A git repository manages data that gets modified. The basic process of using the repository is:

- set up the repository
- add content (from local operations, or copy it from some other repository
- edit the content, then commit those changes
- repeat the last step, as often as necessary

However, this process describes an isolated repository on a single machine, probably (though not necessarily) used by just 1 user. This has its uses, but not for software development that involves several people.

Somehow there has to be a way to share the changes that are being made with all developers.

Traditional revision control systems did this with a centralized system. The committing process either directly or indirectly ended up making changes on a server somewhere. Each developer would then somehow, at some time, fetch the changes from the server to their local machine.

There is no such thing as the "central" or "server" repository with git.

There is, however, the ability to push (send) and pull (fetch) changes from any other git repository into the one you are working with. Amazingly, the repositories do not even need to be related in any way, although there would be little point in doing that. You just need to have the right kind of access to them.

So, working with other people involves two extra steps in addition to the ones shown above:

- push (send) changes made in your repository to somewhere that other people can find/use them
- pull (retrieve) changes made by other people into your repository so that you can use them.

There is no requirement that a "central" or "server" repository be used for this. If two developers made up the whole team, they could just push and pull between each other's repositories. However, as a development team gets larger, pushing and pulling to all N different repositories (e.g. 1 per developer) gets time consuming and error-prone.

So, development teams identify one (often remote) repository as the place that they will all push their changes, and pull changes from. All developers have read access, and many will have write access. In practice, this looks like a "central" or "server" machine, because nobody else will typically be able to see/use your changes until you have pushed them to the agreed-upon shared repository. But with git, this is merely a working convention, an agreed upon practice – it is not a builtin part of how the system works.

Basic operations

1. initializing a repository by copying it from somewhere else:

```
  git clone <repo-address>
```

This will create a full, ready to use copy of the repository in your current working directory.

2. You do not need to do anything before starting to edit. There is no “checkout” or “prepare for edit” step. Just edit.

3. Once edits are done to your satisfaction, commit the change(s). Any of the following forms will work.

`git commit -m“A message about the change” -a` <= ALL changes to ALL files in the repository
`git commit path/to/a/particular/file` <= just changes to one file. An editor will start for the commit message
`git commit path/to/a/folder` <= just changes in a particular folder. An editor will start for the commit message
`git commit -m “A message about the change” file1 file2 file3` <= changes to 3 files

There is a convention for the commit message that it is very helpful to follow: if the change can be summarized entirely on 1 line in less than 60 characters, then just do that. If it cannot, find a 1 line 60 character or less summary, then leave a blank line, then write as much as you think is necessary to explain the change, taking as many lines as you wish but generally keeping them under 60 characters.

4. Letting other developers access your changes:

```
  git push
```

5. Getting changes from other developers:

```
  git pull —rebase
```

It is important to always use the –rebase command. It is possible to tell git to do this automatically.

That's it for “common” workflow. Oh wait. You committed something you did not want to commit:

6. Reverting a commit

Do not plan on ever being able to hide a commit once you've made it. However, you can undo it very easily:

```
  git revert <commit-ID>
```

Looking At Stuff

1. see the commit history

```
git log
```

2. see the short version of the commit history

```
git log –one-line
```

3. see the changes involved in a particular commit

```
git show <commit-ID>
```

4. see the commit history for a particular file (or files)

```
git log path/to/file1 [ path/to/file2 … ]
```

Working Style

Git makes it very easy to commit small changes, partly because nobody else needs to see what you're doing until you push. So it is common and wise to "commit early, commit often". The smaller the changes, the easier it is for other developers to review them and understand what you were doing. It is not uncommon to see commits involving a single line, or a commit that just changes a variable name. The basic rule of thumb is: if a particular change can be explained on its own, without any real need to refer to any other changes, then it should probably show up in its own commit.

It is even possible to use `git add -p` to commits just parts of the changes you've made to a file. That can sometimes be useful if you do a bunch of work in a file but then realize that the changes will be easier for others to understand as separate commits. This is a fairly advanced technique, however.

Git commit specifications

Each commit made to a git repository receives a unique identifier. The identifier is quite long, and can be inconvenient to use. However, you rarely need to use the whole thing. You can any shorter version of it that is still unique. So if the full commit ID is 2a0fc425f881b7cdfeade7997f93aa2d2bd6c531, you will almost be able to refer it by a short form such as 2a0fc4.

Behind the scenes

If you start reading up about how git works internally, things can get a little confusing. Unlike most other revision control systems, git does NOT store things as a series of diffs. Instead, it stores the state of each "object" included in a commit, at the time the commit is made.

You may wonder why a command like git show 2a0fc4 shows a difference, if 2a0fc4 refers to the state of specific objects at the time of the commit. This is entirely convenience for you, the programmer. Rather than show you the state of each object, it shows you the difference between the state of the object(s) in that commit, and the commit that is its parent (think "predecessor"). There are other git commands that will actually show you object state itself, but it is much less common for you to want to see this, so in general, git tends to present commits to the user as “diffs” rather than the actual state that is stored.

Branching

Sometimes you need to do work that has some combination of these characteristics:

- may involve many deep changes
- you're unsure if the planned changes or features will work
- you want to be able to work unaffected by changes other developers are doing
- you're not sure how quickly you will be able to finish the work

When this happens, the right thing to do is to create a “branch”:

```
git checkout -b <branchname>
```

After this command, you are now working in a branch – your changes apply only to that branch, and even if you push them to another repository, they will not affect anybody else (unless they are also involved in the development of that branch).

At any time you can switch back to the normal branch, which is conventionally called “master”:

```
git checkout master
```

and then go back to your branches

```
git checkout <branchname>
```

as often as you want. Git will warn you if you have uncommitted changes in the branch you are switching away from.

Merging and Rebasing

Rebasing is an odd word, invented within the git community. Its meaning can be very hard to understand if you read the technical descriptions of how it works. But it is an important concept to grasp, and is closely related to merging, which is also an important topic as soon as you start using branches.

If you have completed work on a branch, and want to get those changes applied to the normal “master” branch, you need to “merge” branches. But this can happen in two ways that differ substantively in the ordering of changes that is eventually visible.

First, lets consider a really simple, trivial case. You created branch, made one change (commit 10a39de) and are ready to merge it back into master, which has seen no changes at all since you created your branch. Obviously there is only thing to be done: apply the changes related to commit 10a39de to the master branch, and commit in that branch. That commit will be the most recent one in the master branch, often known as “HEAD”. This is very simple to do:

... finish working in your branch ...

```
git checkout master
git merge <branchname>
```

But … this is a rather uncommon scenario. It means that nobody else has made any changes to master while you were working on your branch. It is much more likely that other things have happened in master (first in developers' individual repositories, then pushed to the shared repository). This means that we now have at least two possible ways to bring together the changes you've made in your branch and those made by other people:

- combine the changes in the time order they were made
- combine the changes with yours first, then everybody else's
- combine the changes with everybody else's first, then yours

The first of these is hard to do, and essentially meaningless (the whole point of you working in a branch was to isolate you from what other people were doing, so time ordering is meaningless).

The second of these is conceptually what git merge will do. Importantly, though, it will also keep the commits you made in your branch as part of master.

The third of these is conceptually what git rebase will do. Importantly, though, it will discard the commit IDs from your branch, and try to reapply the same changes that you made.

Let's just look at that in a little more detail. Let's say that you created a branch starting from a commit ID of 111111 (an unlikely ID, but it will help to make things clearer). You make two changes in your branch, with commit ID's 211111 and 311111. Meanwhile, somebody else works in master, and makes two changes there, with commit ID's 111112 and 111113. Now we want to bring everything together. What order will we end up with?

For a merge, we will get a commit history showing

```
111111
211111
311111
111112
111113
```

For a rebase, we get something very different. Git will discard the commit IDs from your branch, and reapply the same changes to the state of things in current master. So our commit history will look like:

```
111111
111112
111113
111114  <= new commit ID containing the same changes you made in your branch as 2111111
111115  <= new commit ID containing the same changes you made in your branch as 3111111
```

When reviewing the history in the merge case, we will clearly see the fact that a branch was created, and then merged back into master. In the rebase case, we see one continuous history, as if all the changes were made in master, with no references to branches. Some projects prefer the results of merging, some prefer the results of rebasing.

The right way to merge with rebase

Assuming that your branch is ready to merge back into master, the steps to take are easy:

1. `git checkout master` *switch back to master branch*
2. `git pull —rebase` *make sure your version of master is up to date*
3. `git checkout <branchname>` *back to your branch*
4. `git rebase master` *see below*
5. `git checkout master` *back to master*
6. `git merge <branchname>` *merge your branch with master*

You will see that essentially rebase is just an extra step in the process. If we missed it out, we would still merge the branches, but the commit history would expose the presence of the branch and the merge. With the rebase step, we get a single, linear history without branches.

Step 4 is the key one, and what it does is simple. Git finds the point in the commit history of the master branch where you created your branch. It removes all the changes you've made to your branch since then. It then finds all the changes in master since then, and applies them to your branch. Then it attempts to (re)apply your changes to your branch to the “new” state of your branch.

This may fail, of course – someone may have modified master in way that makes one or more of your changes fail to apply cleanly anymore (what many revision control systems call a “conflict”, including git). But that's OK, you can edit the result, which will show both the “before your change” and “after your change” versions of the relevant code, fix the conflict, and then tell git to continue attempting to apply your changes.

Importantly: most of the time, it will not run into conflicts.

This leads to another benefit of rebasing: if you do get conflicts, you are still working in your branch, not master. So you can resolve things at your own pace/schedule, and not merge the results into master until everything is ready. After a successful rebase against master (as shown above), the merge step will always work with no conflicts between changes. This is quite a pleasant thing.