Git add filename

Git commit -m “enter message here”

Git status

Git show head

Git log

git checkout HEAD filename

will restore the file in your working directory to look exactly as it did when you last made a commit.

git add filename\_1 filename\_2

What if, before you commit, you accidentally delete an important line from **scene-2.txt**? Unthinkingly, you add **scene-2.txt** to the staging area. The file change is unrelated to the Larry/Laertes swap and you don't want to include it in the commit.

We can unstage that file from the staging area using

git reset HEAD filename

This command resets the file in the staging area to be the same as the HEAD commit. It does not discard file changes from the working directory, it just removes them from the staging area.

Git enables you to rewind to the part before you made the wrong turn and create a new destiny for the project. You can do this with:

git reset SHA

This command works by using the first 7 characters of the SHA of a previous commit. For example, if the SHA of the previous commit is 5d692065cf51a2f50ea8e7b19b5a7ae512f633ba, use:

git reset 5d69206

* git checkout HEAD filename: Discards changes in the working directory.
* git reset HEAD filename: Unstages file changes in the staging area.
* git reset SHA: Can be used to reset to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

git add filename\_1 filename\_2

git branch

git branch new\_branch

The master and fencing branches are identical: they share the same exact commit history. You can switch to the new branch with

git checkout branch\_name

What if you wanted include all the changes made to the fencing branch on the master branch? We can easily accomplish this by merging the branch into master with:

git merge branch\_name

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the masterbranch. After the branch has been integrated into master, it has served its purpose and can be deleted.

The command

git branch -d branch\_name

will delete the specified branch from your Git project.

Sally has created the remote repository, **science-quizzes** in the directory **curriculum**, which teachers on the school's shared network have access to. In order to get your own replica of **science-quizzes**, you'll need to clone it with:

git clone remote\_location clone\_name

* git clone: Creates a local copy of a remote.
* git remote -v: Lists a Git project's remotes.
* git fetch: Fetches work from the remote into the local copy.
* git merge origin/master: Merges origin/master into your local branch.
* git push origin <branch\_name>: Pushes a local branch to the origin remote.

BASIC SNAPSHOTTING

Git is all about composing and saving snapshots of your project and then working with and comparing those snapshots. This section will explain the commands needed to compose and commit snapshots of your project.

An important concept here is that Git has an 'index', which acts as sort of a staging area for your snapshot. This allows you to build up a series of well composed snapshots from changed files in your working directory, rather than having to commit all of the file changes at once.

In a nutshell, you will use git add to start tracking new files and also to stage changes to already tracked files, then git status and git diff to see what has been modified and staged and finally git commit to record your snapshot into your history. This will be the basic workflow that you use most of the time.

docs book git add adds file contents to the staging area

In Git, you have to add file contents to your staging area before you can commit them. If the file is new, you can run git add to initially add the file to your staging area, but even if the file is already "tracked" - ie, it was in your last commit - you still need to call git add to add new modifications to your staging area. Let's see a few examples of this.

Going back to our Hello World example, once we've initiated the project, we would now start adding our files to it and we would do that with git add. We can use git status to see what the state of our project is.

$ git status -s

?? README

?? hello.rb

So right now we have two untracked files. We can now add them.

$ git add README hello.rb

Now if we run git status again, we'll see that they've been added.

$ git status -s

A README

A hello.rb

It is also common to recursively add all files in a new project by specifying the current working directory like this: git add .. Since Git will recursively add all files under a directory you give it, if you give it the current working directory, it will simply start tracking every file there. In this case, a git add . would have done the same thing as a git add README hello.rb, or for that matter so would git add \*, but that's only because we don't have subdirectories which the \* would not recurse into.

OK, so now if we edit one of these files and run git status again, we will see something odd.

$ vim README

$ git status -s

AM README

A hello.rb

The 'AM' status means that the file has been modified on disk since we last added it. This means that if we commit our snapshot right now, we will be recording the version of the file when we last ran git add, not the version that is on our disk. Git does not assume that what the file looks like on disk is necessarily what you want to snapshot - you have to tell Git with the git add command.

In a nutshell, you run git add on a file when you want to include whatever changes you've made to it in your next commit snapshot. Anything you've changed that is not added will not be included - this means you can craft your snapshots with a bit more precision than most other SCM systems.

For a very interesting example of using this flexibility to stage only parts of modified files at a time, see the '-p' option to git add in the Pro Git book.

docs book git status view the status of your files in the working directory and staging area

As you saw in the git add section, in order to see what the status of your staging area is compared to the code in your working directory, you can run the git status command. Using the -s option will give you short output. Without that flag, the git status command will give you more context and hints. Here is the same status output with and without the -s. The short output looks like this:

$ git status -s

AM README

A hello.rb

Where the same status with the long output looks like this:

$ git status

# On branch master

#

# Initial commit

#

# Changes to be committed:

# (use "git rm --cached <file>..." to unstage)

#

# new file: README

# new file: hello.rb

#

# Changed but not updated:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: README

#

You can easily see how much more compact the short output is, but the long output has useful tips and hints as to what commands you may want to use next.

Git will also tell you about files that were deleted since your last commit or files that were modified or staged since your last commit.

$ git status -s

M README

D hello.rb

You can see there are two columns in the short status output. The first column is for the staging area, the second is for the working directory. So for example, if you have the README file staged and then you modify it again without running git add a second time, you'll see this:

$ git status -s

MM README

D hello.rb

In a nutshell, you run git status to see if anything has been modified and/or staged since your last commit so you can decide if you want to commit a new snapshot and what will be recorded in it.

docs book git diff shows diff of what is staged and what is modified but unstaged

There are two main uses of the git diff command. One use we will describe here, the other we will describe later in the "Inspection and Comparison" section. The way we're going to use it here is to describe the changes that are staged or modified on disk but unstaged.

git diff show diff of unstaged changes

Without any extra arguments, a simple git diff will display in unified diff format (a patch) what code or content you've changed in your project since the last commit that are not yet staged for the next commit snapshot.

$ vim hello.rb

$ git status -s

M hello.rb

$ git diff

diff --git a/hello.rb b/hello.rb

index d62ac43..8d15d50 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

So where git status will show you what files have changed and/or been staged since your last commit, git diff will show you what those changes actually are, line by line. It's generally a good follow-up command to git status

git diff --cached show diff of staged changes

The git diff --cached command will show you what contents have been staged. That is, this will show you the changes that will currently go into the next commit snapshot. So, if you were to stage the change to hello.rb in the example above, git diff by itself won't show you any output because it will only show you what is not yet staged.

$ git status -s

M hello.rb

$ git add hello.rb

$ git status -s

M hello.rb

$ git diff

$

If you want to see the staged changes, you can run git diff --cached instead.

$ git status -s

M hello.rb

$ git diff

$

$ git diff --cached

diff --git a/hello.rb b/hello.rb

index d62ac43..8d15d50 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

1481

down vote

accepted

It should just be:

git diff --cached

--cached means show the changes in the cache/index (i.e. staged changes) against the current HEAD. --staged is a synonym for --cached.

EDIT

Just to clarify the above statement, --staged and --cached does not point to HEAD, just difference with respect to HEAD. If you cherry pick what to commit using git add --patch (or git add -p), --staged will return what is staged.

git diff [--options] --cached [<commit>] [--] [<path>…​]

This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

-------------------------------------------------------------------

**git diff HEAD**show diff of all staged or unstaged changes

If you want to see both staged and unstaged changes together, you can run git diff HEAD - this basically means you want to see the difference between your working directory and the last commit, ignoring the staging area. If we make another change to our hello.rb file then we'll have some changes staged and some changes unstaged. Here are what all three diff commands will show you:

**$ vim hello.rb**

**$ git diff**

diff --git a/hello.rb b/hello.rb

index 4f40006..2ae9ba4 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

+ # says hello

def self.hello

puts "hola mundo"

end

end

**$ git diff --cached**

diff --git a/hello.rb b/hello.rb

index 2aabb6e..4f40006 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

**$ git diff HEAD**

diff --git a/hello.rb b/hello.rb

index 2aabb6e..2ae9ba4 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,8 @@

class HelloWorld

+ # says hello

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

**git diff --stat**show summary of changes instead of a full diff

If we don't want the full diff output, but we want more than the git status output, we can use the --stat option, which will give us a summary of changes instead. Here is the same example as above, but using the --stat option instead.

**$ git status -s**

MM hello.rb

**$ git diff --stat**

hello.rb | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

**$ git diff --cached --stat**

hello.rb | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

**$ git diff HEAD --stat**

hello.rb | 3 ++-

1 files changed, 2 insertions(+), 1 deletions(-)

You can also provide a file path at the end of any of these options to limit the diff output to a specific file or subdirectory.

**In a nutshell**, you run git diff to see details of the git status command - *how* files have been modified or staged on a line by line basis.

## git commit**records a snapshot of the staging area**

Now that you have staged the content you want to snapshot with the git add command, you run git commit to actually record the snapshot. Git records your name and email address with every commit you make, so the first step is to tell Git what these are.

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

**$ git config --global user.email you@somedomain.com**

Let's stage and commit all the changes to our hello.rb file. In this first example, we'll use the -m option to provide the commit message on the command line.

**$ git add hello.rb**

**$ git status -s**

M hello.rb

**$ git commit -m 'my hola mundo changes'**

[master 68aa034] my hola mundo changes

1 files changed, 2 insertions(+), 1 deletions(-)

Now we have recorded the snapshot. If we run git status again, we will see that we have a "clean working directory", which means that we have not made any changes since our last commit - there is no un-snapshotted work in our checkout.

**$ git status**

# On branch master

nothing to commit (working directory clean)

If you leave off the -m option, Git will try to open a text editor for you to write your commit message. In vim, which it will default to if it can find nothing else in your settings, the screen might look something like this:

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: hello.rb

#

~

~

".git/COMMIT\_EDITMSG" 9L, 257C

At this point you add your actual commit message at the top of the document. Any lines starting with '#' will be ignored - Git will put the output of the git status command in there for you as a reminder of what you have modified and staged.

In general, it's very important to write a good commit message. For open source projects, it's generally a rule to write your message more or less in this format:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72

characters or so. In some contexts, the first line is treated as the

subject of an email and the rest of the text as the body. The blank

line separating the summary from the body is critical (unless you omit

the body entirely); some git tools can get confused if you run the

two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a

single space, with blank lines in between, but conventions vary

here

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: hello.rb

#

~

~

~

".git/COMMIT\_EDITMSG" 25L, 884C written

The commit message is very important. Since much of the power of Git is this flexibility in carefully crafting commits locally and then sharing them later, it is very powerful to be able to write three or four commits of logically separate changes so that your work may be more easily peer reviewed. Since there is a separation between committing and pushing those changes, do take the time to make it easier for the people you are working with to see what you've done by putting each logically separate change in a separate commit with a nice commit message so it is easier for them to see what you are doing and why.

#### git commit -a automatically stage all tracked, modified files before the commit

If you think the git add stage of the workflow is too cumbersome, Git allows you to skip that part with the -a option. This basically tells Git to run git add on any file that is "tracked" - that is, any file that was in your last commit and has been modified. This allows you to do a more Subversion style workflow if you want, simply editing files and then running git commit -a when you want to snapshot everything that has been changed. You still need to run git add to start tracking new files, though, just like Subversion.

**$ vim hello.rb**

**$ git status -s**

M hello.rb

**$ git commit -m 'changes to hello file'**

# On branch master

# Changed but not updated:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: hello.rb

#

no changes added to commit (use "git add" and/or "git commit -a")

**$ git commit -am 'changes to hello file'**

[master 78b2670] changes to hello file

1 files changed, 2 insertions(+), 1 deletions(-)

Notice how if you don't stage any changes and then run git commit, Git will simply give you the output of the git status command, reminding you that nothing is staged. The important part of that message has been highlighted, saying that nothing is added to be committed. If you use -a, it will add and commit everything at once.

This now lets you complete the entire snapshotting workflow - you make changes to your files, then use git add to stage files you want to change, git status and git diff to see what you've changed, and then finally git commit to actually record the snapshot forever.

**In a nutshell**, you run git commit to record the snapshot of your staged content. This snapshot can then be compared, shared and reverted to if you need to.

**git reset HEAD**unstage files from index and reset pointer to HEAD

First, you can use it to unstage something that has been accidentally staged. Let's say that you have modified two files and want to record them into two different commits. You should stage and commit one, then stage and commit the other. If you accidentally stage both of them, how do you *un-*stage one? You do it with git reset HEAD -- file. Technically you don't have to add the -- - it is used to tell Git when you have stopped listing options and are now listing file paths, but it's probably good to get into the habit of using it to separate options from paths even if you don't need to.

Let's see what it looks like to unstage something. Here we have two files that have been modified since our last commit. We will stage both, then unstage one of them.

**$ git status -s**

M README

M hello.rb

**$ git add .**

**$ git status -s**

M README

M hello.rb

**$ git reset HEAD -- hello.rb**

Unstaged changes after reset:

M hello.rb

**$ git status -s**

M README

M hello.rb

Now you can run a git commit which will just record the changes to the README file, not the now unstaged hello.rb.

In case you're curious, what it's actually doing here is it is resetting the checksum of the entry for that file in the "index" to be what it was in the last commit. Since git addchecksums a file and adds it to the "index", git reset HEAD overwrites that with what it was before, thereby effectively unstaging it.

If you want to be able to just run git unstage, you can easily setup an alias in Git. Just run git config --global alias.unstage "reset HEAD". Once you have run that, you can then just run git unstage [file] instead.

If you forget the command to unstage something, Git is helpful in reminding you in the output of the normal git status command. For example, if you run git statuswithout the -s when you have staged files, it will tell you how to unstage them:

**$ git status**

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README

# modified: hello.rb

#

When you run git reset without specifying a flag it defaults to --mixed. The other options are --soft and --hard.

**git reset --soft**moves HEAD to specified commit reference, index and staging are untouched

The first thing git reset does is undo the last commit and put the files back onto the stage. If you include the --soft flag this is where it stops. For example, if you run git reset --soft HEAD~ (the parent of the HEAD) the last commit will be undone and the files touched will be back on the stage again.

**$ git status -s**

M hello.rb

**$ git commit -am 'hello with a flower'**

[master 5857ac1] hello with a flower

1 files changed, 3 insertions(+), 1 deletions(-)

**$ git status**

# On branch master

nothing to commit (working directory clean)

**$ git reset --soft HEAD~**

**$ git status -s**

M hello.rb

This is basically doing the same thing as git commit --amend, allowing you to do more work before you roll in the file changes into the same commit.

**git reset --hard**unstage files AND undo any changes in the working directory since last commit

The third option is to go --hard. This command discards your staged changes and the changes in your working directory. In other words: it resets your staging area and working directory to the state they were in at the given commit. This is the most dangerous option and is not working directory safe. Any changes not committed will be lost.

**$ git status**

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: README

#

**$ git reset --hard HEAD**

HEAD is now at 5857ac1 hello with a flower

**$ git status**

# On branch master

nothing to commit (working directory clean)

In the above example, while we had both changes ready to commit and ready to stage, a git reset --hard wiped them out. The working tree and staging area are reset to the tip of the current branch or HEAD.

You can replace HEAD with a commit SHA-1 or another parent reference to reset to that specific point.

**In a nutshell**, you run git reset HEAD to undo the last commit, unstage files that you previously ran git add on and wish to not include in the next commit snapshot

## git rm**remove files from the staging area**

git rm will remove entries from the staging area. This is a bit different from git reset HEAD which "unstages" files. To "unstage" means it reverts the staging area to what was there before we started modifying things. git rm on the other hand just kicks the file off the stage entirely, so that it's not included in the next commit snapshot, thereby effectively deleting it.

By default, a git rm file will remove the file from the staging area entirely and also off your disk (the working directory). To leave the file in the working directory, you can use git rm --cached .

#### git mv git rm --cached orig; mv orig new; git add new

Unlike most other version control systems, Git does not track file renames. Instead, it just tracks the snapshots and then figures out what files were likely renamed by comparing snapshots. If a file was removed from one snapshot and another file was added to the next one and the contents are similar, Git figures it was most likely a rename. So, although the git mv command exists, it is superfluous - all it does is a git rm --cached, moves the file on disk, then runs a git add on the new file. You don't really need to use it, but if it's easier, feel free.

In its normal form the command is used to delete files. But it's often easier to just remove the files off your disk and then run git commit -a, which will also automatically remove them from your index.

**In a nutshell**, you run git rm to remove files from being tracked in Git. It will also remove them from your working directory.

## **BRANCHING AND MERGING**

Branching in Git is one of its many great features. If you have used other version control systems, it's probably helpful to forget most of what you think about branches - in fact, it may be more helpful to think of them practically as *contexts* since that is how you will most often be using them. When you checkout different branches, you change contexts that you are working in and you can quickly context-switch back and forth between several different branches.

**In a nutshell** you can create a branch with git branch (branchname), switch into that context with git checkout (branchname), record commit snapshots while in that context, then can switch back and forth easily. When you switch branches, Git replaces your working directory with the snapshot of the latest commit on that branch so you don't have to have multiple directories for multiple branches. You merge branches together with git merge. You can easily merge multiple times from the same branch over time, or alternately you can choose to delete a branch immediately after merging it.

## [docs](http://git-scm.com/docs/git-branch)[book](http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is)git branch**list, create and manage working contexts**

## [docs](http://git-scm.com/docs/git-checkout)[book](http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging)git checkout**switch to a new branch context**

The git branch command is a general branch management tool for Git and can do several different things. We'll cover the basic ones that you'll use most - listing branches, creating branches and deleting branches. We will also cover basic git checkout here which switches you between your branches.

#### git branch list your available branches

Without arguments, git branch will list out the local branches that you have. The branch that you are currently working on will have a star next to it and if you have[**coloring turned on**](http://git-scm.com/book/en/Customizing-Git-Git-Configuration#Colors-in-Git), will show the current branch in green.

$ git branch

\* master

This means that we have a 'master' branch and we are currently on it. When you run git init it will automatically create a 'master' branch for you by default, however there is nothing special about the name - you don't actually have to have a 'master' branch but since it's the default that is created, most projects do.

#### git branch (branchname) create a new branch

So let's start by creating a new branch and switching to it. You can do that by running git branch (branchname).

$ git branch testing

$ git branch

\* master

testing

Now we can see that we have a new branch. When you create a branch this way it creates the branch at your last commit so if you record some commits at this point and then switch to 'testing', it will revert your working directory context back to when you created the branch in the first place - you can think of it like a bookmark for where you currently are. Let's see this in action - we use git checkout (branch) to switch the branch we're currently on.

**$ ls**

README hello.rb

**$ echo 'test content' > test.txt**

**$ echo 'more content' > more.txt**

**$ git add \*.txt**

**$ git commit -m 'added two files'**

[master 8bd6d8b] added two files

2 files changed, 2 insertions(+), 0 deletions(-)

create mode 100644 more.txt

create mode 100644 test.txt

**$ ls**

README hello.rb more.txt test.txt

**$ git checkout testing**

Switched to branch 'testing'

**$ ls**

README hello.rb

So now we can see that when we switch to the 'testing' branch, our new files were removed. We could switch back to the 'master' branch and see them re-appear.

**$ ls**

README hello.rb

**$ git checkout master**

Switched to branch 'master'

**$ ls**

README hello.rb more.txt test.txt

#### git branch -v see the last commit on each branch

If we want to see last commits on each branch we can run git branch -v to see them.

**$ git branch -v**

\* master 54b417d fix javascript issue

development 74c111d modify component.json file

testing 62a557a update test scripts

#### git checkout -b (branchname) create and immediately switch to a branch

In most cases you will be wanting to switch to the branch immediately, so you can do work in it and then merging into a branch that only contains stable work (such as 'master') at a later point when the work in your new context branch is stable. You can do this pretty easily with git branch newbranch; git checkout newbranch, but Git gives you a shortcut for this: git checkout -b newbranch.

**$ git branch**

\* master

**$ ls**

README hello.rb more.txt test.txt

**$ git checkout -b removals**

Switched to a new branch 'removals'

**$ git rm more.txt**

rm 'more.txt'

**$ git rm test.txt**

rm 'test.txt'

**$ ls**

README hello.rb

**$ git commit -am 'removed useless files'**

[removals 8f7c949] removed useless files

2 files changed, 0 insertions(+), 2 deletions(-)

delete mode 100644 more.txt

delete mode 100644 test.txt

**$ git checkout master**

Switched to branch 'master'

**$ ls**

README hello.rb more.txt test.txt

You can see there how we created a branch, removed some of our files while in the context of that branch, then switched back to our main branch and we see the files return. Branching safely isolates work that we do into contexts we can switch between.

If you start on work it is very useful to always start it in a branch (because it's fast and easy to do) and then merge it in and delete the branch when you're done. That way if what you're working on doesn't work out you can easily discard it and if you're forced to switch back to a more stable context your work in progress is easy to put aside and then come back to.

#### git branch -d (branchname) delete a branch

If we want to delete a branch (such as the 'testing' branch in the previous example, since there is no unique work on it), we can run git branch -d (branch) to remove it.

**$ git branch**

\* master

testing

**$ git branch -d testing**

Deleted branch testing (was 78b2670).

**$ git branch**

\* master

#### git push (remote-name) :(branchname) delete a remote branch

When you're done with a remote branch, whether it's been merged into the remote master or you want to abandon it and sweep it under the rug, you'll issue a git pushcommand with a specially placed colon symbol to remove that branch.

**$ git push origin :tidy-cutlery**

To git@github.com:octocat/Spoon-Knife.git

- [deleted] tidy-cutlery

In the above example you've deleted the "tidy-cutlery" branch of the "origin" remote. A way to remember this is to think of the git push remote-name local-branch:remote-branch syntax. This states that you want to push your local branch to match that of the remote. When you remove the local-branch portion you're now matching nothing to the remote, effectively telling the remote branch to become nothing.

Alternatively, you can run git push remote-name --delete branchname which is a wrapper for the colon refspec (a source:destination pair) of deleting a remote branch.

**In a nutshell** you use git branch to list your current branches, create new branches and delete unnecessary or already merged branches.

## [docs](http://git-scm.com/docs/git-merge)[book](http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging#Basic-Merging)git merge**merge a branch context into your current one**

Once you have work isolated in a branch, you will eventually want to incorporate it into your main branch. You can merge any branch into your current branch with the git merge command. Let's take as a simple example the 'removals' branch from above. If we create a branch and remove files in it and commit our removals to that branch, it is isolated from our main ('master', in this case) branch. To include those deletions in your 'master' branch, you can just merge in the 'removals' branch.

**$ git branch**

\* master

removals

**$ ls**

README hello.rb more.txt test.txt

**$ git merge removals**

Updating 8bd6d8b..8f7c949

Fast-forward

more.txt | 1 -

test.txt | 1 -

2 files changed, 0 insertions(+), 2 deletions(-)

delete mode 100644 more.txt

delete mode 100644 test.txt

**$ ls**

README hello.rb

#### more complex merges

Of course, this doesn't just work for simple file additions and deletions. Git will merge file modifications as well - in fact, it's very good at it. For example, let's see what happens when we edit a file in one branch and in another branch we rename it and then edit it and then merge these branches together. Chaos, you say? Let's see.

**$ git branch**

\* master

**$ cat hello.rb**

class HelloWorld

def self.hello

puts "Hello World"

end

end

HelloWorld.hello

So first we're going to create a new branch named 'change\_class' and switch to it so your class renaming changes are isolated. We're going to change each instance of 'HelloWorld' to 'HiWorld'.

**$ git checkout -b change\_class**

Switched to a new branch 'change\_class'

**$ vim hello.rb**

**$ head -1 hello.rb**

class HiWorld

**$ git commit -am 'changed the class name'**

[change\_class 3467b0a] changed the class name

1 files changed, 2 insertions(+), 4 deletions(-)

So now we've committed the class renaming changes to the 'change\_class' branch. To switch back to the 'master' branch the class name will revert to what it was before we switched branches. Here we can change something different (in this case the printed output) and at the same time rename the file from hello.rb to ruby.rb.

**$ git checkout master**

Switched to branch 'master'

**$ git mv hello.rb ruby.rb**

**$ vim ruby.rb**

**$ git diff**

diff --git a/ruby.rb b/ruby.rb

index 2aabb6e..bf64b17 100644

--- a/ruby.rb

+++ b/ruby.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "Hello World"

+ puts "Hello World from Ruby"

end

end

**$ git commit -am 'added from ruby'**

[master b7ae93b] added from ruby

1 files changed, 1 insertions(+), 1 deletions(-)

rename hello.rb => ruby.rb (65%)

Now those changes are recorded in the 'master' branch. Notice that the class name is back to 'HelloWorld', not 'HiWorld'. To incorporate the 'HiWorld' change we can just merge in the 'change\_class' branch. However, the name of the file has changed since we branched, what will Git do?

**$ git branch**

change\_class

\* master

**$ git merge change\_class**

Renaming hello.rb => ruby.rb

Auto-merging ruby.rb

Merge made by recursive.

ruby.rb | 6 ++----

1 files changed, 2 insertions(+), 4 deletions(-)

**$ cat ruby.rb**

class HiWorld

def self.hello

puts "Hello World from Ruby"

end

end

HiWorld.hello

Well, it will just figure it out. Notice that there are no merge conflicts and the file that had been renamed now has the 'HiWorld' class name change that was done in the other branch. Pretty cool.

#### merge conflicts

So, Git merges are magical, we never ever have to deal with merge conflicts again, right? Not quite. In situations where the same block of code is edited in different branches there is no way for a computer to figure it out, so it's up to us. Let's see another example of changing the same line in two branches.

**$ git branch**

\* master

**$ git checkout -b fix\_readme**

Switched to a new branch 'fix\_readme'

**$ vim README**

**$ git commit -am 'fixed readme title'**

[fix\_readme 3ac015d] fixed readme title

1 files changed, 1 insertions(+), 1 deletions(-)

Now we have committed a change to one line in our README file in a branch. Now let's change the same line in a different way back on our 'master' branch.

**$ git checkout master**

Switched to branch 'master'

**$ vim README**

**$ git commit -am 'fixed readme title differently'**

[master 3cbb6aa] fixed readme title differently

1 files changed, 1 insertions(+), 1 deletions(-)

Now is the fun part - we will merge the first branch into our master branch, causing a merge conflict.

**$ git merge fix\_readme**

Auto-merging README

CONFLICT (content): Merge conflict in README

Automatic merge failed; fix conflicts and then commit the result.

**$ cat README**

<<<<<<< HEAD

Many Hello World Examples

=======

Hello World Lang Examples

>>>>>>> fix\_readme

This project has examples of hello world in

nearly every programming language.

You can see that Git inserts standard merge conflict markers, much like Subversion, into files when it gets a merge conflict. Now it's up to us to resolve them. We will do it manually here, but check out [**git mergetool**](http://git-scm.com/docs/git-mergetool) if you want Git to fire up a graphical mergetool (like kdiff3, emerge, p4merge, etc) instead.

**$ vim README**  # here I'm fixing the conflict

**$ git diff**

diff --cc README

index 9103e27,69cad1a..0000000

--- a/README

+++ b/README

@@@ -1,4 -1,4 +1,4 @@@

- Many Hello World Examples

-Hello World Lang Examples

++Many Hello World Lang Examples

This project has examples of hello world in

A cool tip in doing merge conflict resolution in Git is that if you run git diff, it will show you both sides of the conflict and how you've resolved it as shown here. Now it's time to mark the file as resolved. In Git we do that with git add - to tell Git the file has been resolved you have to stage it.

**$ git status -s**

UU README

**$ git add README**

**$ git status -s**

M README

**$ git commit**

[master 8d585ea] Merge branch 'fix\_readme'

And now we've successfully resolved our merge conflict and committed the result.

**In a nutshell** you use git merge to combine another branch context into your current branch. It automatically figures out how to best combine the different snapshots into a new snapshot with the unique work of both.

## [docs](http://git-scm.com/docs/git-log)[book](http://git-scm.com/book/en/Git-Tools-Revision-Selection#Commit-Ranges)git log**show commit history of a branch**

So far we have been committing snapshots of your project and switching between different isolated contexts, but what if we've forgotten how we've got to where we are? Or what if we want to know how one branch differs from another? Git provides a tool that shows you all the commit messages that have lead up to the snapshot you are currently on, which is called git log.

To understand the log command, you have to understand what information is stored when you run the git commit command to store a snapshot. In addition to the manifest of files and commit message and information about the person who committed it, Git also stores the commit that you based this snapshot on. That is, if you clone a project, what was the snapshot that you modified to get to the snapshot that you saved? This is helpful to give context to how the project got to where it is and allows Git to figure out who changed what. If Git has the snapshot you save and the one you based it on, then it can automatically figure out what you changed. The commit that a new commit was based on is called the "parent".

To see a chronological list of the parents of any branch, you can run git log when you are in that branch. For example, if we run git log in the Hello World project that we have been working on in this section, we'll see all the commit messages that we've done.

**$ git log**

commit 8d585ea6faf99facd39b55d6f6a3b3f481ad0d3d

Merge: 3cbb6aa 3ac015d

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jun 4 12:59:47 2010 +0200

Merge branch 'fix\_readme'

Conflicts:

README

commit 3cbb6aae5c0cbd711c098e113ae436801371c95e

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jun 4 12:58:53 2010 +0200

fixed readme title differently

commit 3ac015da8ade34d4c7ebeffa2053fcac33fb495b

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jun 4 12:58:36 2010 +0200

fixed readme title

commit 558151a95567ba4181bab5746bc8f34bd87143d6

Merge: b7ae93b 3467b0a

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jun 4 12:37:05 2010 +0200

Merge branch 'change\_class'

...

To see a more compact version of the same history, we can use the --oneline option.

**$ git log --oneline**

8d585ea Merge branch 'fix\_readme'

3cbb6aa fixed readme title differently

3ac015d fixed readme title

558151a Merge branch 'change\_class'

b7ae93b added from ruby

3467b0a changed the class name

17f4acf first commit

What this is telling us is that this is the history of the development of this project. If the commit messages are descriptive, this can inform us as to what all changes have been applied or have influenced the current state of the snapshot and thus what is in it.

We can also use it to see when the history was branched and merged with the very helpful --graph option. Here is the same command but with the topology graph turned on:

**$ git log --oneline --graph**

\* 8d585ea Merge branch 'fix\_readme'

|\

| \* 3ac015d fixed readme title

\* | 3cbb6aa fixed readme title differently

|/

\* 558151a Merge branch 'change\_class'

|\

| \* 3467b0a changed the class name

\* | b7ae93b added from ruby

|/

\* 17f4acf first commit

Now we can more clearly see when effort diverged and then was merged back together. This is very nice for seeing what has happened or what changes are applied, but it is also incredibly useful for managing your branches. Let's create a new branch, do some work in it and then switch back and do some work in our master branch, then see how the log command can help us figure out what is happening on each.

First we'll create a new branch to add the Erlang programming language Hello World example - we want to do this in a branch so that we don't muddy up our stable branch with code that may not work for a while so we can cleanly switch in and out of it.

**$ git checkout -b erlang**

Switched to a new branch 'erlang'

**$ vim erlang\_hw.erl**

**$ git add erlang\_hw.erl**

**$ git commit -m 'added erlang'**

[erlang ab5ab4c] added erlang

1 files changed, 5 insertions(+), 0 deletions(-)

create mode 100644 erlang\_hw.erl

Since we're having fun playing in functional programming languages we get caught up in it and also add a Haskell example program while still in the branch named 'erlang'.

**$ vim haskell.hs**

**$ git add haskell.hs**

**$ git commit -m 'added haskell'**

[erlang 1834130] added haskell

1 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 haskell.hs

Finally, we decide that we want to change the class name of our Ruby program back to the way it was. So, we can go back to the master branch and change that and we decide to just commit it directly in the master branch instead of creating another branch.

**$ git checkout master**

Switched to branch 'master'

**$ ls**

README ruby.rb

**$ vim ruby.rb**

**$ git commit -am 'reverted to old class name'**

[master 594f90b] reverted to old class name

1 files changed, 2 insertions(+), 2 deletions(-)

So, now say we don't work on the project for a while, we have other things to do. When we come back we want to know what the 'erlang' branch is all about and where we've left off on the master branch. Just by looking at the branch name, we can't know that we made Haskell changes in there, but using git log we easily can. If you give Git a branch name, it will show you just the commits that are "reachable" in the history of that branch, that is the commits that influenced the final snapshot.

**$ git log --oneline erlang**

1834130 added haskell

ab5ab4c added erlang

8d585ea Merge branch 'fix\_readme'

3cbb6aa fixed readme title differently

3ac015d fixed readme title

558151a Merge branch 'change\_class'

b7ae93b added from ruby

3467b0a changed the class name

17f4acf first commit

This way, it's pretty easy to see that we have Haskell code included in the branch (highlighted in the output). What is even cooler is that we can easily tell Git that we only are interested in the commits that are reachable in one branch that are not reachable in another, in other words which commits are unique to a branch in comparison to another.

In this case if we are interested in merging in the 'erlang' branch we want to see what commits are going to effect our snapshot when we do that merge. The way we tell Git that is by putting a ^ in front of the branch that we don't want to see. For instance, if we want to see the commits that are in the 'erlang' branch that are not in the 'master' branch, we can do erlang ^master, or vice versa. Note that the Windows command-line treats ^ as a special character, in which case you'll need to surround ^masterin quotes.

**$ git log --oneline erlang ^master**

1834130 added haskell

ab5ab4c added erlang

**$ git log --oneline master ^erlang**

594f90b reverted to old class name

This gives us a nice, simple branch management tool. It allows us to easily see what commits are unique to which branches so we know what we're missing and what we would be merging in if we were to do a merge.

**In a nutshell** you use git log to list out the commit history or list of changes people have made that have lead to the snapshot at the tip of the branch. This allows you to see how the project in that context got to the state that it is currently in.

## [docs](http://git-scm.com/docs/git-tag)[book](http://git-scm.com/book/en/Git-Basics-Tagging)git tag**tag a point in history as important**

If you get to a point that is important and you want to forever remember that specific commit snapshot, you can tag it with git tag. The tag command will basically put a permanent bookmark at a specific commit so you can use it to compare to other commits in the future. This is often done when you cut a release or ship something.

Let's say we want to release our Hello World project as version "1.0". We can tag the last commit (HEAD) as "v1.0" by running git tag -a v1.0. The -a means "make an annotated tag", which allows you to add a tag message to it, which is what you almost always want to do. Running this without the -a works too, but it doesn't record when it was tagged, who tagged it, or let you add a tag message. It's recommended you always create annotated tags.

**$ git tag -a v1.0**

When you run the git tag -a command, Git will open your editor and have you write a tag message, just like you would write a commit message.

Now, notice when we run git log --decorate, we can see our tag there.

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

\* 594f90b (HEAD, tag: v1.0, master) reverted to old class name

\* 8d585ea Merge branch 'fix\_readme'

|\

| \* 3ac015d (fix\_readme) fixed readme title

\* | 3cbb6aa fixed readme title differently

|/

\* 558151a Merge branch 'change\_class'

|\

| \* 3467b0a changed the class name

\* | b7ae93b added from ruby

|/

\* 17f4acf first commit

If we do more commits, the tag will stay right at that commit, so we have that specific snapshot tagged forever and can always compare future snapshots to it.

We don't have to tag the commit that we're on, however. If we forgot to tag a commit that we released, we can retroactively tag it by running the same command, but with the commit SHA at the end. For example, say we had released commit 558151a (several commits back) but forgot to tag it at the time. We can just tag it now:

**$ git tag -a v0.9 558151a**

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

\* 594f90b (HEAD, tag: v1.0, master) reverted to old class name

\* 8d585ea Merge branch 'fix\_readme'

|\

| \* 3ac015d (fix\_readme) fixed readme title

\* | 3cbb6aa fixed readme title differently

|/

\* 558151a (tag: v0.9) Merge branch 'change\_class'

|\

| \* 3467b0a changed the class name

\* | b7ae93b added from ruby

|/

\* 17f4acf first commit

Tags pointing to objects tracked from branch heads will be automatically downloaded when you fetch from a remote repository. However, tags that aren't reachable from branch heads will be skipped. If you want to make sure all tags are always included, you must include the --tags option.

**$ git fetch origin --tags**

remote: Counting objects: 1832, done.

remote: Compressing objects: 100% (726/726), done.

remote: Total 1519 (delta 1000), reused 1202 (delta 764)

Receiving objects: 100% (1519/1519), 1.30 MiB | 1.21 MiB/s, done.

Resolving deltas: 100% (1000/1000), completed with 182 local objects.

From git://github.com:example-user/example-repo

\* [new tag] v1.0 -> v1.0

\* [new tag] v1.1 -> v1.1

If you just want a single tag, use git fetch <remote> tag <tag-name>.

By default, tags are not included when you push to a remote repository. In order to explicitly update these you must include the --tags option when using git push.

**In a nutshell** you use git tag to mark a commit or point in your repo as important. This also allows you to refer to that commit with a more memorable reference than a SHA.

<https://git-scm.com/book/en/v1/Git-Tools-Stashing> - STASHING