Chapter 1: Getting started with Git

Version	Release Date
2.13	2017-05-10
2.12	2017-02-24
2.11.1	2017-02-02
2.11	2016-11-29
2.10.2	2016-10-28
2.10	2016-09-02
2.9	2016-06-13
2.8	2016-03-28
2.7	2015-10-04
2.6	2015-09-28
2.5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2.0	2014-05-28
1.9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02
1.5	2007-02-14
1.4	2006-06-10
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

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 from a server to the local machine.

For example, to clone a GitHub project:

```
cd <path where you would like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

To clone a BitBucket project:

```
cd <path where you would like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

This creates a directory called `projectname` on the local machine, containing all the files in the remote Git repository. This includes source files for the project, as well as a `.git` sub-directory which contains the entire history and configuration for the project.

To specify a different name of the directory, e.g. `MyFolder`:

```
git clone https://github.com/username/projectname.git MyFolder
```

Or to clone in the current directory:

```
git clone https://github.com/username/projectname.git .
```

Note:

1. When cloning to a specified directory, the directory must be empty or non-existent.
2. You can also use the `ssh` version of the command:

   ```
   git clone git@github.com:username/projectname.git
   ```

The `https` version and the `ssh` version are equivalent. However, some hosting services such as GitHub recommend that you use `https` rather than `ssh`.

Section 1.3: Sharing code

To share your code you create a repository on a remote server to which you will copy your local repository.

To minimize the use of space on the remote server you create a bare repository: 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
```

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 from HTTPS to SSH you can run the following command:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

In order to clone a new 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 from source or to install an existing package for your platform.

Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you're on a very up-to-date distro or are using backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has yum (such as Fedora) or apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

```
$ yum install curl-devel expat-devel gettext-devel \
```

```
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

http://git-scm.com/download Then, compile and install:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git
```

Or if you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ apt-get install git
```

Installing on Mac

There are three easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the SourceForge page.

http://sourceforge.net/projects/git-osx-installer/

Figure 1-7. Git OS X installer. The other major way is to install Git via MacPorts (http://www.macports.org). If you have MacPorts installed, install Git via

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

You don't have to add all the extras, but you'll probably want to include +svn in case you ever have to use Git with Subversion repositories (see Chapter 8).

Homebrew (http://brew.sh/) is another alternative to install Git. If you have Homebrew installed, install Git via

```
$ brew install git
```

Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it:

```
http://msysgit.github.io
```

After it's installed, you have both a command-line version (including an SSH client that will come in handy later) and the standard GUI.

Note on Windows usage: you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of single quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

Chapter 2: Browsing the history

Parameter	Explanation
-q, --quiet	Quiet, suppresses diff output
--source	Shows source of commit
--use-mailmap	Use mail map file (changes user info for committing user)
--decorate[=...]	Decorate options
--L <n,m:file>	Show log for specific range of lines in a file, counting from 1. Starts from line n, goes to line m. Also shows diff.
--show-signature	Display signatures of signed commits
-i, --regexp-ignore-case	Match the regular expression limiting patterns without regard to letter case

Section 2.1: "Regular" Git Log

```
git log
```

will display all your commits with the author and hash. This will be shown over multiple lines per commit. (If you wish to show a single line per commit, look at onelineing). Use the q key to exit the log.

> By default, with no arguments, git log lists the commits made in that repository 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 (from **Free Code Camp** repository):

```
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 (from **Free Code Camp** 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 (from Trello Scientist repository)

```
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
```

Chapter 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 from a specific remote:

```
git fetch [remote-name] --prune
```

To prune deleted branches from *all* remotes:

```
git fetch --all --prune
```

Section 3.5: Updating from Upstream Repository

Assuming you set the upstream (as in the "setting an upstream repository")

```
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
```

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 from 'origin' to 'destination'
```

Verify new name

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

=== Posible Errors ===

1. Could not rename config section 'remote.[old name]' to 'remote.[new name]'

   This error means that the remote you tried the old remote name (**origin**) doesn't exist.

2. Remote [new name] already exists.

   Error message is self explanatory.

Section 3.11: Show information about a Specific Remote

Output some information about a known remote: `origin`

```
git remote show origin
```

Print just the remote's URL:

```
git config --get remote.origin.url
```

With 2.7+, it is also possible to do, which is arguably better than the above one that uses the `config` command.

```
git remote get-url origin
```

Section 3.12: Set the URL for a Specific Remote

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

```
git remote set-url remote-name url
```

Section 3.13: Get the URL for a Specific Remote

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

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

By default, this will be

```
git remote get-url origin
```

Section 3.14: Changing a Remote Repository

To change the URL of the repository 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
```

Chapter 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 from being tracked, but also select subsection of changes to put in the index, by selecting chunks of changes to be added, splitting those chunks, or even editing the diff. Many graphical commit tools for Git (like e.g. `git gui`) include such feature; this might be easier to use than the command line version.

It is very useful (1) if you have entangled changes in the working directory that you want to put in separate commits, and not all in one single commit (2) if you are in the middle of an interactive rebase and want to split too large commit.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

The top half of this output shows the current state of the index broken up into staged and unstaged columns:

1. `index.js` has had 4 lines added and 4 lines removed. It is currently not staged, as the current status reports "unchanged." When this file becomes staged, the `+4/-4` bit will be transferred to the staged column and the unstaged column will read "nothing."
2. `package.json` has had one line added and has been staged. There are no further changes since it has been staged as indicated by the "nothing" line under the unstaged column.

The bottom half shows what you can do. Either enter a number (1-8) or a letter (s, u, r, a, p, d, q, h).

`status` shows output identical to the top part of the output above.

`update` allows you to make further changes to the staged commits with additional syntax.

`revert` will revert the staged commit information back to HEAD.

`add untracked` allows you to add filepaths previously untracked by version control.

`patch` allows for one path to be selected out of an output similar to `status` for further analysis.

`diff` displays what will be committed.

`quit` exits the command.

`help` presents further help on using this command.

Section 4.5: Show Staged Changes

To display the hunks that are staged for commit:

```
git diff --cached
```

Section 4.6: Staging A Single File

To stage a file for committing, run

```
git add <filename>
```

Section 4.7: Stage deleted files

```
git rm filename
```

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

```
git rm --cached filename
```

Chapter 5: Ignoring Files and Folders

This topic illustrates how to avoid adding unwanted files (or file changes) in a Git repo. There are several ways (global or local `.gitignore`, `.git/exclude`, `git update-index --assume-unchanged`, and `git update-index --skip-tree`), but keep in mind Git is managing *content*, which means: ignoring actually ignores a folder *content* (i.e. files). An empty folder would be ignored by default, since it cannot be added anyway.

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

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

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

1. temporary resources e.g. caches, log files, compiled code, etc.
2. local configuration files that should not be shared with other developers
3. files containing secret information, such as login passwords, keys and credentials

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

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/
```

```
# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

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

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

Most `.gitignore` files are standard across various languages, so to get started, here is set of sample `.gitignore` files listed by language from which to clone or copy/modify into your project. Alternatively, for a fresh project you may consider auto-generating a starter file using an online tool.

Other forms of .gitignore

`.gitignore` files are intended to be committed as part of the repository. 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 from the index:

```
git rm --cached <file>
```

This will remove the file from the repository and prevent further changes from being tracked by Git. The `--cached` option will make sure that the file is not physically deleted.

Note that previously added contents of the file will still be visible via the Git history.

Keep in mind that if anyone else pulls from the repository after you removed the file from the index, **their copy will be physically deleted**.

You can make Git pretend that the working directory version of the file is up to date and read the index version instead (thus ignoring changes in it) with "skip worktree" bit:

```
git update-index --skip-worktree <file>
```

Writing is not affected by this bit, content safety is still first priority. You will never lose your precious ignored changes; on the other hand this bit conflicts with stashing: to remove this bit, use

```
git update-index --no-skip-worktree <file>
```

It is sometimes ***wrongly*** recommended to lie to Git and have it assume that file is unchanged without examining it. It looks at first glance as ignoring any further changes to the file, without removing it from its index:

```
git update-index --assume-unchanged <file>
```

This will force git to ignore any change made in the file (keep in mind that if you pull any changes to this file, or you stash it, **your ignored changes will be lost**)

If you want git to "care" about this file again, run the following command:

```
git update-index --no-assume-unchanged <file>
```

Section 5.6: Ignore files locally without committing ignore rules

`.gitignore` ignores files locally, but it is intended to be committed to the repository 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/
```

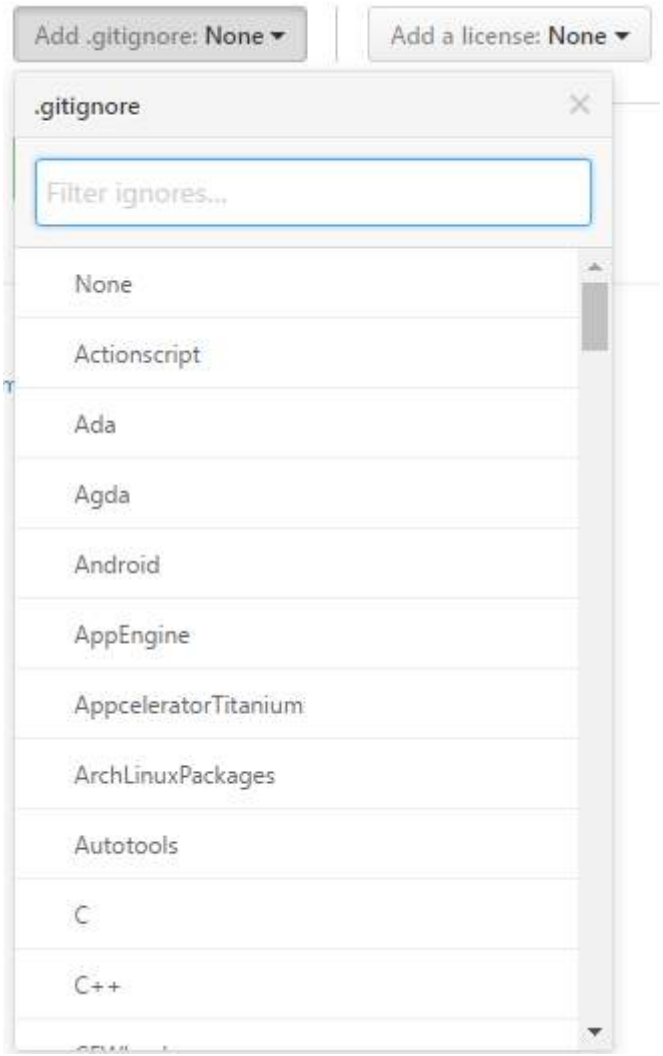
Section 5.9: Prefilled .gitignore Templates

If you are unsure which rules to list in your `.gitignore` file, or you just want to add generally accepted exceptions

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:

```
# /.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
```

```
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 - `--untracked-files=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 from file `file1.c`:

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

You don't want to publish `NOCOMMIT` lines anywhere.

Create "nocommit" filter by adding this to Git config file like `.git/config`:

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Add (or create) this to `.git/info/attributes` or `.gitmodules`:

```
file1.c filter=nocommit
```

And your NOCOMMIT lines are hidden from Git.

Caveats:

- Using clean filter slows down processing of files, especially on Windows.
- The ignored line may disappear from file when Git updates it. It can be counteracted with a smudge filter, but it is trickier.
- Not tested on Windows

Section 5.14: Ignoring changes in tracked files. [stub]

.gitignore and `.git/info/exclude` work only for untracked files.

To set ignore flag on a tracked file, use the command update-index:

```
git update-index --skip-worktree myfile.c
```

To revert this, use:

```
git update-index --no-skip-worktree myfile.c
```

You can add this snippet to your global git config to have more convenient `git hide`, `git unhide` and `git hidden` commands:

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

You can also use the option --assume-unchanged with the update-index function

```
git update-index --assume-unchanged <file>
```

If you want to watch this file again for the changes, use

```
git update-index --no-assume-unchanged <file>
```

When --assume-unchanged flag is specified, the user promises not to change the file and allows Git to assume that the working tree file matches what is recorded in the index.Git will fail in case it needs to modify this file in the index e.g. when merging in a commit; thus, in case the assumed-untracked file is changed upstream, you will need to handle the situation manually.The focus lies on performance in this case.

While --skip-worktree flag is useful when you instruct git not to touch a specific file ever because the file is going to be changed locally and you don't want to accidentally commit the changes (i.e configuration/properties file configured for a particular environment). Skip-worktree takes precedence over assume-unchanged when both are set.

Section 5.15: Clear already committed files, but included in .gitignore

Sometimes it happens that a file was being tracked by git, but in a later point in time was added to .gitignore, in order to stop tracking it. It's a very common scenario to forget to clean up such files before its addition to .gitignore. In this case, the old file will still be hanging around in the repository.

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 from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

Chapter 6: Git Diff

Parameter	Details
-p, -u, --patch	Generate patch
-s, --no-patch	Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`
--raw	Generate the diff in raw format
--diff-algorithm=	Choose a diff algorithm. The variants are as follows: `myers`, `minimal`, `patience`, `histogram`
--summary	Output a condensed summary of extended header information such as creations, renames and mode changes
--name-only	Show only names of changed files
--name-status	Show names and statuses of changed files The most common statuses are M (Modified), A (Added), and D (Deleted)
--check	Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code
--full-index	Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output
--binary	In addition to `--full-index`, output a binary diff that can be applied with `git apply`
-a, --text	Treat all files as text.
--color	Set the color mode; i.e. use `--color=always` if you would like to pipe a diff to less and keep git's coloring

Section 6.1: Show differences in working branch

```
git diff
```

This will show the *unstaged* changes on the current branch from the commit before it. It will only show changes relative to the index, meaning it shows what you *could* add to the next commit, but haven't. To add (stage) these changes, you can use `git add`.

If a file is staged, but was modified after it was staged, `git diff` will show the differences between the current file and the staged version.

Section 6.2: Show changes between two commits

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

E.g.: Show the changes made in the last 3 commits:

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

Note: the two dots (..) is optional, but adds clarity.

This will show the textual difference between the commits, regardless of where they are in the tree.

Section 6.3: Show differences for staged files

```
git diff --staged
```

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

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

```
git diff --cached
```

Which is just a synonym for `--staged` or

```
git status -v
```

Which will trigger the verbose settings of the `status` command.

Section 6.4: Comparing branches

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

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

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

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

Using only one parameter such as

git diff original

is equivalent to

git diff original..HEAD

Section 6.5: Show both staged and unstaged changes

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

```
git diff HEAD
```

NOTE: You can also use the following command:

```
git status -vv
```

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

Section 6.6: Show differences for a specific file or directory

```
git diff myfile.txt
```

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

This also works for directories:

```
git diff documentation
```

The above shows the changes between the previous commit of all files in the specified directory (`documentation/`) and the locally-modified versions of these files, that have not yet been staged.

To show the difference between some version of a file in a given commit and the local `HEAD` version you can specify the commit you want to compare against:

```
git diff 27fa75e myfile.txt
```

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

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

To show the difference between the version specified by the hash `ada9b57` and the latest commit on the branch `my_branchname` for only the relative directory called `my_changed_directory/` you can do this:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Section 6.7: Viewing a word-diff for long lines

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

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

```
-Hello world
+Hello world!
```

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

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

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

```
@@ -1 +1 @@
Hello worldworld!
```

Section 6.8: Show differences between current version and last version

```
git diff HEAD^ HEAD
```

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

Section 6.9: Produce a patch-compatible diff

Sometimes you just need a diff to apply using patch. The regular `git --diff` does not work. Try this instead:

```
git diff --no-prefix > some_file.patch
```

Then somewhere else you can reverse it:

```
patch -p0 < some_file.patch
```

Section 6.10: difference between two commit or branch

To view difference between two branch

```
git diff <branch1>..<branch2>
```

To view difference between two branch

```
git diff <commitId1>..<commitId2>
```

To view diff with current branch

```
git diff <branch/commitId>
```

To view summary of changes

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

To view files that changed after a certain commit

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

To view files that are different than a branch

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

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

```
git diff --name-only <commitId> <folder_path>
```

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

```
git difftool -t meld --dir-diff
```

will show the working directory changes. Alternatively,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

will show the differences between 2 specific commits.

Section 6.12: Diff UTF-16 encoded text and binary plist files

You can diff UTF-16 encoded files (localization strings file os iOS and macOS are examples) by specifying how git should diff these files.

Add the following to your `~/.gitconfig` file.

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` is a program to convert different encodings.

Then edit or create a `.gitattributes` file in the root of the repository 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
```

Chapter 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 from the current directory, it will undo all changes in the working copy.

```
git checkout -- .
```

To only undo parts of the changes use `--patch`. You will be asked, for each change, if it should be undone or not.

```
git checkout --patch -- dir
```

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

```
git reset --hard
```

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

With local commits that you have yet to push to a remote you can also do a soft reset. You can thus rework the files

and then the commits.

```
git reset HEAD~2
```

The above example would unwind your last two commits and return the files to your working copy. You could then make further changes and new commits.

Beware: All of these operations, apart from soft resets, will permanently delete your changes. For a safer option, use `git stash -p` or `git stash`, respectively. You can later undo with `stash pop` or delete forever with `stash drop`.

Section 7.3: Using reflog

If you screw up a rebase, one option to start again is to go back to the commit (pre rebase). You can do this using `reflog` (which has the history of everything you've done for the last 90 days - this can be configured):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

You can see the commit before the rebase was `HEAD@{3}` (you can also checkout the hash):

```
git checkout HEAD@{3}
```

Now you create a new branch / delete the old one / try the rebase again.

You can also reset directly back to a point in your `reflog`, but only do this if you're 100% sure it's what you want to do:

```
git reset --hard HEAD@{3}
```

This will set your current git tree to match how it was at that point (See Undoing Changes).

This can be used if you're temporarily seeing how well a branch works when rebased on another branch, but you don't want to keep the results.

Section 7.4: Undoing merges

Undoing a merge not yet pushed to a remote

If you haven't yet pushed your merge to the remote repository 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 from the branch. However, you will need to know what SHA to reset back to, this can be tricky as your `git log` will now show commits from both branches. If you reset to the wrong commit (e.g. one on the other branch) **it can destroy committed work.**

```
> git reset --hard <last commit from the branch you are on>
```

Or, assuming the merge was your most recent commit.

```
> git reset HEAD~
```

A revert is safer, in that it won't destroy committed work, but involves more work as you have to revert the revert before you can merge the branch back in again (see the next section).

Undoing a merge pushed to a remote

Assume you merge in a new feature (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

Afterwards you discover that the feature you just merged in broke the system for other developers, it must be undone right away, and fixing the feature itself will take too long so you simply want to undo the merge.

```
> git revert -m 1 17a51fd
...
> git push
...
   17a51fd..e443799  master -> master
```

At this point the gremlins are out of the system and your fellow developers have stopped yelling at you. However, we are not finished just yet. Once you fix the problem with the add-gremlins feature you will need to undo this revert before you can merge back in.

```
> git checkout feature/add-gremlins
...
   #Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

At this point your feature is now successfully added. However, given that bugs of this type are often introduced by merge conflicts a slightly different workflow is sometimes more helpful as it lets you fix the merge conflict on your branch.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
```

```
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Section 7.5: Revert some existing commits

Use git revert to revert existing commits, especially when those commits have been pushed to a remote repository. 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.

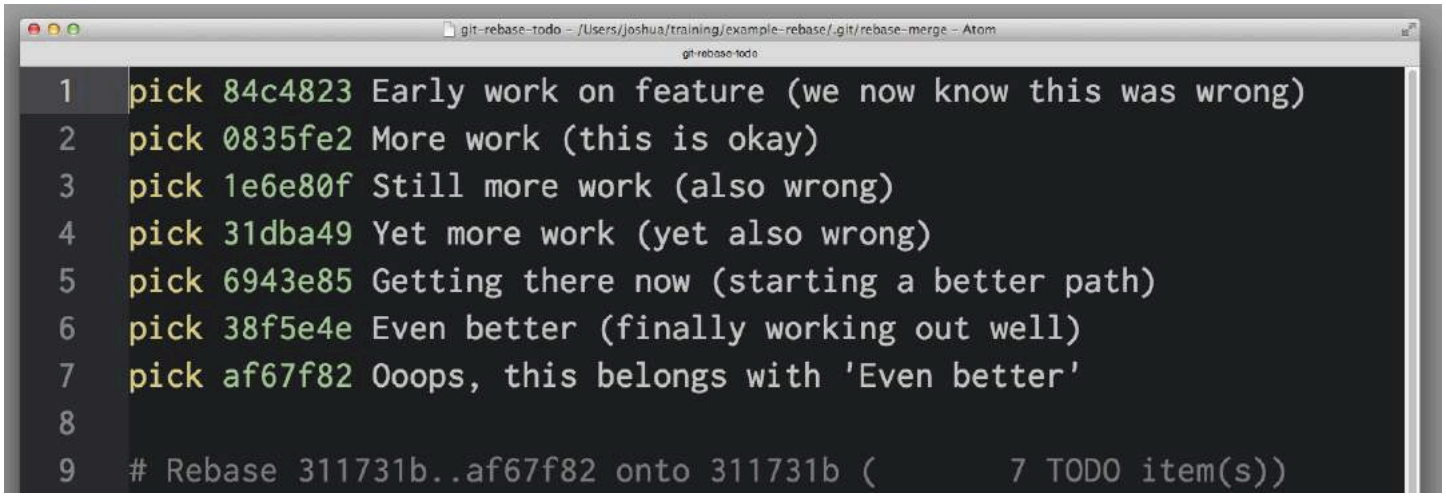
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Section 7.6: Undo / Redo a series of commits

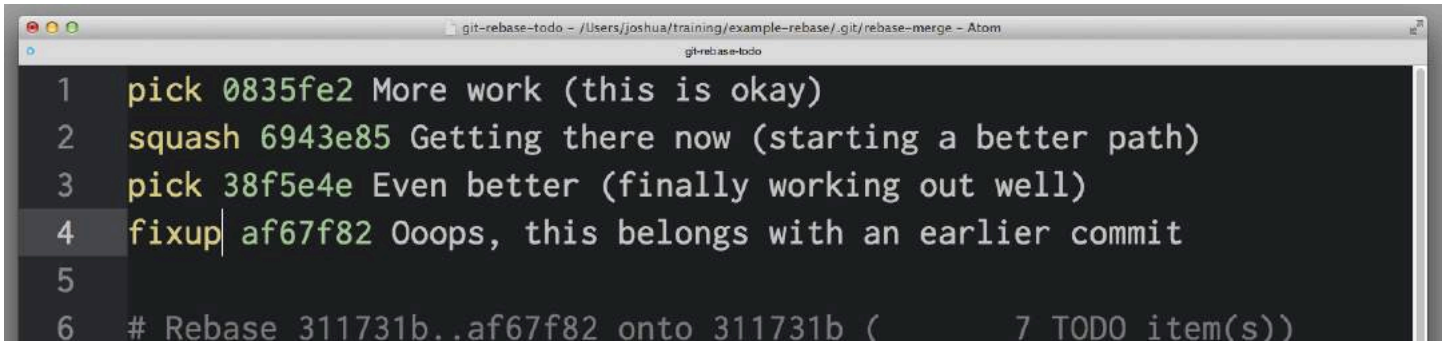
Assume you want to undo a dozen of commits and you want only some of them.

```
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



Chapter 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 from only one side of a merge

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

```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Section 8.6: Merge one branch into another

```
git merge incomingBranch
```

This merges the branch `incomingBranch` into the branch you are currently in. For example, if you are currently in `master`, then `incomingBranch` will be merged into `master`.

Merging can create conflicts in some cases. If this happens, you will see the message `Automatic merge failed; fix conflicts and then commit the result.` You will need to manually edit the conflicted files, or to undo your merge attempt, run:

```
git merge --abort
```

Chapter 9: Submodules

Section 9.1: Cloning a Git repository having submodules

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

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

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

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
```

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 (from the parent repo) a:

```
git submodule update --remote --recursive
```

Since the SHA1 of the submodule would change, you would still need to follow that with:

```
git add .
git commit -m "update submodules"
```

That supposes the submodules were:

- either added with a branch to follow:

  ```
  git submodule -b abranch -- /url/of/submodule/repo
  ```

- or configured (for an existing submodule) to follow a branch:

  ```
  cd /path/to/parent/repo
  git config -f .gitmodules submodule.asubmodule.branch abranch
  ```

Section 9.5: Moving a submodule

`Version > 1.8`

Run:

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

`Version ≤ 1.8`

1. Edit `.gitmodules` and change the path of the submodule appropriately, and put it in the index with `git add .gitmodules`.
2. If needed, create the parent directory of the new location of the submodule (`mkdir -p` /path/to).
3. Move all content from the old to the new directory (`mv -vi` /path/to/module *new/path/to/submodule*).
4. Make sure Git tracks this directory (`git add` /path/to).
5. Remove the old directory with `git rm --cached` /path/to/module.
6. Move the directory `.git/modules/`/path/to/module with all its content to `.git/modules/`/path/to/module.
7. Edit the `.git/modules/`/path/to/config file, make sure that worktree item points to the new locations, so in this example it should be `worktree = ../../../../../`/path/to/module. Typically there should be two more `../` then directories in the direct path in that place. . Edit the file /path/to/module/.git, make sure that the path

in it points to the correct new location inside the main project `.git` folder, so in this example `gitdir`: `../../../.git/modules/`/path/to/module.

`git status` output looks like this afterwards:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   .gitmodules
#       renamed:    old/path/to/submodule -> new/path/to/submodule
#
```

8. Finally, commit the changes.

This example from Stack Overflow, by Axel Beckert

Section 9.6: Removing a submodule

`Version > 1.8`

You can remove a submodule (e.g. `the_submodule`) by calling:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` deletes `the_submodules`' entry from .git/config. This excludes the_submodule from `git submodule update`, `git submodule sync` and `git submodule foreach` calls and deletes its local content (source). Also, this will not be shown as change in your parent repository. `git submodule init` and `git submodule update` will restore the submodule, again without commitable changes in your parent repository.
- `git rm the_submodule` will remove the submodule from the work tree. The files will be gone as well as the submodules' entry in the `.gitmodules` file (source). If only `git rm the_submodule` (without prior `git submodule deinit the_submodule` is run, however, the submodules' entry in your .git/config file will remain.

`Version < 1.8`

Taken from here:

1. Delete the relevant section from the `.gitmodules` file.
2. Stage the `.gitmodules` changes `git add .gitmodules`
3. Delete the relevant section from `.git/config`.
4. Run `git rm --cached path_to_submodule` (no trailing slash).
5. Run `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Delete the now untracked submodule files
8. `rm -rf path_to_submodule`

Chapter 10: Committing

Parameter	Details
--message, -m	Message to include in the commit. Specifying this parameter bypasses Git's normal behavior of opening an editor.
--amend	Specify that the changes currently staged should be added (amended) to the *previous* commit. Be careful, this can rewrite history!
--no-edit	Use the selected commit message without launching an editor. For example, `git commit --amend --no-edit` amends a commit without changing its commit message.
--all, -a	Commit all changes, including changes that aren't yet staged.
--date	Manually set the date that will be associated with the commit.
--only	Commit only the paths specified. This will not commit what you currently have staged unless told to do so.
--patch, -p	Use the interactive patch selection interface to chose which changes to commit.
--help	Displays the man page for `git commit`
-S[keyid], -S --gpg-sign[=keyid], -S --no-gpg-sign	Sign commit, GPG-sign commit, countermand `commit.gpgSign` configuration variable
-n, --no-verify	This option bypasses the pre-commit and commit-msg hooks. See also Hooks

Commits with Git provide accountability by attributing authors with changes to code. Git offers multiple features for the specificity and security of commits. This topic explains and demonstrates proper practices and procedures in committing with Git.

Section 10.1: Stage and commit changes

The basics

After making changes to your source code, you should **stage** those changes with Git before you can commit them.

For example, if you change `README.md` and `program.py`:

```
git add README.md program.py
```

This tells git that you want to add the files to the next commit you do.

Then, commit your changes with

```
git commit
```

Note that this will open a text editor, which is often vim. If you are not familiar with vim, you might want to know that you can press `i` to go into *insert* mode, write your commit message, then press `Esc` and `:wq` to save and quit. To avoid opening the text editor, simply include the `-m` flag with your message

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

Commit messages often follow some specific formatting rules, see Good commit messages for more information.

Shortcuts

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

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

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

```
git add .
```

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

```
git add -u
```

If desired, review the staged changes:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Finally, commit the changes:

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

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

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

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

Sensitive data

You should never commit any sensitive data, such as passwords or even private keys. If this case happens and the changes are already pushed to a central server, consider any sensitive data as compromised. Otherwise, it is possible to remove such data afterwards. A fast and easy solution is the usage of the "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/.

The command `bfg --replace-text passwords.txt my-repo.git` reads passwords out of the `passwords.txt` file and replaces these with `***REMOVED***`. This operation considers all previous commits of the entire repository.

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?
```

```
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 from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Manually wrap each line of the body at 72 characters
7. Use the body to explain *what* and *why* instead of *how*

7 rules from Chris Beam's blog.

Section 10.3: Amending a commit

If your **latest commit is not published yet** (not pushed to an upstream repository) 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 from the branch's history. This should be kept in mind when working with public repositories and on branches with other collaborators.

This means that if the earlier commit had already been pushed, after amending it you will have to push `--force`.

Section 10.4: Committing without opening an editor

Git will usually open an editor (like `vim` or `emacs`) when you run `git commit`. Pass the `-m` option to specify a message from the command line:

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

Your commit message can go over multiple lines:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

Alternatively, you can pass in multiple `-m` arguments:

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

See How to Write a Git Commit Message.

Udacity Git Commit Message Style Guide

Section 10.5: Committing changes directly

Usually, you have to use `git add` or `git rm` to add changes to the index before you can `git commit` them. Pass the `-a` or `--all` option to automatically add every change (to tracked files) to the index, including removals:

```
git commit -a
```

If you would like to also add a commit message you would do:

```
git commit -a -m "your commit message goes here"
```

Also, you can join two flags:

```
git commit -am "your commit message goes here"
```

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

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

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

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

Section 10.6: Selecting which lines should be staged for committing

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

```
git add -p
```

or

```
git add -p [file]
```

Each of your changes will be displayed individually, and for each change you will be prompted to choose one of he following options:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
It will open the hunk in a text editor and you can edit it as needed.
```

This will stage the parts of the files you choose. Then you can commit all the staged changes like this:

```
git commit -m 'Commit Message'
```

The changes that were not staged or committed will still appear in your working files, and can be committed later if required. Or if the remaining changes are unwanted, they can be discarded with:

```
git reset --hard
```

Apart from breaking up a big change into smaller commits, this approach is also useful for *reviewing* what you are about to commit. By individually confirming each change, you have an opportunity to check what you wrote, and can avoid accidentally staging unwanted code such as println/logging statements.

Section 10.7: Creating an empty commit

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

However, when testing build hooks, CI systems, and other systems that trigger off a commit, it's handy to be able to easily create commits without having to edit/touch a dummy file.

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

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

Section 10.8: Committing on behalf of someone else

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

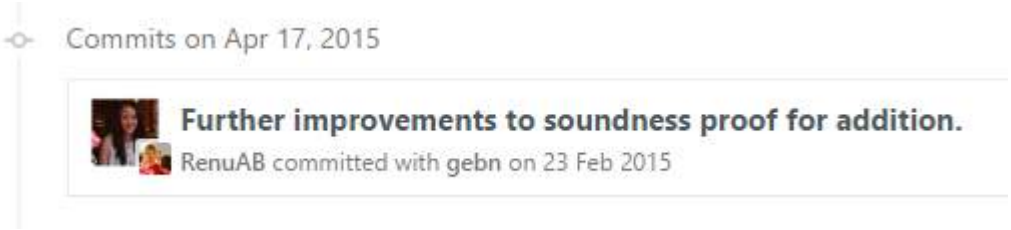
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

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

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

In this case, the author information from the most recent commit with an author containing "John" will be used.

On GitHub, commits made in either of the above ways will show a large author's thumbnail, with the committer's smaller and in front:



Section 10.9: GPG signing commits

1. Determine your key ID

   ```
   gpg --list-secret-keys --keyid-format LONG

   /Users/davidcondrey/.gnupg/secring.gpg
   ------------------------------------
   sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
   ```

   Your ID is a alphanumeric 16-digit code following the first forward-slash.

2. Define your key ID in your git config

   ```
   git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
   ```

3. As of version 1.7.9, git commit accepts the -S option to attach a signature to your commits. Using this option will prompt for your GPG passphrase and will add your signature to the commit log.

   ```
   git commit -S -m "Your commit message"
   ```

Section 10.10: Commiting changes in specific files

You can commit changes made to specific files and skip staging them using `git add`:

```
git commit file1.c file2.h
```

Or you can first stage the files:

```
git add file1.c file2.h
```

and commit them later:

```
git commit
```

Section 10.11: Committing at a specific date

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

The `--date` parameter sets the *author date*. This date will appear in the standard output of `git log`, for example.

To force the *commit date* too:

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

The date parameter accepts the flexible formats as supported by GNU date, for example:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

When the date doesn't specify time, the current time will be used and only the date will be overridden.

Section 10.12: Amending the time of a commit

You cam amend the time of a commit using

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

or even

```
git commit --amend --date="now"
```

Section 10.13: Amending the author of a commit

If you make a commit as the wrong author, you can change it, and then amend

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```