Edit the .git/info/exclude file (using the same syntax as .gitignore). The rules will be global in the scope of the repository;
Set up a global gitignore file that will apply ignore rules to all your local repositories:
Furthermore, you can ignore local changes to tracked files without changing the global git configuration with:
`git update-index --skip-worktree [<file>...]`: for minor local modifications
`git update-index --assume-unchanged [<file>...]`: for production ready, non-changing files upstream
See more details on differences between the latter flags and the **git update-index** documentation for further options.

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.
See the **git clean** documentation for more details.
See the Git manual for more details.

Section 5.2: Checking if a file is ignored

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
```

Here, only *.o files are defined in .gitignore, so Readme.md is not listed in the output of **git** check-ignore.
If you want to see line of which .gitignore is responsible for ignoring a file, add -v to the git check-ignore command:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o example.o
```

From Git 1.7.6 onwards you can also use **git status** --ignored in order to see ignored files. You can find more info on this in the official documentation or in Finding files ignored by .gitignore.

Section 5.3: Exceptions in a .gitignore file

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.
If the file is in an ignored folder, you can **NOT** re-include it so easily:

```
folder/
!folder/*.txt
```

In this example all .txt files in the folder would remain ignored.
The right way is re-include the folder itself on a separate line, then ignore all files in folder by *, finally re-include the *.txt in folder, as the following:

```
!folder/
folder/*
!folder/*.txt
```

Note: For file names beginning with an exclamation mark, add two exclamation marks or escape with the \ character:

```
!!includethis
\!excludethis
```

Section 5.4: A global .gitignore file

To have Git ignore certain files across all repositories you can create a global .gitignore with the following command
in your terminal or command prompt:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git will now use this in addition to each repository's own .gitignore file. Rules for this are:
If the local .gitignore file explicitly includes a file while the global .gitignore ignores it, the local .gitignore takes priority (the file will be included)
If the repository is cloned on multiple machines, then the global .gigignore must be loaded on all machines or at least include it, as the ignored files will be pushed up to the repo while the PC with the global .gitignore wouldn't update it. This is why a repo specific .gitignore is a better idea than a global one if the project is worked on by a team
This file is a good place to keep platform, machine or user specific ignores, e.g. OSX .DS_Store, Windows Thumbs.db
or Vim *.ext~ and *.ext.swp ignores if you don't want to keep those in the repository. So one team member working on OS X can add all .DS_STORE and _MACOSX (which is actually useless), while another team member on
Windows can ignore all thumbs.bd

Section 5.5: Ignore files that have already been committed to

a Git repository

If you have already added a file to your Git repository and now want to **stop tracking it** (so that it won't be present in future commits), you can remove it   and prevent further changes   after you removed the file   and shared with other
contributors and users. You can set a global .gitignore, but then all your repositories would share those settings.
If you want to ignore certain files in a repository locally and not make the file part of any repository, edit .git/info/exclude inside your repository.
For example:

```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
```

```
pushReports.py
server/
```

Section 5.7: Ignoring subsequent changes to a file (without removing it)

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, e.g.:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env
# commit to Git
git add .env
git commit -m "Adding .env template"
# ignore future changes to .env
git update-index --assume-unchanged .env
# update your password
vi .env
```

```
# no changes!
git status
```

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
```

Or you can create a .gitignore file in the bar/ directory. Equivalent to the previous example would be creating file bar/.gitignore with these contents:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

Section 5.9: Prefilled .gitignore Templates

If you are unsure which rules to list in your .gitignore file, or you just want to add generally accepted exceptions
0
to your project, you can choose or generate a .gitignore file:

https://www.gitignore.io/

https://github.com/github/gitignore

Many hosting services such as GitHub and BitBucket offer the ability to generate .gitignore files based upon the programming languages and IDEs you may be using:

Section 5.10: Ignoring files in subfolders (Multiple gitignore files)

Suppose you have a repository structure like this:

```
examples/
output.log
src/
<files not shown>
output.log
README.md
```

output.log in the examples directory is valid and required for the project to gather an understanding while the one beneath src/ is created while debugging and should not be in the history or part of the repository.

There are two ways to ignore this file. You can place an absolute path into the .gitignore file at the root of the working directory:
1

```
# /.gitignore
src/output.log
```

Alternatively, you can create a .gitignore file in the src/ directory and ignore the file that is relative to this .gitignore:

```
# /src/.gitignore
output.log
```

Section 5.11: Create an Empty Folder

It 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:

Method one: .gitkeep

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

Method two: dummy.txt

Another hack for this is very similar to the above and the same steps can be followed, but instead of a .gitkeep, just use a dummy.txt instead. This has the added bonus of being able to easily create it in Windows using the context menu. And you get to leave funny messages in them too.You can also use .gitkeep file to track the empty directory. .gitkeep normally is an empty file that is added to track the empty 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
```

So if we have repository structure like this:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

...and .gitignore file containing:

```
example_2
```

...than result of the command will be:

```
$ git status --ignored
 2
On branch master
Initial commit
Untracked files:
(use "git add <file>..." to include in what will be committed)
.gitignore
.example_1
Ignored files:
(use "git add -f <file>..." to include in what will be committed)
dir/
example_2
```

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

Result will look like this:

```
$ git status --ignored --untracked-files=all
On branch master
Initial commit
Untracked files:
(use "git add <file>..." to include in what will be committed)
.gitignore
example_1
Ignored files:
(use "git add -f <file>..." to include in what will be committed)
dir/example_2
example_2
```

Section 5.13: Ignoring only part of a file [stub]

Sometimes you may want to have local changes in a file you don't want to commit or publish. Ideally local settings should be concentrated in a separate file that can be placed into .gitignore, but sometimes as a short-term solution it can be helpful to have something local in a checked-in file.

You can make Git "unsee" those lines using clean filter. They won't even show up in diffs.

Suppose here is snippet .

To fix this problem, one could perform a "dry-run" removal of everything in the repository, followed by re-adding all

the files back. As long as you don't have pending changes and the --cached parameter is passed, this command is fairly safe to run:

# Remove everything   where you want to use it. Or just edit
~/.gitattributes.

```
*.strings diff=utf16
```

This will convert all files ending in .strings before git diffs.

You can do similar things for other files, that can be converted to text.

For binary plist files you edit .gitconfig

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

and .gitattributes

```
*.plist diff=plist
```

Part 7: Undoing

Section 7.1: Return to a previous commit

To jump back to a previous commit, first find the commit's hash using git log.
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 to a previous commit while keeping the changes:

```
git reset --soft 789abcd
```

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

```
git reset --soft HEAD~
```

To permanently discard any changes made after a specific commit, use:

```
git reset --hard 789abcd
```

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

```
git reset --hard HEAD~
```

Beware: While you can recover the discarded commits using reflog and reset, uncommitted changes cannot be recovered. Use git stash; git reset instead of git reset --hard to be safe.

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 then you can follow the same procedure as in undo
the commit although there are some subtle differences.
A reset is the simplest option as it will undo both the merge commit and any commits added .
It records some new commits to reverse the effect of some earlier commits, which you can push safely without rewriting history.
Don't use git push --force unless you wish to bring down the opprobrium of all other users of that repository.
Never rewrite public history.
If, for example, you've just pushed up a commit that contains a bug and you need to back it out, do the following:

```
git revert HEAD~1
git push
```

Now you are free to revert the revert commit locally, fix your code, and push the good code:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

If the commit you want to revert is already further back in the history, you can simply pass the commit hash. Git will
create a counter-commit undoing your original commit, which you can push to your remote safely.

```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Section 7.6: Undo / Redo a series of commits

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

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

-i puts rebase in "interactive mode". It starts off like the rebase discussed above, but before replaying any commits,
it pauses and allows you to gently modify each commit as it's replayed.*rebase -i* will open in your default text editor, with a list of commits being applied, like this:

To drop a commit, just delete that line in your editor. If you no longer want the bad commits in your project, you can delete lines 1 and 3-4 above.If you want to combine two commits together, you can use the squash or fixup commands

Part 8: Merging

Parameter Details

-m	Message to be included in the merge commit
-v	Show verbose output
--abort	Attempt to revert all files back to their state
--ff-only	Aborts instantly when a merge-commit would be required
--no-ff	Forces creation of a merge-commit, even if it wasn't mandatory
--no-commit	Pretends the merge failed to allow inspection and tweaking of the result
--stat	Show a diffstat after merge completion
-n/--no-stat	Don't show the diffstat
--squash	Allows for a single commit on the current branch with the merged changes

Section 8.1: Automatic Merging

When the commits on two branches don't conflict, Git can automatically merge them:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
file_a | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Section 8.2: Finding all branches with no merged changes

Sometimes you might have branches lying around that have already had their changes merged into master. This finds all branches that are not master that have no unique commits as compared to master. This is very useful for finding branches that were not deleted after the PR was merged into master.

```
for branch in $(git branch -r) ; do
[ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] && echo -
e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
done | sort -r
```

Section 8.3: Aborting a merge

After starting a merge, you might want to stop the merge and return everything to its pre-merge state. Use --abort:

```
git merge --abort
```

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   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.

Section 9.2: Updating a Submodule

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
```

Sometimes instead of using the state that is referenced you want to update to your local checkout to the latest state of that submodule on a remote. To check out all submodules to the latest state on the remote with a single command, you can use

```
git submodule foreach git pull <remote> <branch>
```

or use the default git pull arguments

```
git submodule foreach git pull
```

Note that this will just update your local working copy. Running git status will list the submodule directory as dirty if it changed because of this command. To update your repository to reference the new state instead, you have to commit the changes:

```
git add <submodule_directory>
git commit
```

There might be some changes you have that can have merge conflict if you use git pull so you can use git pull --rebase to rewind your changes to top, most of the time it decreases the chances of conflict. Also it pulls all the branches to local.

```
git submodule foreach git pull --rebase
```

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
```

48

You should add and commit the new .gitmodules file; this tells Git what submodules should be cloned when git submodule update is run.

Section 9.4: Setting a submodule to follow a branch

A submodule is always checked out at a specific commit SHA1 (the "gitlink", special entry in the index of the parent repo)
But one can request to update that submodule to the latest commit of a branch of the submodule remote repo.
Rather than going in each submodule, doing a git checkout abranch --track origin/abranch, git pull, you can simply do (. 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 .

Section 10.2: Good commit messages

It is important for someone traversing through the git log to easily understand what each commit was all about.
Good commit messages usually include a number of a task or an issue in a tracker and a concise description of what has been done and why, and sometimes also how it has been done.
Better messages may look like:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Whereas the following messages would not be quite as useful:

```
fix // What has been fixed?
```

52

```
just a bit of a change // What has changed?
TASK-371 // No description at all, reader will need to look at the tracker
themselves for an explanation
```

Implemented IFoo in IBar // Why it was needed?
A way to test if a commit message is written in the correct mood is to replace the blank with the message and see if
it makes sense:
If I add this commit, I will ___ to my repository.
The seven rules of a great git commit message
1. Separate the subject line) then you can amend your
commit.

```
git commit --amend
```

This will put the currently staged changes onto the previous commit.
Note: This can also be used to edit an incorrect commit message. It will bring up the default editor (usually vi / vim / emacs) and allow you to change the prior message.
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>"
```

Note: Be aware that amending the most recent commit replaces it entirely and the previous commit is removed
 you work on (on Linux, this file is
located at ~/.gitconfig
--local Edits the respository-specific configuration file, which is located at .git/config in your repository; this is the default setting

Section 13.1: Setting which editor to use

There are several ways to set which editor to use for committing, rebasing, etc.
Change the core.editor configuration setting.

```
$ git config --global core.editor nano
```

Set the GIT_EDITOR environment variable.
For one command:

```
$ GIT_EDITOR=nano git commit
```

Or for all commands run in a terminal. **Note:** This only applies until you close the terminal.

```
$ export GIT_EDITOR=nano
```

To change the editor for *all* terminal programs, not just Git, set the VISUAL or EDITOR environment variable.
(See VISUAL vs EDITOR.)

```
$ export EDITOR=nano
```

Note: As above, this only applies to the current terminal; your shell will usually have a configuration file to allow you to set it permanently. (On bash, for example, add the above line to your ~/.bashrc or ~/.bash_profile.)
Some text editors (mostly GUI ones) will only run one instance at a time, and generally quit if you already have an instance of them open. If this is the case for your text editor, Git will print the message Aborting commit due to empty commit message. without allowing you to edit the commit message first. If this happens to you, consult your text editor's documentation to see if it has a --wait flag (or similar) that will make it pause until the document is closed.

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.

However, bigger mistakes will be considered as missing commands, so typing something like git testingit would result in testingit is not a git command.

Section 13.3: List and edit the current configuration

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
```

git config is the command to get or set options
--global means that the configuration file specific to your user account will be edited
user.name and user.email are the keys for the configuration variables; user is the section of the configuration file. name and email are the names of the variables.
"Mr. Bean" and mrbean@example.com are the values that you're storing in the two variables. Note the quotes around "Mr. Bean", which are required because the value you are storing contains a space.

Section 13.5: Multiple usernames and email address

Since Git 2.13, multiple usernames and email addresses could be configured by using a folder filter.

Example for Windows:

.gitconfig

Edit: git config --global -e

Add:

```
[includeIf "gitdir:D:/work"]
path = .gitconfig-work.config
[includeIf "gitdir:D:/opensource/"]
path = .gitconfig-opensource.config
```

Notes
The order is depended, the last one who matches "wins".
the / at the end is needed - e.g. "gitdir:D:/work" won't work.
the gitdir: prefix is required.

.gitconfig-work.config

File in the same directory as *.gitconfig*

```
[user]
name = Money
email = work@somewhere.com
```

.gitconfig-opensource.config

File in the same directory as *.gitconfig*

```
[user]
```

name = Nice
email = cool@opensource.stuff
Example for Linux
[includeIf "gitdir:~/work/"]
path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
path = .gitconfig-opensource
The file content and notes under section Windows.

Section 13.6: Multiple git configurations

You have up to 5 sources for git configuration:
6 files:
%ALLUSERSPROFILE%\Git\Config (Windows only)
(system) **<git>/etc/gitconfig**, with **<git>** being the git installation path.
(on Windows, it is **<git>\mingw64\etc\gitconfig**)
(system) **$XDG_CONFIG_HOME/git/config** (Linux/Mac only)
(global) ~/.gitconfig (Windows: %USERPROFILE%\.gitconfig)
(local) .git/config (within a git repo $GIT_DIR)
a **dedicated file** (with git config -f), used for instance to modify the config of submodules: git config -f .gitmodules ...
the command line with git -c: git -c core.autocrlf=false fetch would override *any* other core.autocrlf to false, *just* for that fetch command.
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.
"resolved" means it lists only the final overridden config value.
72
Since git 2.8, if you want to see which config comes , 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
76
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>
```

Section 14.4: Quick switch to the previous branch

You can quickly switch to the previous branch using
```
git checkout -
```

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 14.6: Delete a branch locally

```
$ 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
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run '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 14.7: Create an orphan branch (i.e. branch with no parent commit)

```
git checkout --orphan new-orphan-branch
```

The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected .

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>
```

Section 14.11: Move current branch HEAD to an arbitrary commit

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.

Part 15: Rev-List

Parameter Details

--oneline Display commits as a single line with their title.

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 , especially when you work in a small team / proprietary project

Part 19: Git Clean

Parameter Details

-d
Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use -f option twice if you really want to remove such a directory.
-f, --force
If the Git configuration variable clean. requireForce is not set to false, git clean will refuse to delete files or directories unless given -f, -n or -i. Git will refuse to delete directories with .git sub directory or file unless a second -f is given.
-i, --interactive Interactively prompts the removal of each file.
-n, --dry-run Only displays a list of files to be removed, without actually removing them.
-q,--quiet Only display errors, not the list of successfully removed files.

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:

```
Would remove the following items:
folder/file1.py
folder/file2.py
*** Commands ***
1: clean 2: filter by pattern 3: select by numbers 4: ask each
5: quit 6: help
What now>
```

Interactive option i can be added along with other options like X, d, etc.

Section 19.2: Forcefully remove untracked files

```
git clean -f
```

Will remove all untracked files.

Section 19.3: Clean Ignored Files

```
git clean -fX
```

Will remove all ignored files 's server, pulling with Git takes the current code on the server and 'pulls' it down 's server to your local machine. This topic explains the process of pulling code using Git as well as the situations one might encounter while pulling different code into the local copy.

Section 23.1: Pulling changes to a local repository

Simple pull

When you are working on a remote repository (say, GitHub) with someone else, you will at some point want to share your changes with them. Once they have pushed their changes to a remote repository, you can retrieve those
changes by *pulling* .

```
git pull
```

Will do it, in the majority of cases.
Pull 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
```

99

Making it the default behavior

To make this the default behavior for newly created branches, type the following command:

```
git config branch.autosetuprebase always
```

To change the behavior of an existing branch, use this:

```
git config branch.BRANCH_NAME.rebase true
```

And

```
git pull --no-rebase
```

To perform a normal merging pull.

Check if fast-forwardable

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.

Section 23.6: Pull, "permission denied"

Some problems can occur if the .git folder has wrong permission. Fixing this problem by setting the owner of the complete .git folder. Sometimes it happen that another user pull and change the rights of the .git folder or files.
To fix the problem:

```
chown -R youruser:yourgroup .git/
```

100

Part 24: Hooks

Section 24.1: Pre-push

Available in Git 1.8.2 and above.

Version ≥ 1.8

Pre-push hooks can be used to prevent a push .

```
$ cat .git/hooks/post-receive
#!/bin/bash
IFS=' '
while read local_ref local_sha remote_ref remote_sha
do
echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
```

102

```
echo Forwarding feature branch to other repository: $ref
git push -q --force other_repos $ref
}
done
```

In this example, the **egrep** regexp looks for a specific branch format (here: JIRA-12345 as used to name Jira issues).
You can leave this part off if you want to forward all branches, of course.

Section 24.4: Commit-msg

This hook is similar to the prepare-commit-msg hook, but it's called after the user enters a commit message rather than before. This is usually used to warn developers if their commit message is in an incorrect format.
The only argument passed to this hook is the name of the file that contains the message. If you don't like the message that the user has entered, you can either alter this file in-place (same as prepare-commit-msg) or you can abort the commit entirely by exiting with a non-zero status.
The following example is used to check if the word ticket followed by a number is present on the commit message

```
word="ticket [0-9]"
```

```
isPresent=$(grep -Eoh "$word" $1)
if [[ -z $isPresent ]]
then echo "Commit message KO, $word is missing"; exit 1;
else echo "Commit message OK"; exit 0;
fi
```

Section 24.5: Local hooks

Local hooks affect only the local repositories in which they reside. Each developer can alter their own local hooks, so they can't be used reliably as a way to enforce a commit policy. They are designed to make it easier for developers to adhere to certain guidelines and avoid potential problems down the road.
There are six types of local hooks: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout, and pre-rebase.
The first four hooks relate to commits and allow you to have some control over each part in a commit's life cycle. The final two let you perform some extra actions or safety checks for the git checkout and git rebase commands.
All of the "pre-" hooks let you alter the action that' s about to take place, while the "post-" hooks are used primarily
for notifications.

Section 24.6: Post-checkout

This hook works similarly to the post-commit hook, but it's called whenever you successfully check out a reference with **git checkout**. This could be a useful tool for clearing out your working directory of auto-generated files that would otherwise cause confusion.
This hook accepts three parameters:
1. the ref of the previous HEAD,
2. the ref of the new HEAD, and
3. a flag indicating if it was a branch checkout or a file checkout (1 or 0, respectively).

Its exit status has no affect on the **git checkout** command.

Section 24.7: Post-commit

This hook is called immediately after the commit-msg hook. It cannot alter the outcome of the **git commit** operation, therefore it's used primarily for notification purposes.
The script takes no parameters, and its exit status does not affect the commit in any way.

Section 24.8: Post-receive

This hook is called after a successful push operation. It is typically used for notification purposes.
The script takes no parameters, but is sent the same information as pre-receive via standard input:
<old-value> <new-value> <ref-name>

Section 24.9: Pre-commit

This hook is executed every time you run **git commit**, to verify what is about to be committed. You can use this hook to inspect the snapshot that is about to be committed.
This type of hook is useful for running automated tests to make sure the incoming commit doesn't break existing functionality of your project. This type of hook may also check for whitespace or EOL errors.
No arguments are passed to the pre-commit script, and exiting with a non-zero status aborts the entire commit.

Section 24.10: Prepare-commit-msg

This hook is called after the pre-commit hook to populate the text editor with a commit message. This is typically used to alter the automatically generated commit messages for squashed or merged commits.
One to three arguments are passed to this hook:
The name of a temporary file that contains the message.
The type of commit, either
message (-m or -F option),
template (-t option),
merge (if it's a merge commit), or

squash (if it's squashing other commits).
The SHA1 hash of the relevant commit. This is only given if -c, -C, or --amend option was given.
Similar to pre-commit, exiting with a non-zero status aborts the commit.

Section 24.11: Pre-rebase

This hook is called before **git rebase** begins to alter code structure. This hook is typically used for making sure a rebase operation is appropriate.
This hook takes 2 parameters:
1. the upstream branch that the series was forked from, and
2. the branch being rebased (empty when rebasing the current branch).

You can abort the rebase operation by exiting with a non-zero status.

Section 24.12: Pre-receive

This hook is executed every time somebody uses **git push** to push commits to the repository. It always resides in the remote repository that is the destination of the push and not in the originating (local) repository.
The hook runs before any references are updated. It is typically used to enforce any kind of development policy.
The script takes no parameters, but each ref that is being pushed is passed to the script on a separate line on standard input in the following format:

```
<old-value> <new-value> <ref-name>
```

Section 24.13: Update

This hook is called after pre-receive, and it works the same way. It's called before anything is actually updated, but is called separately for each ref that was pushed rather than all of the refs at once.
This hook accepts the following 3 arguments:
name of the ref being updated,
old object name stored in the ref, and
new object name stored in the ref.
This is the same information passed to pre-receive, but since update is invoked separately for each ref, you can reject some refs while allowing others.

Part 25: Cloning Repositories

Section 25.1: Shallow Clone

Cloning a huge repository (like a project with multiple years of history) might take a long time, or fail because of the
amount of data to be transferred. In cases where you don't need to have the full history available, you can do a shallow clone:

```
git clone [repo_url] --depth 1
```

The above command will fetch just the last commit .
Be aware that you may not be able to resolve merges in a shallow repository. It's often a good idea to take at least as many commits are you are going to need to backtrack to resolve merges. For example, to instead get the last 50 commits:

```
git clone [repo_url] --depth 50
```

Later, if required, you can the fetch the rest of the repository:
Version ≥ 1.8.3

```
git fetch --unshallow # equivalent of git fetch --depth=2147483647
# fetches the rest of the repository
```

Version < 1.8.3

```
git fetch --depth=1000 # fetch the last 1000 commits
```

Section 25.2: Regular Clone

To download the entire repository including the full history and all branches, type:

```
git clone <url>
```
The example above will place it in a directory with the same name as the repository name.
To download the repository and save it in a specific directory, type:
```
git clone <url> [directory]
```
For more details, visit Clone a repository.

Section 25.3: Clone a specific branch

To clone a specific branch of a repository, type --branch <branch name> before the repository url:
```
git clone --branch <branch name> <url> [directory]
```
To use the shorthand option for --branch, type -b. This command downloads entire repository and checks out <branch name>.
To save disk space you can clone history leading only to single branch with:
```
git clone --branch <branch_name> --single-branch <url> [directory]
```
106

If --single-branch is not added to the command, history of all branches will be cloned into [directory]. This can be issue with big repositories.
To later undo --single-branch flag and fetch the rest of repository use command:
```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

Section 25.4: Clone recursively

Version ≥ 1.6.5
```
git clone <url> --recursive
```
Clones the repository and also clones all submodules. If the submodules themselves contain additional submodules, Git will also clone those.

Section 25.5: Clone using a proxy

If you need to download files with git under a proxy, setting proxy server system-wide couldn't be enough. You could also try the following:
```
git config --global http.proxy http://<proxy-server>:<port>/
```
107

Part 26: Stashing

Parameter Details
show Show the changes recorded in the stash as a diff between the stashed state and its original parent.
When no <stash> is given, shows the latest one.
list
List the stashes that you currently have. Each stash is listed with its name (e.g. stash@{0} is the latest stash, stash@{1} is the one before, etc.), the name of the branch that was current when the stash was made, and a short description of the commit the stash was based on.
pop Remove a single stashed state browser showing you *every single commit in the repository ever*, regardless of whether it
is reachable or not.
You can replace gitk there with something like git log --graph --oneline --decorate if you prefer a nice graph on the console over a separate GUI app.
To spot stash commits, look for commit messages of this form:
WIP on *somebranch*: *commithash Some old commit message*
Once you know the hash of the commit you want, you can apply it as a stash:
```
git stash apply sh_hash
```
Or you can use the context menu in gitk to create branches for any unreachable commits you are interested in.
After that, you can do whatever you want with them with all the normal tools. When you' re done, just blow those
branches away again.
113

Part 27: Subtrees

Section 27.1: Create, Pull, and Backport Subtree

Create Subtree

Add a new remote called plugin pointing to the plugin's repository:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Then Create a subtree specifying the new folder prefix plugins/demo. plugin is the remote name, and master refers to the master branch on the subtree's repository:

```
git subtree add --prefix=plugins/demo plugin master
```

Pull Subtree Updates

Pull normal commits made in plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Backport Subtree Updates

1. Specify commits made in superproject to be backported:

```
git commit -am "new changes to be backported"
```

2. Checkout new branch for merging, set to track subtree repository:

```
git checkout -b backport plugin/master
```

3. Cherry-pick backports:

```
git cherry-pick -x --strategy=subtree master
```

4. Push changes back to plugin source:

```
git push plugin backport:master
```

Part 28: Renaming

Parameter Details

-f or --force Force renaming or moving of a file even if the target exists

Section 28.1: Rename Folders

To rename a folder   using this command:

```
git branch -m old_name new_name
```

Part 29: Pushing

Parameter Details

--force Overwrites the remote ref to match your local ref. *Can cause the remote repository to lose commits, so use with care*.

--verbose Run verbosely.

<remote> The remote repository that is destination of the push operation.

<refspec>... Specify what remote ref to update with what local ref or object.

After changing, staging, and committing code with Git, pushing is required to make your changes available to others
and transfers your local changes to the repository server. This topic will cover how to properly push code using Git.

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
```
Push a single commit
If you have a single commit in your branch that you want to push to a remote without pushing anything else, you can use the following
```
git push <remotename> <commit SHA>:<remotebranchname>
```
Example
Assuming a git history like this

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```
to push only commit *347d700* to remote *master* use the following command
```
git push origin 347d700:master
```
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
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 , 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.
Pushing to a new repository
To push to a repository that you haven't made yet, or is empty:
1. Create the repository on GitHub (if applicable)
2. Copy the url given to you, in the form https://github.com/USERNAME/REPO_NAME.git
3. Go to your local repository, and execute git remote add origin URL
To verify it was added, run git remote -v
4. Run git push origin master
Your code should now be on GitHub

For more information view Adding a remote repository
Explanation
Push code means that git will analyze the differences of your local commits and remote and send them to be written on the upstream. When push succeeds, your local repository and remote repository are synchronized and other users can see your commits.
For more details on the concepts of "upstream" and "downstream", see Remarks.
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
```

Important notes

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.

As a rule of thumb, only force push when:

Nobody except you pulled the changes you are trying to overwrite

You can force everyone to clone a fresh copy after the forced push and make everyone apply their changes to it (people may hate you for this).

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 29.5: Changing the default push behavior

Current updates the branch on the remote repository that shares a name with the current working branch.

```
git config push.default current
```

Simple pushes to the upstream branch, but will not work if the upstream branch is called something else.

```
git config push.default simple
```

Upstream pushes to the upstream branch, no matter what it is called.

```
git config push.default upstream
```

Matching pushes all branches that match on the local and the remote git config push.default upstream

After you've set the preferred style, use

```
git push
```

to update the remote repository.

Part 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

Section 30.3: HEAD ref

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
```

But it can also point directly to an object:

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

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.

Section 30.4: Refs

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
```

This is commonly what are called branches. However, you'll note that in **git** there is no such thing as a branch - only a ref.
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.

Section 30.5: Commit Object

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 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>
First commit!
```

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).

Section 30.6: Tree Object

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

Just as you can use ls or **dir** to list the contents of a folder, you can list the contents of a **tree** object.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
...
```

You can look up the files in a commit by first finding the hash of the **tree** in the commit, and then looking at that **tree**:

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
...
```

Section 30.7: Blob Object

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.

```
$ git cat-file -p d429810
package com.example.project
class Foo {
...
}
...
```

For example, you can browse a **tree** as above, and then look at one of the blobs in it.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
100644 blob cae391ff Readme.txt
$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
```

```
...
```

Section 30.8: Creating new Commits

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**

Cloning   is ten times faster than cloning directly   by doing the regular git-tfs clone

first. Then the new repository can be bootstrapped to work with TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

Section 31.3: git-tfs install via Chocolatey

The following assumes you will use kdiff3 for file diffing and although not essential it is a good idea.

```
C:\> choco install kdiff3
```

Git can be installed first so you can state any parameters you wish. Here all the Unix tools are also installed and 'NoAutoCrlf' means checkout as is, commit as is.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

This is all you really need to be able to install git-tfs via chocolatey.

```
C:\> choco install git-tfs
```

Section 31.4: git-tfs Check In

Launch the Check In dialog for TFVS.

```
$ git tfs checkintool
```

This will take all of your local commits and create a single check-in.

Section 31.5: git-tfs push

Push all local commits to the TFVS remote.

```
$ git tfs rcheckin
```

Note: this will fail if Check-in Notes are required. These can be bypassed by adding git-tfs-force: rcheckin to the commit message.

Part 32: Empty directories in Git

Section 32.1: Git doesn't track directories

Assume you've initialized a project with the following directory structure:

```
/build
app.js
```

Then you add everything so you've created so far and commit:

```
git init
git add .
git commit -m "Initial commit"
```

Git will only track the file app.js.

Assume you added a build step to your application and rely on the "build" directory to be there as the output directory (and you don't want to make it a setup instruction every developer has to follow), a *convention* is to include a ".gitkeep" file inside the directory and let Git track that file.

```
/build
.gitkeep
app.js
```

Then add this new file:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git will now track the file build/.gitkeep file and therefore the build folder will be made available on checkout.

Again, this is just a convention and not a Git feature.

Part 33: git-svn

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 33.2: Pushing local changes to SVN

The command
git svn dcommit
will create a SVN revision for each of your local git commits. As with SVN, your local git history must be in sync with the latest changes in the SVN repository, so if the command fails, try performing a git svn rebase first.

Section 33.3: Working locally

Just use your local git repository as a normal git repo, with the normal git commands:
git add FILE and git checkout -- FILE To stage/unstage a file
git commit To save your changes. Those commits will be local and will not be "pushed" to the SVN repo, just like in a normal git repository
git stash and git stash pop Allows using stashes
git reset HEAD --hard Revert all your local changes
git log Access all the history in the repository
git rebase -i so you can rewrite your local history freely
git branch and git checkout to create local branches
As the git-svn documentation states "Subversion is a system that is far less sophisticated than Git" so you can't use all the full power of git without messing up the history in the Subversion server. Fortunately the rules are very simple: **Keep the history linear**
This means you can make almost any git operation: creating branches, removing/reordering/squashing commits, move the history around, delete commits, etc. Anything *but merges*. If you need to reintegrate the history of local branches use git rebase instead.
When you perform a merge, a merge commit is created. The particular thing about merge commits is that they have two parents, and that makes the history non-linear. Non-linear history will confuse SVN in the case you "push"
a merge commit to the repository.
However do not worry: **you won't break anything if you "push" a git merge commit to SVN**. If you do so, when

the git merge commit is sent to the svn server it will contain all the changes of all commits for that merge, so you will lose the history of those commits, but not the changes in your code.

Section 33.4: Getting the latest changes

and applies them *on top* of your local commits in your
current branch.
You can also use the command
git svn fetch
to retrieve the changes and bring them to your local machine but without applying them to
your local branch.

Section 33.5: Handling empty folders

git does not recognice the concept of folders, it just works with files and their filepaths. This means git does not track empty folders. SVN, however, does. Using git-svn means that, by default, *any change you do involving empty folders with git will not be propagated to SVN*.
Using the --rmdir flag when issuing a comment corrects this issue, and removes an empty folder in SVN if you locally delete the last file inside it:
git svn dcommit --rmdir
Unfortunately **it does not removes existing empty folders**: you need to do it manually.
To avoid adding the flag each time you do a dcommit, or to play it safe if you are using a git GUI tool (like SourceTree) you can set this behaviour as default with the command:

```
git config --global svn.rmdir true
```
This changes your .gitconfig file and adds these lines:
```
[svn]
rmdir = true
```
To remove all untracked files and folders that should be kept empty for SVN use the git command:
```
git clean -fd
```
Please note: the previous command will remove all untracked files and empty folders, even the ones that should be
tracked by SVN! If you need to generate againg the empty folders tracked by SVN use the command
```
git svn mkdirs
```
In practices this means that if you want to cleanup your workspace **<repo>** rather than the local repository.

--exec=<git-upload-archive> Used with --remote to specify the path to the **<git-upload-archive** on the remote.

<tree-ish> The tree or commit to produce an archive for.

<path>

Without an optional parameter, all files and directories in the current working directory are included in the archive. If one or more paths are specified, only these are included.

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.

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 **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
```
Create an archive of files inside a specific sub directory (sub-dir) of revision HEAD:
```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Part 35: Rewriting history with filterbranch

Section 35.1: Changing the author of commits

You can use an environment filter to change the author of commits. Just modify and export $GIT_AUTHOR_NAME in
the script to change who authored the commit.
Create a file filter.sh with contents like so:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
export GIT_AUTHOR_NAME="Author to Change To"
export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Then run filter-branch to git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Section 36.2: Migrate .

Next, you need to generate an authors file. Subversion tracks changes by the committer's username only. Git, however, uses two pieces of information to distinguish a user: a real name and an email address. The following command will generate a text file mapping the subversion usernames to their Git equivalents:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

where **<svn-repo>** is the URL of the subversion repository you wish to convert. After running this command, the contributors' identification information will be mapped in authors.txt. The email addresses will be of the form **<username>**@mycompany.com. In the authors file, you will need to manually change each person's default name (which by default has become their username) to their actual names. Make sure to also check all of the email addresses for correctness before proceeding.
The following command will clone an svn repo as a Git one:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

where **<svn-repo>** is the same repository URL used above and **<git-repo-name>** is the folder name in the current directory to clone the repository into. There are a few considerations before using this command:
The --stdlayout flag in order to eliminate network overhead.
git svn clone imports the subversion branches (and trunk) as remote branches including subversion tags (remote branches prefixed with tags/). To convert these to actual branches and tags, run the following commands on a Linux machine in the order they are provided. After running them, **git branch** -a should show the correct branch names, and **git tag** -l should show the repository tags.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname; do git
tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git branch
"$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname"; done
```

The conversion to your remote.

```
git add .
git commit -a -m "Coverted solution source control
```

with the standard layout (ie. branches, tags and trunk at the root level of the
repository):

```
$ svn2git http://svn.example.com/path/to/repo
```

To migrate a svn repository which is not in standard layout:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

In case you do not want to migrate (or do not have) branches, tags or trunk you can use options --notrunk, --nobranches, and --notags.
For example, $ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches

will migrate only trunk history.
To reduce the space required by your new repository you may want to exclude any directories or files you once added while you should not have (eg. build directory or archives):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Post-migration optimization

If you already have a few thousand of commits (or more) in your newly created git repository, you may want to reduce space used before pushing your repository on a remote. This can be done using the following command:

```
$ git gc --aggressive
```

Note: The previous command can take up to several hours on large repositories (tens of thousand of commits and/or hundreds of megabytes of history).

133

Part 37: Show

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

on machines that have no network connection.
Bundles allow you to package git objects and references in a repository on one machine and import those into a repository on another.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Somehow transfer the **changes_between_tags.bundle** file to the remote machine; e.g., via thumb drive. Once you have it there:

```
git bundle verify changes_between_tags.bundle # make sure bundle arrived intact
git checkout [some branch] # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference
```

you can bundle up the deltas; put the changes on, e.g., a thumb drive, and merge them back into the local repository so the two can stay in sync without requiring direct git, ssh, rsync, or http protocol access between the machines.

136

Part 40: Display commit history graphically with Gitk

Section 40.1: Display commit history for one file

```
gitk path/to/myfile
```

Section 40.2: Display all commits between two commits

Let's say you have two commits d9e1db9 and 5651067 and want to see what happened between them. d9e1db9 is the oldest ancestor and 5651067 is the final descendant in the chain of commits.

```
gitk --ancestry-path d9e1db9 5651067
```

Section 40.3: Display commits since version tag

If you have the version tag v2.3 you can display all commits since that tag.

```
gitk v2.3..
```

Part 41: Bisecting/Finding faulty commits

Section 41.1: Binary search (git bisect)

git bisect allows you to find which commit introduced a bug using a binary search.
Start by bisecting a session by providing two commit references: a good commit before the bug, and a bad commit after the bug. Generally, the bad commit is HEAD.

```
# start the git bisect session
$ git bisect start
# give a commit where the bug doesn't exist
$ git bisect good 49c747d
# give a commit where the bug exist
$ git bisect bad HEAD
```

git starts a binary search: It splits the revision in half and switches the repository to the intermediate revision.
Inspect the code to determine if the revision is good or bad:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good
# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

git will continue to run the binary search on each remaining subset of bad revisions depending on your instructions. git will present a single revision that, unless your flags were incorrect, will represent exactly the revision where the bug was introduced.
Afterwards remember to run git bisect reset to end the bisect session and return to HEAD.

```
$ git bisect reset
```

If you have a script that can check for the bug, you can automate the process with:

```
$ git bisect run [script] [arguments]
```

Where [script] is the path to your script and [arguments] is any arguments that should be passed to your script.
Running this command will automatically run through the binary search, executing git bisect good or git bisect bad at each step depending on the exit code of your script. Exiting with 0 indicates good, while exiting with 1-124, 126, or 127 indicates bad. 125 indicates that the script cannot test that revision (which will trigger a git bisect skip).

Section 41.2: Semi-automatically find a faulty commit

Imagine you are on the master branch and something is not working as expected (a regression was introduced), but
you don't know where. All you know is, that is was working in the last release (which was e.g., tagged or you know the commit hash, lets take old-rel here).

Git has help for you, finding the faulty commit which introduced the regression with a very low number of steps (binary search).
First of all start bisecting:

```
git bisect start master old-rel
```

This will tell git that master is a broken revision (or the first broken version) and old-rel is the last known version.
Git will now check out a detached head in the middle of both commits. Now, you can do your testing. Depending on

whether it works or not issue

```
git bisect good
```

or

```
git bisect bad
```

. In case this commit cannot be tested, you can easily git reset and test that one, git willl take care of this.
After a few steps git will output the faulty commit hash.
In order to abort the bisect process just issue

```
git bisect reset
```

and git will restore the previous state.

139

Part 42: Blaming

Parameter Details

filename Name of the file for which details need to be checked
-f Show the file name in the origin commit
-e Show the author email instead of author name
-w Ignore white spaces while making a comparison between child and parent's version
-L start,end Show only the given line range Example: `git blame -L 1,2 [filename]`
--show-stats Shows additional statistics at end of blame output
-l Show long rev (Default: off)
-t Show raw timestamp (Default: off)
-reverse Walk history forward instead of backward
-p, --porcelain Output for machine consumption
-M Detect moved or copied lines within a file
-C In addition to -M, detect lines moved or copied .

Section 43.2: Symbolic ref names: branches, tags, remotetracking branches

```
$ git log master # specify branch
$ git show v1.0 # specify tag
$ git show HEAD # specify current branch
$ git show origin # specify default remote-tracking branch for remote 'origin'
```

You can specify revision using a symbolic ref name, which includes branches (for example 'master', 'next', 'maint'), tags (for example 'v1.0', 'v0.6.3-rc2'), remote-tracking branches (for example 'origin', 'origin/master'), and special refs such as 'HEAD' for current branch.
If the symbolic ref name is ambiguous, for example if you have both branch and tag named 'fix' (having branch and tag with the same name is not recommended), you need to specify the kind of ref you want to use:

```
$ git show heads/fix # or 'refs/heads/fix', to specify branch
$ git show tags/fix # or 'refs/tags/fix', to specify tag
```

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.

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving  .
$ 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}
```

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.

143

This syntax is composable.

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'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND
```

--add Adds <name> to list of currently tracked branches (set-branches)
--add Instead of changing some URL, new URL is added (set-url)
--all Push all branches.
--delete All urls matching <url> are deleted. (set-url)
--push Push URLS are manipulated instead of fetch URLS
-n The remote heads are not queried first with git ls-remote **<**name**>**, cached information is used
instead

--dry-run report what branches will be pruned, but do not actually prune them
--prune Remove remote branches that don't have a local counterpart

Section 45.1: Display Remote Repositories

To list all configured remote repositories, use git remote.
It shows the short name (aliases) of each remote handle that you have configured.

```
$ git remote
premium
premiumPro
origin
```

To show more detailed information, the --verbose or -v flag can be used. The output will include the URL and the type of the remote (push or pull):

```
$ git remote -v
premiumPro https://github.com/user/CatClickerPro.git (fetch)
premiumPro https://github.com/user/CatClickerPro.git (push)
premium https://github.com/user/CatClicker.git (fetch)
premium https://github.com/user/CatClicker.git (push)
origin https://github.com/ud/starter.git (fetch)
origin https://github.com/ud/starter.git (push)
```

Section 45.2: Change remote url of your Git repository

You may want to do this if the remote repository is migrated. The command for changing the remote url is:

```
git remote set-url
```

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

```
git remote -v
origin https://bitbucket.com/develop/myrepo.git (fetch)
origin https://bitbucket.com/develop/myrepo.git (push)
```

Change your remote url:

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

Check again your remote url:

```
git remote -v
origin https://localserver/develop/myrepo.git (fetch)
origin https://localserver/develop/myrepo.git (push)
```

Section 45.3: Remove a Remote Repository

Remove the remote named **<name>**. All remote-tracking branches and configuration settings for the remote are removed.
To remove a remote repository dev:

```
git remote rm dev
```

Section 45.4: Add a Remote Repository

To add a remote, use git remote add in the root of your local repository.
For adding a remote Git repository <url> as an easy short name <name> use

```
git remote add <name> <url>
```

The command git fetch <name> can then be used to create and update remote-tracking branches **<name>**/**<branch>**.

Section 45.5: Show more information about remote repository

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

```
git remote show origin
```

result:

```
remote origin
Fetch URL: https://localserver/develop/myrepo.git
Push URL: https://localserver/develop/myrepo.git
```

```
HEAD branch: master
Remote branches:
master tracked
Local branches configured for 'git pull':
master merges with remote master
```
149
```
Local refs configured for 'git push':
master pushes to master (up to date)
```

Section 45.6: Rename a Remote Repository

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
```

150

Part 46: Git Large File Storage (LFS)

Section 46.1: Declare certain file types to store externally

A common workflow for using Git LFS is to declare which files are intercepted through a rules-based system, just like .gitignore files.
Much of time, wildcards are used to pick certain file-types to blanket track.
e.g. `git lfs track "*.psd"`
When a file matching the above pattern is added them committed, when it is then pushed to the remote, it will be uploaded separately, with a pointer replacing the file in the remote repository.
After a file has been tracked with lfs, your .gitattributes file will be updated accordingly. Github recommends committing your local .gitattributes file, rather than working with a global .gitattributes file, to help ensure you don't have any issues when working with different projects.

Section 46.2: Set LFS config for all clones

To set LFS options that apply to all clones, create and commit a file named .lfsconfig at the repository root. This file can specify LFS options the same way as allowed in .git/config.
For example, to exclude a certain file locations on your you can run the following

```
find $HOME -type d -name ".git"
```

Assuming you have **locate**, this should be much faster:

```
locate .git |grep git$
```

If you have gnu **locate** or mlocate, this will select only the git dirs:

```
locate -ber \\.git$
```

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
```

155

Part 49: git send-email

Section 49.1: Use git send-email with Gmail

Background: if you work on a project like the Linux kernel, rather than make a pull request you will need to submit your commits to a listserv for review. This entry details how to use git-send email with Gmail.

Add the following to your .gitconfig file:

```
[sendemail]
smtpserver = smtp.googlemail.com
smtpencryption = tls
smtpserverport = 587
smtpuser = name@gmail.com
```

Then on the web: Go to Google -> My Account -> Connected Apps & Sites -> Allow less secure apps -> Switch ON
To create a patch set:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Then send the patches to a listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

To create and send updated version (version 2 in this example) of the patch:

```
git format-patch -v 2 HEAD~~~~ ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Section 49.2: Composing

-- , and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from any ref, not just HEAD. Gitk' s interface looks like this:

Figure 1-1. The gitk history viewer.

On the top is something that looks a bit like the output of git log --graph; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

You can access many git related functions via right-click on a branch name or a commit message. For example checking out a different branch or cherry pick a commit is easily done with one click.

158

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke then Tortoise Git -> Create Branch...

166

New window will open -> Give branch a name -> Tick the box Switch to new branch (Chances are you want to start working with it after branching). -> Click OK and you should be done.

167

Part 53: External merge and di tools

Section 53.1: Setting up KDi3 as merge tool

The following should be added to your global .gitconfig file

```
[merge]
tool = kdiff3
[mergetool "kdiff3"]
path = D:/Program Files (x86)/KDiff3/kdiff3.exe
keepBackup = false
keepbackup = false
trustExitCode = false
```

Remember to set the path property to point to the directory where you have installed KDiff3

Section 53.2: Setting up KDi3 as di tool

```
[diff]
tool = kdiff3
guitool = kdiff3
[difftool "kdiff3"]
path = D:/Program Files (x86)/KDiff3/kdiff3.exe
```

```
cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Section 53.3: Setting up an IntelliJ IDE as merge tool (Windows)

```
[merge]
tool = intellij
[mergetool "intellij"]
cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd
$(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") && pwd)/$(basename
"$MERGED")\"
keepBackup = false
keepbackup = false
trustExitCode = true
```

The one gotcha here is that this cmd property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in Program Files (x86), you'll have to create a symlink

Section 53.4: Setting up an IntelliJ IDE as di tool (Windows)

```
[diff]
tool = intellij
guitool = intellij
[difftool "intellij"]
path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname "$LOCAL")
&& pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")\"
```

The one gotcha here is that this cmd property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in Program Files (x86), you'll have to create a symlink

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
```

Part 54: Update Object Name in Reference

Section 54.1: Update Object Name in Reference

Use

Update the object name which is stored in reference

SYNOPSIS

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

General Syntax

1. Dereferencing the symbolic refs, update the current branch head to the new object.

```
git update-ref HEAD <newvalue>
```

2. Stores the newvalue in ref, after verify that the current value of the ref matches oldvalue.

```
git update-ref refs/head/master <newvalue> <oldvalue>
```

above syntax updates the master branch head to newvalue only if its current value is oldvalue.

Use -d flag to deletes the named **<ref>** after verifying it still contains **<oldvalue>**.

Use --create-reflog, update-ref will create a reflog for each ref even if one would not ordinarily be created.

Use -z flag to specify in NUL-terminated format, which has values like update, create, delete, verify.

Update

Set **<ref>** to **<newvalue>** after verifying **<oldvalue>**, if given. Specify a zero **<newvalue>** to ensure the ref does not exist after the update and/or a zero **<oldvalue>** to make sure the ref does not exist before the update.

Create

Create **<ref>** with **<newvalue>** after verifying it does not exist. The given **<newvalue>** may not be zero.

Delete

Delete **<ref>** after verifying it exists with **<oldvalue>**, if given. If given, **<oldvalue>** may not be zero.

Verify

Verify **<ref>** against **<oldvalue>** but do not change it. If **<oldvalue>** zero or missing, the ref must not exist.

170

Part 55: Git Branch Name on Bash Ubuntu

This documentation deals with the **branch name** of the git on the **bash** terminal. We developers need to find the git branch name very frequently. We can add the branch name along with the path to the current directory.

Section 55.1: Branch Name in terminal

What is PS1

PS1 denotes Prompt String 1. It is the one of the prompt available in Linux/UNIX shell. When you open your terminal, it will display the content defined in PS1 variable in your bash prompt. In order to add branch name to bash prompt we have to edit the PS1 variable (set value of PS1 in ~/.bash_profile).

Display git branch name

Add following lines to your ~/.bash_profile

```
git_branch() {
git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

This git_branch function will find the branch name we are on. Once we are done with this changes we can navigate to the git repo on the terminal and will be able to see the branch name.

171

Part 56: Git Client-Side Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Section 56.1: Git pre-push hook

pre-push script is called by **git push** after it has checked the remote status, but before anything has been pushed. If this script exits with a non-zero status nothing will be pushed.

This hook is called with the following parameters:

$1 -- Name of the remote to which the push is being done (Ex: origin)

$2 -- URL to which the push is being done (Ex: https://:/.git)

Information about the commits which are being pushed is supplied as lines to the standard input in the form:

<local_ref> <local_sha1> <remote_ref> <remote_sha1>

Sample values:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Below example pre-push script was taken   is initialized with **git init**

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).
remote="$1"
url="$2"
z40=0000000000000000000000000000000000000000
while read local_ref local_sha remote_ref remote_sha
do
if [ "$local_sha" = $z40 ]
then
# Handle delete
:
else
if [ "$remote_sha" = $z40 ]
then
# New branch, examine all commits
range="$local_sha"
else
# Update to existing branch, examine new commits
range="$remote_sha..$local_sha"
fi
```

```
# Check for WIP commit
commit=`git rev-list -n 1 --grep '^WIP' "$range"`
if [ -n "$commit" ]
then
echo >&2 "Found WIP commit in $local_ref, not pushing"
exit 1
fi
fi
done
exit 0
```

Section 56.2: Installing a Hook

The hooks are all stored in the hooks sub directory of the Git directory. In most projects, that's .git/hooks.
To enable a hook script, put a file in the hooks subdirectory of your .git directory that is named appropriately (without any extension) and is executable.

Part 57: Git rerere

rerere (reuse recorded resolution) allows you to tell git to remember how you resolved a hunk conflict. This allows it to be automatically resolved the next time that git encounters the same conflict.

Section 57.1: Enabling rerere

To enable rerere run the following command:

```
$ git config --global rerere.enabled true
```

This can be done in a specific repository as well as globally.

Part 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.)
```

175

Part 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*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

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:

176

```
git push origin tag-name
```

Push all the tags at once

```
git push origin --tags
```

177

Part 60: Tidying up your local and remote repository

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
```

Part 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

Section 61.2: Usage

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diffoptions>]
<tree-ish> [<tree-ish>] [<path>...]
```

Option	Explanation
-r	diff recursively
--root	include the initial commit as diff against /dev/null

Section 61.3: Common di options

Option	Explanation
-z	output diff-raw with lines terminated with NUL.
-p	output patch format.
-u	synonym for -p.
--patch-with-raw	output both a patch and the diff-raw format.
--stat	show diffstat instead of patch.
--numstat	show numeric diffstat instead of patch.
--patch-with-stat	output a patch and prepend its diffstat.
--name-only	show only names of changed files.
--name-status	show names and status of changed files.
--full-index	show full object name on index lines.
--abbrev=<n>	abbreviate object names in diff-tree header and diff-raw.
-R	swap input file pairs.
-B	detect complete rewrites.
-M	detect renames.
-C	detect copies.
--find-copies-harder	try unchanged files as candidate for copy detection.
-l<n>	limit rename attempts up to paths.
-O	reorder diffs according to the .
-S	find filepair whose only one side contains the string.
--pickaxe-all	show all files diff when -S is used and hit is found.
-a --text	treat all files as text.