# Introduction to distributed version control with git

Mark Longair

April 19, 2011

**Abstract**

This document is a companion to a talk I gave at the Institute for Neuroinformatics at the University / ETH Zürich. The aim is to introduce people to the version control system **git**[1] who haven't had experience of such a system before. git currently has huge momentum in the software development community, and offers a new way of working that makes development faster, more flexible and safer.

Many tutorials try to teach people to migrate to git from centralized version control systems, such as CVS, Subversion, Perforce, etc. by providing lists of approximately equivalent commands. While this helps people to use git exactly as a centralized version control system, it means that people miss out on the power (and fun) of using git. This introduction is an attempt to provide the right grounding for using git in as flexible a way as possible. There are lots of tutorials that explain how simple git can be to use, but the emphasis of this one is more on giving you an idea of the power of the system.

## 1 Why git rather than Mercurial?

If you aren't aware of Mercurial, you might as well go straight to section 2. However, since these are competing systems, I should discuss the choice before proceeding.

The only real reason that I prefer git to Mercurial[2], which is the other major distributed version control system used nowadays, is that I've had much more experience with git. Nowadays, the two systems are very similar in their model and capabilities, so both have all the advantages over CVS and Subversion that I think are so important. This document shouldn't be considered to be an attempt to advocate git over Mercurial, just to show how either system can help your work.

## 2 Forgetting CVS and Subversion

A typical introduction to git for Subversion users will start by presenting a set of git commands that are roughly equivalent to those that the readers are used to. For example, the git wiki's "SVN Crash Course" page[3] suggests the following:

[1] http://git-scm.com/
[2] http://mercurial.selenic.com/
[3] https://git.wiki.kernel.org/index.php/GitSvnCrashCourse

| svn command | git command |
|---|---|
| `svn checkout` *`url`* | `git clone` *`url`* |
| `svn update` | `git pull` |
| `svn commit` | `git commit -a && git push` |

It's quite correct that you can get along OK using these commands, but they only give you obscure hints[4] of why git is so much more flexible than Subversion - if you *just* follow tutorials like these you'll end up using git exactly like Subversion and, while you'll have the advantage of a much faster and more robust system, you won't gain from features like:

- Cheap and easy branching
- Disconnected use of the system
- Staging just the changes that you want in each new revision
- Sending experimental versions between systems without making them public
- Rewriting history before publishing your code

Indeed, it's best to try to completely forget about the Subversion / CVS model of the world when you're learning git.[5] Not only does this free you to think about version control in a more flexible way, you'll need to do it since many of git's commands have *very* confusing names if you've come from one of those earlier systems. (In particular, watch out for "checkout", "commit" and "update".)

With any way of using git, however, you'll find plenty of benefits - in particular:

- Painless (almost magical) merging
- Excellent performance, even with large repositories
- It's difficult to lose data with git
- Always having the complete history of your repository - every clone of a repository is like a backup
- The complete history of the project is stored very efficiently - this often takes less space than the working copy

# 3 Basic principles

These next sections provide a general description of some principles that you need to keep in mind when using git.

[4] e.g. Why are "commit" and "push" separate? What is the "-a" there for?

[5] The "hg-init" introduction to Mercurial by Joel Spolsky makes the same point as this very forcefully: http://hginit.com/00.html

## 3.1 No repository is special

In git there is no concept of a central repository - you can push to or pull from any repository that you have access to. In practice, many projects do have one repository that's considered distinguished, in that it has been decided by the developers that that is the repository where they will publish completed work, and that's the repository they will tell people to clone from. However, this is just a convention that's decided on by the developers, not any requirement of git. (In contrast, to share code with Subversion, it must go through the central repository.)

## 3.2 Each repository is completely standalone

The most critical change of mindset for using git is that every repository is completely functional on its own, without any dependency on other repositories. Very few of the everyday commands that you use in git need a remote repository - the important ones that do are just `git pull`, `git push` and `git fetch`, which are all specifically for transferring versions between repositories. All the commands for creating new versions, comparing versions, examining the repository history, etc. etc. all work whether you have internet access or not, so even if you're in a situation where internet access is awkward, (e.g. on a plane or a train) you can carry on working and tracking your changes as normal.

Another nice associated feature is that when you initially get a copy of a repository with `git clone` (or get new changes from a repository with `git fetch`) you get every branch from that repository, and everything that's required to work with them.[6] So, if you're setting off on a trip, you can just run `git fetch origin` and you know that you'll be able to use any branch that anyone has pushed to the `origin` repository.

## 3.3 Commits have a globally unique name

What you might refer to in other version control systems as a "version" is described more precisely in git as a **commit**. Each commit has an identifier, known as an **object name** that looks something like:

`7472ea9087e25cec0b682189520b5fde7d48b53e`

You see these identifiers a lot when using git, so it's important to understand what they mean. These are keys that refer to a git object (such as a commit, a file, a directory, a commit, etc.) absolutely *uniquely*. They are based on a cryptographically strong hash of the object's content, so you can be sure that if you and a collaborator in another country have a commit with the object name `7472ea9087e25cec0b682189520b5fde7d48b53e` that you both have the same version of the source tree you're working on. In fact, that statement can be even stronger, since the object name is calculated by hashing information that includes the following:

1. The author's name and email address.

[6] You can see these with `git branch -a`, which shows "remote-tracking branches" as well as the branches that you're working on locally. The remote-tracking branches typically have names that look like `origin/whatever`.

2. The name and email address of the person who created the commit.
3. The object name of a **tree**, which completely defines the state of the files in your source code in this commit.
4. A commit message, describing what changes were introduced by this new state of the tree.
5. Any parent commits.

Point 5 is particularly important - each commit's object name depends on the object names of its parents, which in turn depend on the object names of *their* parents, and so on. In other words, the name of a commit depends on its exact history. Two commits which end up being identical in every way, except for different histories, will have different object names - this also means that if you cryptographically sign a commit, you're also providing a way for people to be sure that the history of the version they have is the same as your original.

The long hashes that make up object names may look awkward to deal with, but git commands will always understand an abbreviated version if it is unambigous in your repository. For example, you could normally just use `7472ea` to refer to the commit above.

### 3.4 The history of your project is a directed acyclic graph

The other important implication of a "commit" being partly defined by zero or more parents, is that the history of your project is a graph. For example, see Figure 1. In this example, 0b6821 has zero parents, since it was the first commit in the repository.[7] All the rest of the commits have one parent, apart from the most recent one, e1f67a, which has two parents. This is a merge commit, whose purpose is to represent the state of the tree after the work done in both parents is merged together. Merge commits aren't particularly special - they still just represent the exact state of your source tree, but they have more than one parent.

It's essential for understanding branching and merging in git that you think of the history as a graph like this. In Subversion and CVS, branching is awkward enough that people usually just do linear development, so this may be a new way of thinking about the history of a project. Every installation of git comes with a useful tool called "gitk", which is a graphical tool for examining your project's history that shows you this commit graph - try: `gitk --all` in a repository to see the history of all branches.

### 3.5 What makes up your repository?

It's useful to know some basics about the structure of a git repository as it is stored on you filesystem. Usually you will be using a **non-bare repository**, or a repository with a **working tree**. If you've created such a repository in a directory called toaster-simulator, then the directory structure might look like:

[7] You can have multiple such root comits in project, but that's not the typical case.

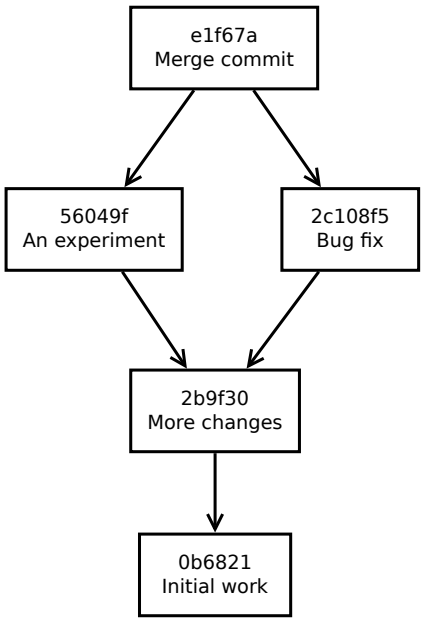


Figure 1: An simple example commit graph.

```
toaster-simulator/
    .git/
        HEAD
        config
        index
        objects/
        refs/
        ...
    README
    Makefile
    src/
        toaster.c
```

There is only one `.git` directory at the top level of your repository.[8] This directory contains the entire history of your repository, all its branches, the objects that make up very file of every commit in your history, etc. (For example, the `.git/objects` directory contains the database that maps object names to files, commits, and so on; `HEAD` points to your current branch; etc.) You should never manually delete or change files in this directory, or you will risk corrupting your repository. Everything outside the `.git` directory is your working tree. You would just edit these files as normal when developing your project.

Perhaps the most surprising thing to people who are new to git is that if you switch from one branch to another, the files in your working tree may completely change. Although this may be alarming the first time you see it, this is a great feature - you almost never need to have more than one copy of a repository on your computer, since switching from one branch to another is so fast and easy. (To stop you from losing data, git will prevent you from switching branches like this if you have changes in your working tree that haven't been recorded in a commit, and those changes are in files that would be altered in any way by the switch of branch. Once you have created a commit with your files at a particular state, git makes it very difficult for you to lose those files.)

The other type of repository is a **bare repository**, which is essentially like the `.git` directory but without a working tree. You would typically use this for a repository that many people will be pushing their changes to - you won't be developing on this repository so the working tree would just get in the way. Conventionally you would name the directory that contains the bare repository with the project name and the .git extension. For example, on a remote server you might have a bare repository for the toaster-simulator project that looks like this:

```
toaster-simulator.git/
    HEAD
```

[8] In fact it is possible to have nested repositories, each with their own .git directory, using git's **submodule** feature - that is beyond the scope of this tutorial, however.

```
config
index
objects/
refs/
...
```

When you're first using git, you probably won't need to use bare repositories, but it's good to be aware of them.

# 4 Tutorial examples

This section contains some conventional recipes for doing basic operations with git. A golden rule for using git is that if you're ever confused about what's going on, try runing:

```
git status
```

... which will tell you a summary of:

- Which branch you are on.
- Which files have changes that you have "staged" to be part of the next commit.
- Which files are being tracked but have unrecorded changes in them.
- Which files in this directory are not yet being tracked in git.

As mentioned above, `gitk --all` may also be helpful for you to understand the situation that your repository is current in.

## 4.1 Tell git who you are

Whenever you create new commits in git, it will list you as the author. So that you don't have to remember to specify your details every time, the first thing you should do is to globally set your name and email address. For example:

```
$ git config --global user.name "My Full Name"
$ git config --global user.email "an.invented@email.address"
```

The "–global" option means to store these configuration values in ".gitconfig" in your home directory, so you should only have to do this once for each user account. While you're setting up helpful defaults, it makes the output of git more readable if you tell it to use colours, which you can do with:

```
$ git config --global color.ui auto
```

## 4.2 Creating a new repository

Typically you create a new git repository in one of two ways, depending on whether you're starting a new project or you're wanting to work on an existing one.

If you're starting a new repository from scratch, you just create a new directory and then run `git init` in that directory. For example:

```
$ mkdir toaster-simulator
$ cd toaster-simulator
$ git init
Initialized empty Git repository in /home/mark/tmp/toaster-simulator/.git/
$ ls -a
.  ..  .git
```

On the other hand, if you want to work on an existing repository, you would use the command `git clone <URL>`, where `<URL>` is a URL that points to the repository. That URL should be published by the project's developers somewhere. For example, if you wanted to clone a repository with the source code for ImageJ, you might do:

```
$  git clone ssh://contrib@pacific.mpi-cbg.de/srv/git/imagej.git
Initialized empty Git repository in /home/mark/tmp/imagej/.git/
remote: Counting objects: 7033, done.
remote: Compressing objects: 100% (2301/2301), done.
remote: Total 7033 (delta 5611), reused 5897 (delta 4726)
Receiving objects: 100% (7033/7033), 7.81 MiB | 1.38 MiB/s, done.
Resolving deltas: 100% (5611/5611), done.
$ cd imagej
$ ls
.            compile.sh    images       policy
..           Fakefile      macros       release-notes.html
applet.html  .git          Makefile     run_appletviewer.bat
aREADME.txt  .gitignore    MANIFEST.MF
build.xml    ij            nbproject
compile.bat  IJ_Props.txt  plugins
```

The URL that you're using here says to use SSH as a transport. Typically you would try to clone over SSH if you want to be able to push back your changes to the repository, or use the `git://` or `http://` protocols if you only need a read-only copy. (In fact, it's now possible to push to repositories over HTTP and HTTPS if the server in question supports git's Smart HTTP protocol, which is very useful if you're behind a restrictive firewall.) `git clone` does a number of things so that you can start working on the repository straight away:

- Fetches all the branches from the remote repository, and all the objects you need to work from them
- Checks out the main branch for you - this branch is usually called `master`
- Sets up a so-called "remote" for you called `origin`, that refers to the repository you cloned from.[9]

Next, you need to know how to add new files to your repository and create new commits.

## 4.3 Add some files and creating your first commit

Let's assume that you've started the `toaster-simulator` project by writing a README and a Makefile.

To add those files, you can do:

```
$ git add README Makefile
```

If you type `git status` now, you should see that both of these files are in the "Changes to be committed" section, and each is listed as a "new file".

Then, to create a commit you should type:

```
$ git commit
```

This will launch your default editor[10] so that you can write a message that describes this commit. It's recommended that you make the first line a short summary of what has changed, then leave an empty line and finally leave a more detailed description of your changes. For example:[11]

```
Added a skeleton Makefile and a README explaining the project

In the future we will use a different build system, but initially
GNU make will be quite sufficient.  The README describes the
goals of the toaster-simulator project.
```

You should save the file, and exit your editor. Then you should see on the command line the following output:

```
[master (root-commit) 1ddff69] Added a skeleton Makefile and a README explaining the proj
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 Makefile
create mode 100644 README
```

[9] "Remotes" are like a nickname for another repository, so you don't have to remember the URL. There is more on remotes in Section 4.6

[10] Your default editor can be changed by setting the `GIT_EDITOR` environment variable, or by a number of other mechanisms listed in the documentation for git commit.

[11] Note that this isn't a great example of a commit in general. Ideally each commit should only introduced logically grouped changes, and adding a README and adding a Makefile really should be distinct.

... which indicates that everything went well. Those changes are now safely recorded in a commit, and will be difficult for you to lose. You can now run:

```
$ git log
```

... to see the history of your current branch, which should now just have one commit in it. This output will also show you the full object name of the commit you just created. `git log`, like many git commands, presents its output in a pager (typically `less`) - to exit the pager, just press "q".

Similarly, if you want to remove a file from your repository, you could run:

```
$ git rm Makefile
```

... and then commit that change as before. Perhaps you only want to include a one-line commit message this time, in which case you can just specify it on the command line:

```
$ git commit -m "Remove the unnecessary Makefile"
[master bb0856f] Remove the unnecessary Makefile
1 files changed, 0 insertions(+), 1 deletions(-)
delete mode 100644 Makefile
```

Similarly, to rename or move a file or directory, you can use `git mv <OLD-NAME> <NEW-NAME>.` In git, this is just the same as removing a file and adding it with a different name. git doesn't track renames explicitly - it just works out that they must have happened by comparing the trees from one commit to another.

## 4.4 What do "git add", "git rm" and "git mv" really do?

One of the important differences between git and any other version control system that I know of is its concept of the **index**, which is also known as the **staging area**. This is a file in your .git directory that records the exact state of your project that will make up the next commit. (If you haven't made any changes, the index is exactly the same as the state at your last commit.)

When you run `git add Test.java`, that's really saying, "I want the current state of the file Test.java to be in the next commit." If you then go on to make more changes to `Test.java`, that won't affect what's recorded in the next commit unless you run `git add Test.java` again.

This can be confusing to people who are new to git, since if you have run `git add Test.java`, then made more changes to it, `git status` will tell you:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
# modified:   Test.java
#
# 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:   Test.java
```

... but that makes perfect sense if you remember about the staging area - there are changes to `Test.java` to be commited (the state when you ran `git add Test.java`), but there are also further changes on top of that that you haven't staged.

So why is this concept of a staging area useful? A typical situation would be if you've been working away making lots of changes and bug fixes, and then realise you ought to commit them. However, you realise that really there are two logically different bug fixes that you've done, one which affects the files `foo.c` and `bar.c`, while the other one just affects `quux.c`. Then you can easily create two commits by doing:

```
$ git add foo.c
$ git add bar.c
$ git commit -m "Fix consistent mispellings of 'receive'."
$ git add quux.c
$ git commit -m "Remove a redundant function."
```

This helps you to create commits that make sense so that you and other people can easily understand the changes you've been made in the future. Having small and logically groups changes also means that when you discover the awesome tool `git bisect`, you'll be able to track down the changes that introduced a bug much more accurately.

A more advanced technique that might be of interest is if you want to only commit some of the changes *within a particular file*. In that case you can use `git add -p foo.c`, and only stage particular parts of the file that you've changed.

Similarly, `git rm Makefile` says to stage the removal of the file for the next commit, but also to remove it from your working copy. Because of the staging area, however, it's sometimes useful to say that you just want to stop tracking the file in git, but leave the copy in your working directory. You can do that with `git rm --cached Makefile` - the file will be deleted in the staging area, but your working copy won't be touched. (The `--cached` parameter in many git commands means "I only want this to apply to the staging area".)

There are two pairs of commands that are useful for working out what you've already staged, and what's left:

- `git diff` always shows you the difference between your working tree and what's been staged. i.e. that shows you the changes that you might still want to consider for staging with `git add`.
- `git diff --cached` always shows you the difference between the last commit and what's been stage. i.e. everything that you've already staged.

It takes a little while to get used to this idea of having a staging area, but it's very powerful.

## 4.5 Branches

### 4.5.1 What are branches?

When using git, it's a good idea to create lots of branches to organize your work. These are often known as "topic branches", to suggest that you should create a new branch for each new idea or feature that you want to work on. This makes sense because branches in git are so lightweight, and merging is very reliable compared to CVS or Subversion. The implementation of branches in git is really very simple - they're just stored as a file on disk which contains the object name of the last commit on that branch. Every commit that is an ancestor of the commit at the tip is considered to be a part of the branch. If you create a commit when you're on a particular branch, the object name in that file is updated to refer to the new commit. This means that most operations that involve branches are really cheap to perform.

To switch between different branches, you use the command

```
$ git checkout <branch-name>
```

(This is the command I described above that will potentially make your working tree look completely different.) As a shortcut, you can always switch back to your previous branch with:

```
$ git checkout -
```

### 4.5.2 Listing your branches

To see all the branches that you can work directly on, you can use the command "git branch":

```
$ git branch
  experiment
* master
```

The "*" (star) indicates the branch that you are currently working on.

### 4.5.3 Creating new branches

Suppose you're on the default branch, `master`, and want to create a new branch called `experiment`. There are two common ways of doing this - the first one creates the new branch pointing to the current commit, but will leave you still on `master`:

```
$ git branch experiment
```

The second version will do the same, but also switch to the new branch:

```
$ git checkout -b experiment
```

(In other words, the latter is a shorthand for `git branch experiment` followed by `git checkout experiment`.)

You don't have to create a branch just based on the commit you're at, though - you can add another parameter to either of those commands to use a different starting point. For example:

```
$ git checkout -b experiment 1ddff69
```

... will switch you to a new branch that is at the commit `1ddff69`.

#### 4.5.4 Not being on a branch, or "detached HEAD"

It's worth briefly mentioning that in git there's also a possibility of not being on *any* branch. To explain this, the current branch is tracked by a file called HEAD, which normally points to the name of a branch, such as `master` or `experiment`. However, if you tell `git checkout` that you want to switch to a `1ddff69` instead of a branch name, it will produce a warning, but still change HEAD to point to that exact commit. (That is why this is sometimes known as the "detached HEAD" state - HEAD is no longer linked to a branch.) This turns out to often be useful - for example, if you're trying to jump around in your repository's history, seeing what the project looked like at different commits, you don't want to have to change your branch to be able to do that. You can still create new commits when you're in the detached HEAD state, but there won't be any branch pointing to them. To make sure that such commits don't get garbage collected, it's a good idea to create a new branch for them with:

```
$ git checkout -b another-experiment
```

#### 4.5.5 Merging branches

When you've finished the feature that you were working on in the branch `experiment`, you should merge that work into your main branch, `master`. This is easy to do with the command `git merge`. In its simple usage, this only takes one parameter, which is the name of the branch that you want to merge into your current branch. (It's important to remember that it's always this way round - git generally only changes the branch that you're currently on, and merging will advance the branch that you're merging into.) So, you could do this with:

```
$ git checkout master
$ git merge experiment
```

The merge strategies that git uses are very good - they will very rarely discover conflicts where there isn't really some conflicting change that needs to be resolved.[12] However, when you do have a real conflict, these are easy to resolve. If you run `git status`, it will tell you which files are "unmerged", and you should fix the conflicts that are marked in each of them with the '<<<<<', '=====' and '>>>>>>' markers that you should be familiar with from other version control systems. When you've resolved the problems in a particular file, say `TestRunner.java`, you just stage that fixed version of the file with:

```
$ git add TestRunner.java
```

... and once you've similarly fixed all the other files, so `git status` shows none left unmerged, you should just run:

```
$ git commit
```

... and leave the default commit message, showing which files you resolved conflicts in.

## 4.6 Interacting with other repositories

So far, everything I've discussed has been local to your repository. However, you also typically need to know how to transfer work from one repository to another. Sending your changes to a remote bare repository can be done with `git push`, and getting changes from a remote repository into your current one can be done with `git pull`. In fact, `git pull` is a convenient way to do two operations in one - `git fetch`, followed by a `git merge`. I think that for learning git, it's more generally useful to start using `git fetch` and `git merge` instead of `git pull`, and so that's the treatment given here.

Suppose you're working on a project, and someone tells you that they've forked (cloned) your project on github, and they'd like you try out a branch of theirs - let's say the branch is called `subtle-fix` and the repository's URL is `git://github.com/whoever/toaster-simulation.git`. The first thing you need to know is how to refer to that repository in your local repository. You do this by adding a "remote".

### 4.6.1 Add a new remote

"Remotes" are like nicknames for particular repositories. In this case you would want to add one to refer to this contributor's repository on GitHub - let's say you decide to then call the remote `github`. You could then add that with the command:

```
$ git remote add github git://github.com/whoever/toaster-simulation.git
```

Now if you run the command "git remote -v" it will show you all the remotes that you have set up. That might look like:

[12] This is largely because the merge can consider not only the two branch tips that are being merged, but also their common ancestor.

```
origin git@github.com:mhl/toaster-simulation.git (fetch)
origin git@github.com:mhl/toaster-simulation.git (push)
github github git://github.com/whoever/toaster-simulation.git (fetch)
github github git://github.com/whoever/toaster-simulation.git (push)
```

(Note that two URLs are listed for each remote in that output, since you can actually have a different URL set up for fetching and pushing. You're unlikely to need to use that, however.)

Now you can use that remote to tell `git fetch` and `git push` which repository to talk to.

### 4.6.2 Fetch branches from that remote

To make everything from that contributor's repository available in your own repository, you can just run:

```
$ git fetch github
```

Hopefully you will see output that looks like:

```
From git://github.com/whoever/toaster-simulation.git
 * [new branch]      master     -> github/master
 * [new branch]      subtle-fix -> github/subtle-fix
```

That tells you that two branches were found in that repository, `master` and `subtle-fix`, and git fetch has created a so-called "remote-tracking branch" for each of those, called `github/master` and `github/subtle-fix`. The standard naming scheme for remote-tracking branches is `<name-of-remote>/<name-of` Remote-tracking branches are similar to normal branches, in that again they're just a pointer to a particular commit. However, they're just like a cache of the states of those branches the last time you ran `git fetch` - you can't directly work on them. If you want to see what that contributor's branch looks like, you have to create your own branch based on the remote-tracking branch. You would do that with:

```
$ git checkout -b subtle-fix github/subtle-fix
```

... just as you would to create a branch based on any other commit.[13] If that contributor tells you later that they've made more changes to their branch, and you want to incorporate them, you just need to update the remote-tracking branch, and then merge it into your subtle-fix branch:

```
$ git fetch github
From git://github.com/whoever/toaster-simulation.git
  02e6cbe..7472ea9  subtle-fix     -> github/subtle-fix
```

[13] There is some additional magic here, though, when you base a branch on a remote-tracking branch - git will set up configuration variable that associate that branch with the remote-tracking branch. That's beyond the scope of this tutorial at the moment, however.

```
$ git checkout subtle-fix
$ git merge github/subtle-fix
```

If you want to list all the remote-tracking branches in your repository, you can use the following command:

```
$ git branch -r
  origin/add-new-menu
  origin/experiment
  origin/master
  origin/subtle-fix
```

When you originally clone a repository, one of the commands that's run by "git clone" is "git fetch origin", so if you run "git branch -r" after cloning you'll see all the branches that people have pushed to that repository.

### 4.6.3 Pushing changes to another repository

You can update branches in a remote repository by using:

```
$ git push <remote> <local-branch-name>:<remote-branch-name>
```

For example, suppose you wanted to push some very experimental work to your origin remote, you might do:

```
$ git push origin risky-experiment:risky-experiment
```

... however, if the branch name is the same on both sides, you can abbreviate this to:

```
$ git push origin risky-experiment
```

I would recommend that you always specify the branch name when using `git push`, since just doing `git push origin` will push every branch of yours that has a branch with the same name on the remote. Normally this isn't what you would want.

# 5 Further Reading

There are many git commands and topics that I haven't covered in this brief introduction, in particular those for rewriting or copying history, such as `git rebase` and `git cherry-pick`. Fortunately there are many good resources on learning git.

A nice introduction to the concepts in git can be found on the web page "Git for Computer Scientists": http://eagain.net/articles/git-for-computer-scientists/

As far as books go, “Pro Git” by Scott Chacon is an excellent introduction to git, and you can read it online:

`http://progit.org/book/`

The git community book is another high quality and freely available resource for learning git:

`http://book.git-scm.com/`