1)What does "git init" do

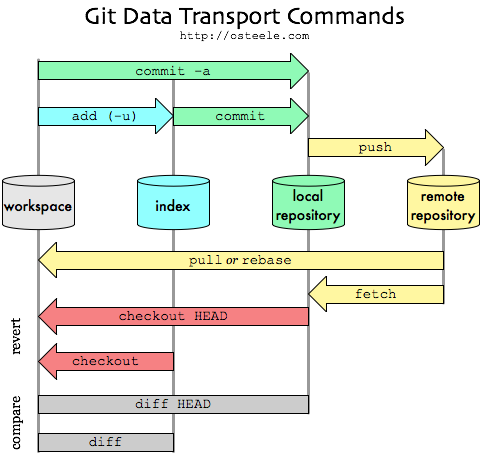
The G**it init** command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

2) What is staging in git?

* staging helps you split up one large change into multiple commits - Let's say you worked on a large-ish change, involving a lot of files and quite a few different subtasks. You didn't actually commit any of these -- you were "in the zone", as they say, and you didn't want to think about splitting up the commits the right way just then. (And you're smart enough not to make the whole thing on honking big commit!). Now the change is all tested and working, you need to commit all this properly, in several clean commits each focused on one aspect of the code changes. With the index, just stage each set of changes and commit until no more changes are pending. Really works well with git gui if you're into that too, or you can use git add -p or, with newer gits, git add -e.
* staging helps in reviewing changes - Staging helps you "check off" individual changes as you review a complex commit, and to concentrate on the stuff that has not yet passed your review. Let me explain. Before you commit, you'll probably review the whole change by using git diff. If you stage each change as you review it, you'll find that you can concentrate better on the changes that are not yet staged. git gui is great here. It's two left panes show unstaged and staged changes respectively, and you can move files between those two panes (stage/unstage) just by clicking on the icon to the left of the filename. Even better, you can even stage partial changes to a file. In the right pane of git gui, right click on a change that you approve of and choose "stage hunk". Just that change (not the entire file) is now staged; in fact, if there are other, unstaged, changes in that same file, you'll find that the file now appears on both top and bottom left panes!
* Staging helps when a merge has conflicts - When a merge happens, changes that merge cleanly are updated both in the staging area as well as in your work tree. Only changes that did not merge cleanly (i.e., caused a conflict) will show up when you do a git diff, or in the top left pane of git gui. Again, this lets you concentrate on the stuff that needs your attention -- the merge conflicts.
* Staging helps you keep extra local files hanging around - Usually, files that should not be committed go into .gitignore or the local variant, .git/info/exclude. However, sometimes you want a local change to a file that cannot be excluded (which is not good practice but can happen sometimes). For example, perhaps you upgraded your build environment and it now requires an extra flag or option for compatibility, but if you commit the change to the Makefile, the other developers will have a problem. Of course you have to discuss with your team and work out a more permanent solution, but right now, you need that change in your working tree to do any work at all! Another situation could be that you want a new local file that is temporary, and you don't want to bother with the ignore mechanism. This may be some test data, a log file or trace file, or a temporary shell script to automate some test... whatever. In git, all you have to do is never to stage that file or that change. That's it.
* staging helps you sneak in small changes - Let's say you're in the middle of a somewhat large-ish change and you are told about a very important bug that needs to be fixed asap. The usual recommendation is to do this on a separate branch, but let's say this fix is really just a line or two, and can be tested just as easily without affecting your current work. With git, you can quickly make and commit only that change, without committing all the other stuff you're still working on. Again, if you use git gui, whatever's on the bottom left pane gets committed, so just make sure only that change gets there and commit, then push!

3) What happens when you create a commit?

**git commit** is to **commit** the files that is staged in the local repo. **git** push is to fast-forward merge the master branch of local side with the remote master branch. But the merge won't always success. If rejection appears, **you** have to pull so that **you** can make a successful **git** push .



4) What are branches in git? Just learn basics about it

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

The “master” branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don’t bother to change it.

### Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called testing. You do this with the git branch command:

$ git branch testing

### Switching Branches

To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

$ git checkout testing

switch back to the master branch:

$ git checkout master

Switching branches changes files in your working directory

It’s important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.