1.1 Getting Started - About Version Control

This chapter will be about getting started with Git. We will begin by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it set up to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all set up to do so.

About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

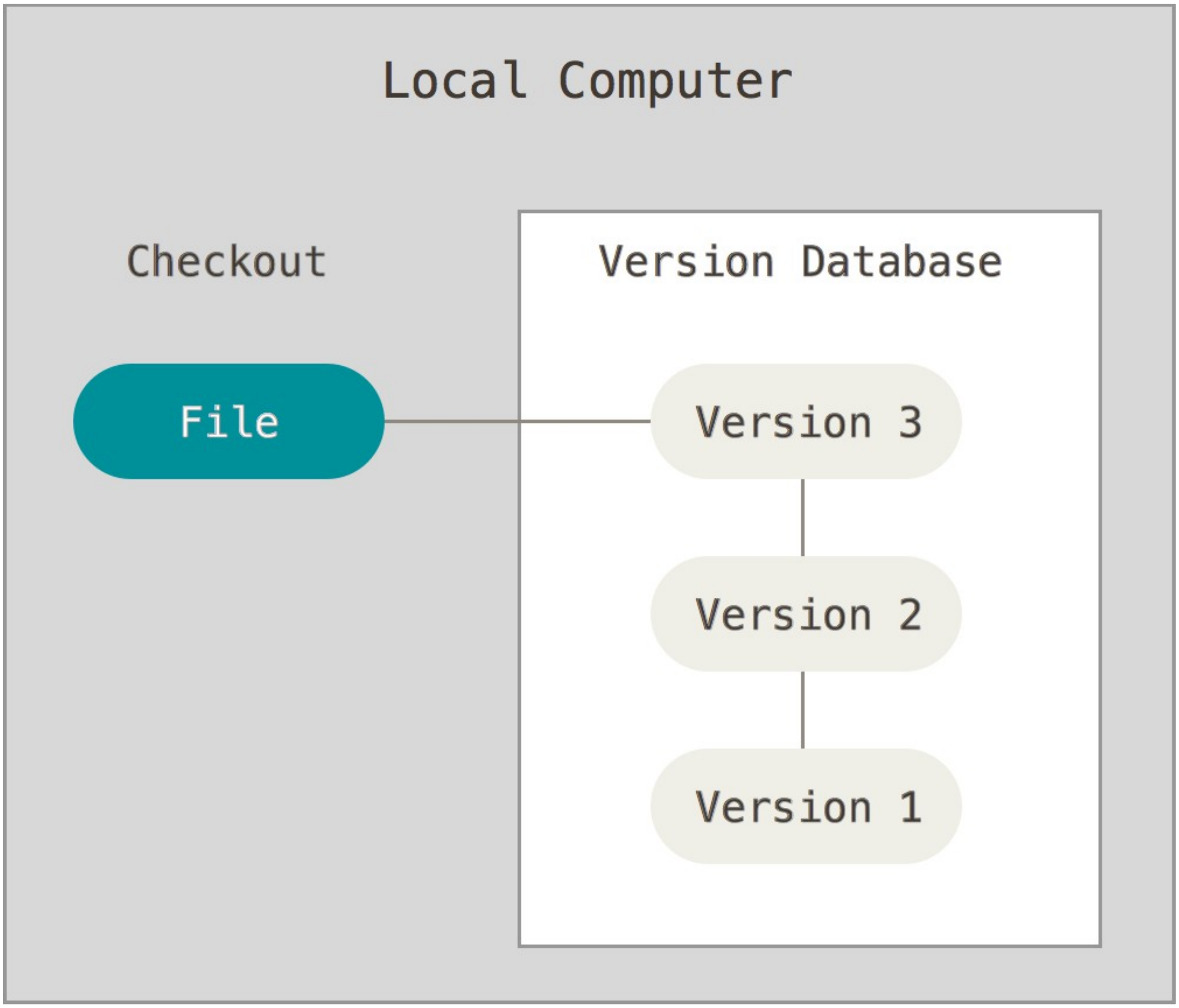

Figure 1-1. Local version control.

One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. Even the popular Mac OS X operating system includes the `rcs` command when you install the Developer Tools. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

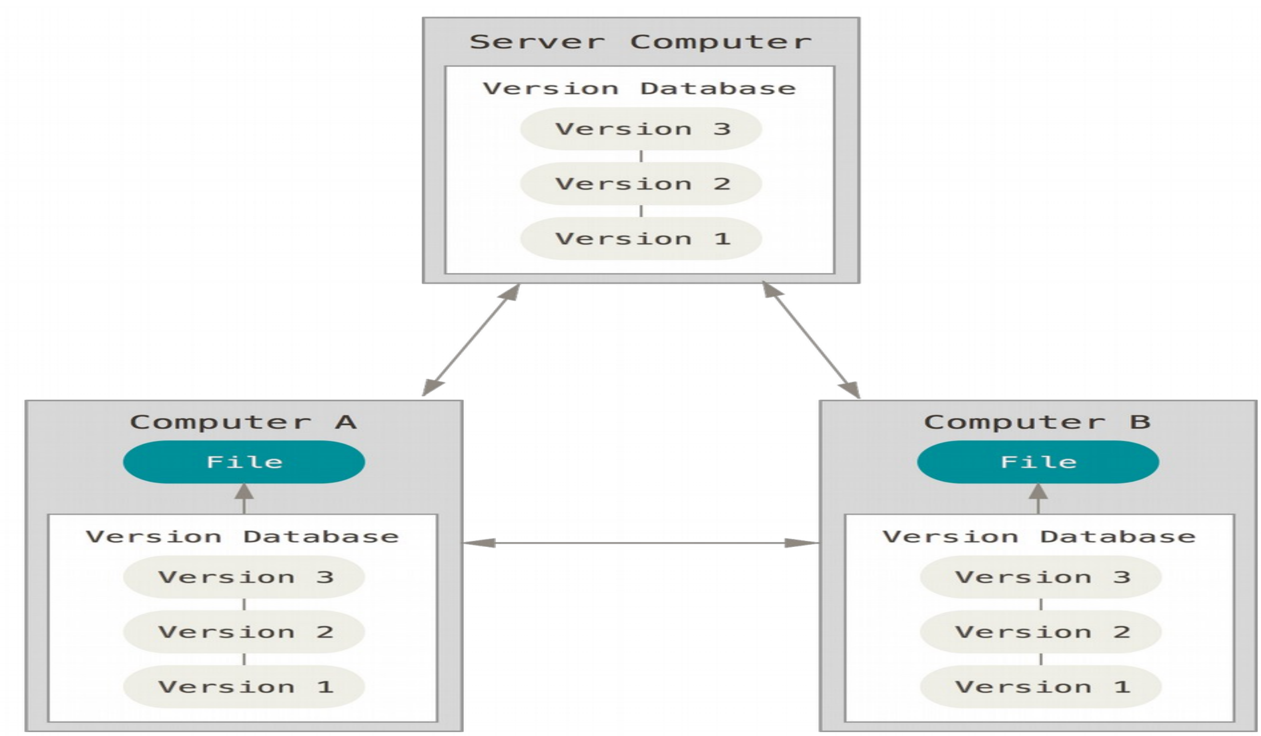

Figure 1-3. Distributed version control.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

1.2 Getting Started - A Short History of Git

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

The Linux kernel is an open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's incredibly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (See Git Branching).

1.3 Getting Started - Git Basics

Git Basics

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as Subversion and Perforce; doing so will help you avoid subtle confusion when using the tool. Git stores and thinks about information much differently than these other systems, even though the user interface is fairly similar, and understanding those differences will help prevent you from becoming confused while using it.

Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

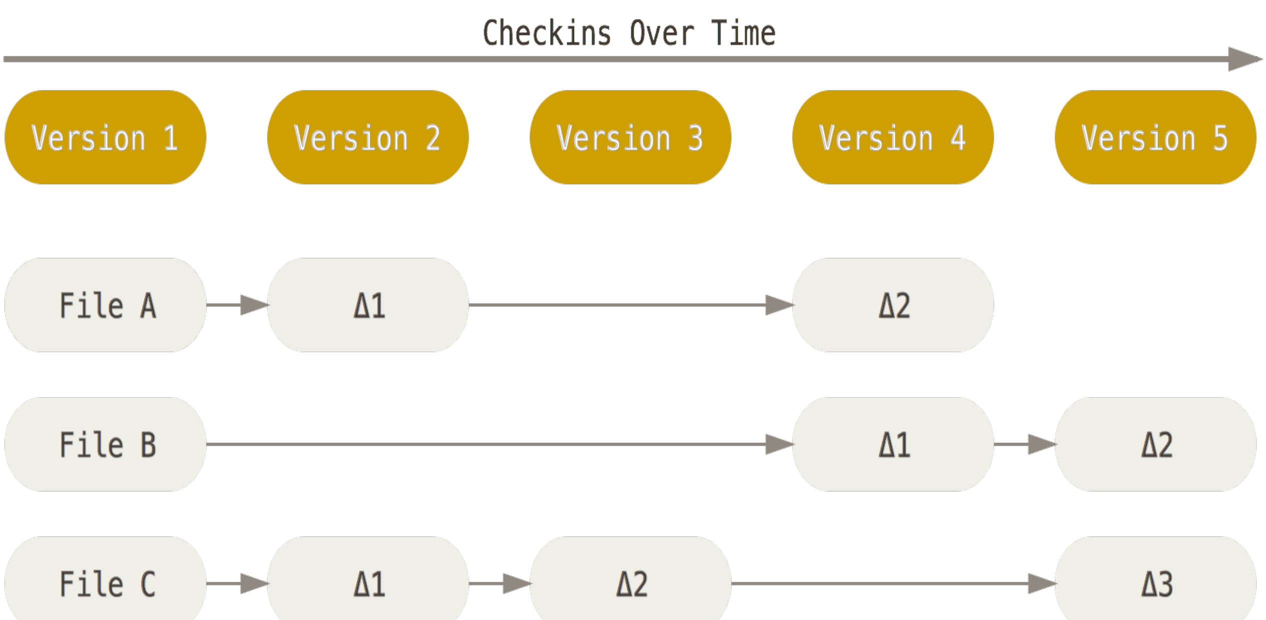

Figure 1-4. Storing data as changes to a base version of each file.

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

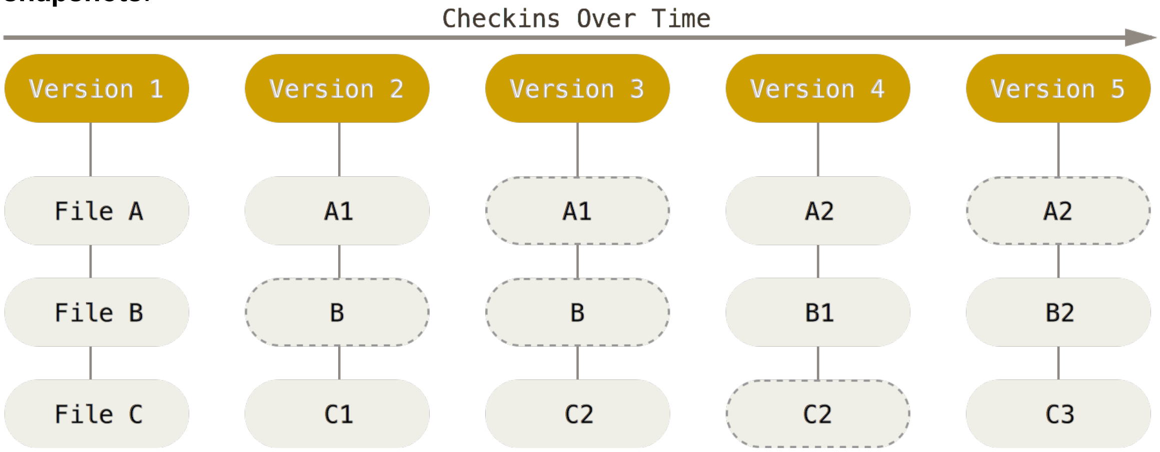

Figure 1-5. Storing data as snapshots of the project over time.

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We’ll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in [Git Branching](#).

Nearly Every Operation Is Local

Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network. If you’re used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for you – it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can’t do much when you aren’t connected to the server; and in Subversion and CVS, you can edit files, but you can't commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Git Generally Only Adds Data

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As in any VCS, you can lose or mess up changes you haven't committed yet; but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see Undoing Things.

The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.

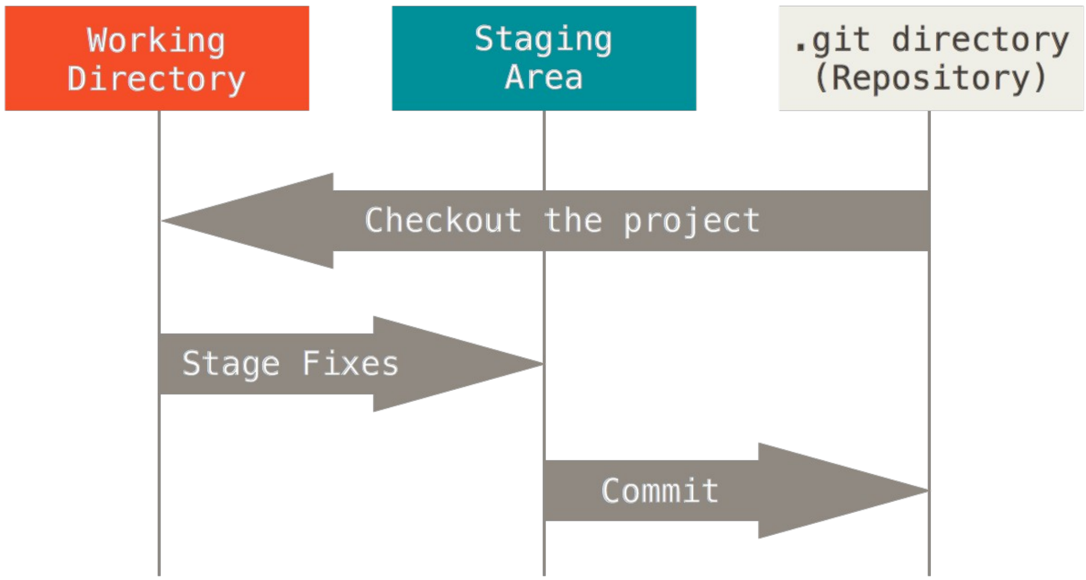

Figure 1-6. Working directory, staging area, and Git directory.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It's sometimes referred to as the "index", but it's also common to refer to it as the staging area.

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it's considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In [Git Basics](), you'll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

1.4 Getting Started - The Command Line

The Command Line

There are a lot of different ways to use Git. There are the original command line tools, and there are many graphical user interfaces of varying capabilities. For this book, we will be using Git on the command line. For one, the command line is the only place you can run **all** Git commands – most of the GUIs only implement some subset of Git functionality for simplicity. If you know how to run the command line version, you can probably also figure out how to run the GUI version, while the opposite is not necessarily true. Also, while your choice of graphical client is a matter of personal taste, *all* users will have the command-line tools installed and available.

So we will expect you to know how to open Terminal in Mac or Command Prompt or Powershell in Windows. If you don't know what we're talking about here, you may need to stop and research that quickly so that you can follow the rest of the examples and descriptions in this book.

1.5 Getting Started - Installing Git

Installing Git

Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

NOTE

This book was written using Git version **2.0.0**. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently if you're using an older version. Since Git is quite excellent at preserving backwards compatibility, any version after 2.0 should work just fine.

Installing on Linux

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

```
$ sudo yum install git-all
```

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

```
$ sudo apt-get install git-all
```

For more options, there are instructions for installing on several different Unix flavors on the Git website, at http://git-scm.com/download/linux.

Installing on Mac

There are several ways to install Git on a Mac. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run *git* from the Terminal the very first time. If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. An OSX Git installer is maintained and available for download at the Git website, at http://git-scm.com/download/mac.

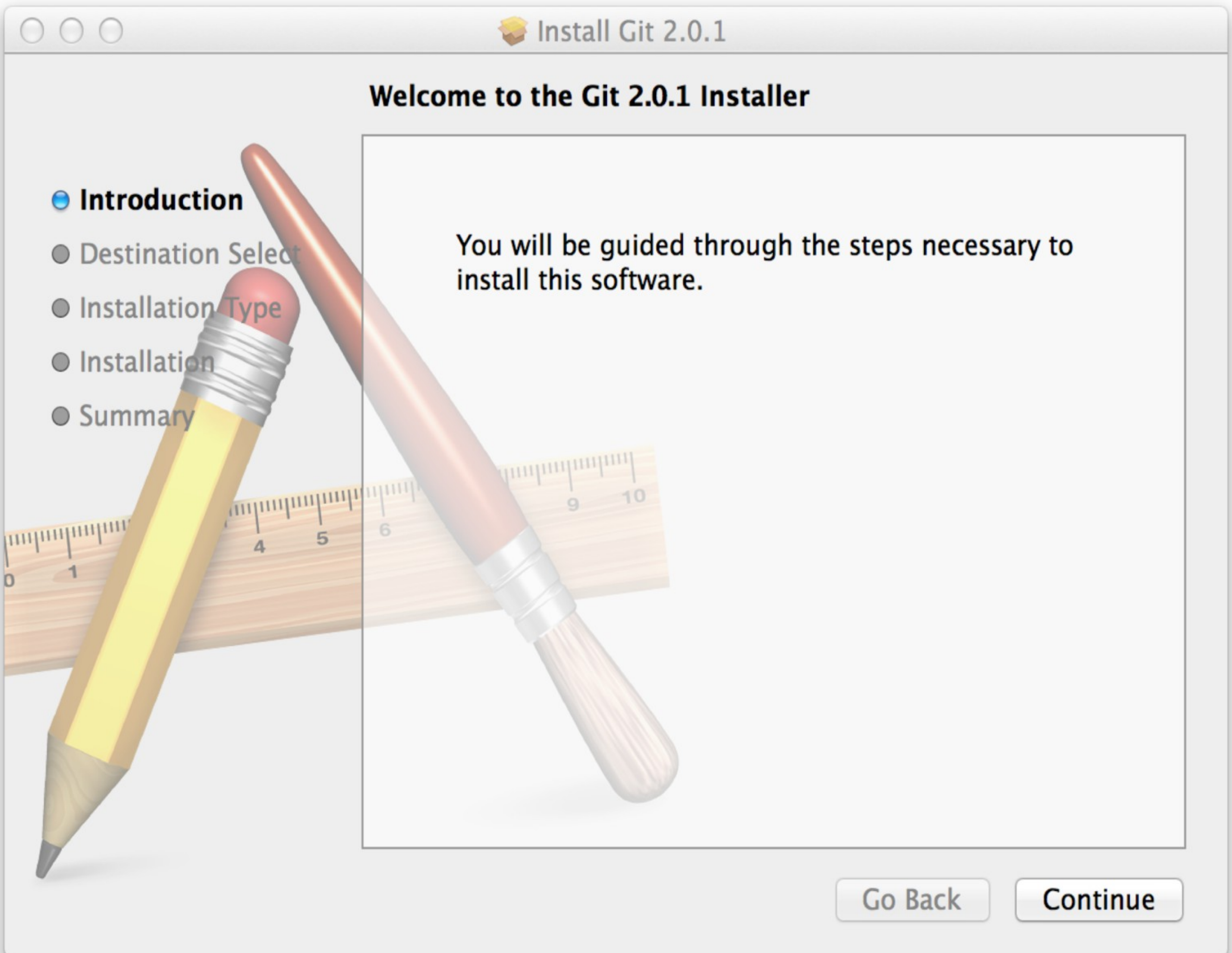

Figure 1-7. Git OS X Installer.

You can also install it as part of the GitHub for Mac install. Their GUI Git tool has an option to install command line tools as well. You can download that tool from the GitHub for Mac website, at*http://mac.github.com*.

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to *http://git-scm.com/download/win* and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to *https://git-for-windows.github.io/*.
Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub for Windows website, at*http://windows.github.com*.

Installing from Source

Some people may instead find it useful to install Git from source, because you'll get the most recent version. The binary installers tend to be a bit behind, though as Git has matured in recent years, this has made less of a difference.

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

```
$ sudo yum install curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

In order to be able to add the documentation in various formats (doc, html, info), these additional dependencies are required (Note: users of RHEL and RHEL-derivatives like CentOS and Scientific Linux will have to enable the EPEL repository to download the `docbook2X` package):

```
$ sudo yum install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Additionally, if you're using Fedora/RHEL/RHEL-derivatives, you need to do this

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

due to binary name differences.

When you have all the necessary dependencies, you can go ahead and grab the latest tagged release tarball from several places. You can get it via the Kernel.org site, at*https://www.kernel.org/pub/software/scm/git*, or the mirror on the GitHub web site, at*https://github.com/git/git/releases*. It's generally a little clearer what the latest version is on the GitHub page, but the kernel.org page also has release signatures if you want to verify your download.

Then, compile and install:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

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

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

1.6 Getting Started - First-Time Git Setup

First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer; they'll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.
2. `~/.gitconfig` or `~/.config/git/config` file: Specific to your user. You can make Git read and write to this file specifically by passing the `--global` option.
3. `config` file in the Git directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository.

Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Users\$USER` for most people). It also still looks for `/etc/gitconfig`, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using Git for Windows 2.x or later, there is also a system-level config file at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in `C:\ProgramData\Git\config` on Windows Vista and newer. This config file can only be changed by `git config -f <file>` as an admin.

Your Identity

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the `--global` option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the `--global` option when you're in that project.

Many of the GUI tools will help you do this when you first run them.

Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system's default editor, which is system dependant.

If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

While on a Windows system, if you want to use a different text editor, such as Notepad++, you can do the following:

On a x86 system

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -nosession"
```

On a x64 system

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad+
+.exe' -multiInst -nosession"
```

NOTE
Vim, Emacs and Notepad++ are popular text editors often used by developers on Unix based systems like Linux and OS X or a Windows system. If you are not familiar with either of these editors, you may need to search for specific instructions for how to set up your favorite editor with Git.

WARNING
You may find, if you don't setup an editor like this, you will likely get into a really confusing state when they are launched. Such example on a Windows system may include a prematurely terminated Git operation during a Git initiated edit.

Checking Your Settings

If you want to check your settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.
You can also check what Git thinks a specific key's value is by typing `git config <key>`:

```
$ git config user.name
John Doe
```

1.7 Getting Started - Getting Help

Getting Help

If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help &lt;verb&gt;
$ git &lt;verb&gt; --help
$ man git-&lt;verb&gt;
```

For example, you can get the manpage help for the config command by running

```
$ git help config
```

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren't enough and you need in-person help, you can try the `#git` or `#github` channel on the Freenode IRC server (irc.freenode.net). These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

1.8 Getting Started - Summary

Summary

You should have a basic understanding of what Git is and how it's different from the centralized version control system you may have previously been using. You should also now have a working version of Git on your system that's set up with your personal identity. It's now time to learn some Git basics.

2.1 Git Basics - Getting a Git Repository

If you can read only one chapter to get going with Git, this is it. This chapter covers every basic command you need to do the vast majority of the things you'll eventually spend your time doing with Git. By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes. We'll also show you how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

[Initializing a Repository in an Existing Directory]

If you're starting to track an existing project in Git, you need to go to the project's directory and type:

```
$ git init
```

This creates a new subdirectory named `.git` that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet. (See Git Internals for more information about exactly what files are contained in the `.git` directory you just created.)

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few `git add` commands that specify the files you want to track, followed by a `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

We'll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is `git clone`. If you're familiar with other VCS systems such as Subversion, you'll notice that the command is "clone" and not "checkout". This is an important distinction – instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run `git clone`. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there – see Getting Git on a Server for more details).

You clone a repository with `git clone [url]`. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

```
$ git clone https://github.com/libgit2/libgit2
```

That creates a directory named "libgit2", initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new `libgit2` directory, you'll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than "libgit2", you can specify that as the next command-line option:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

That command does the same thing as the previous one, but the target directory is called `mylibgit`.

Git has a number of different transfer protocols you can use. The previous example uses the `https://` protocol, but you may also see `git://` or `user@server:path/to/repo.git`, which uses the SSH transfer protocol. Getting Git on a Server will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

2.2 Git Basics - Recording Changes to the Repository

Recording Changes to the Repository

You have a bona fide Git repository and a checkout or working copy of the files for that project. You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

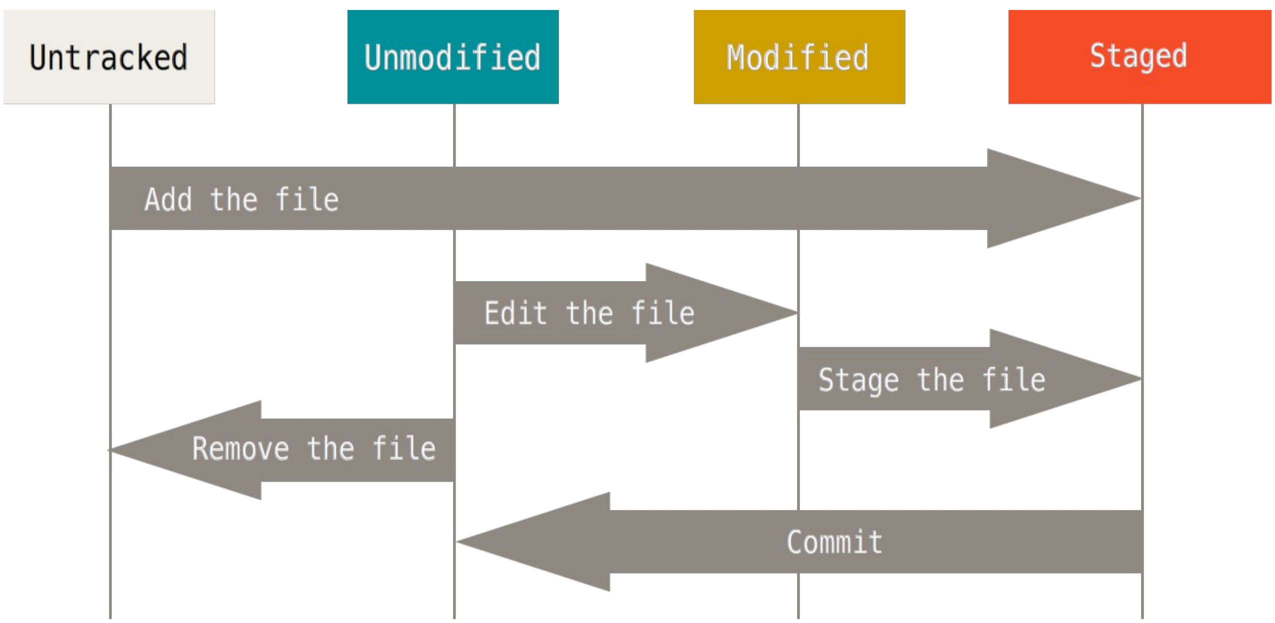

Figure 2-1. The lifecycle of the status of your files.

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

This means you have a clean working directory – in other words, there are no tracked and modified files. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server. For now, that branch is always "master", which is the default; you won't worry about it here. Git Branching will go over branches and references in detail.

Let's say you add a new file to your project, a simple README file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' &gt; README
$ git status
On branch master
Untracked files:
  (use "git add &lt;file&gt;..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new README file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let's start tracking the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the README file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    new file:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add (files)` – that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called "CONTRIBUTING.md" and then run your `git status` command again, you get something that looks like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

The "CONTRIBUTING.md" file appears under a section named "Changes not staged for commit" – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as "add this content to the next commit" rather than "add this file to the project". Let's run `git add` now to stage the "CONTRIBUTING.md" file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you're ready to commit. However, let's run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on. There are two columns to the output - the left hand column indicates that the file is staged and the right hand column indicates that it's modified. So for example in that output, the `README` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged. The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`.Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files that end with a tilde (`~`), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work.
- You can start patterns with a forward slash (`/`) to avoid recursivity.
- You can end patterns with a forward slash (`/`) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified regular expressions that shells use. An asterisk (`*`) matches zero or more characters; `[abc]` matches any character inside the brackets (in this case a, b, or c); a question mark (`?`) matches a single character; and brackets enclosing characters separated by a hyphen(`[0-9]`) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; `a/**/z` would match `a/z`, `a/b/z`, `a/b/c/z`, and so on.

Here is another example .gitignore file:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO
```

```
# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

TIP

GitHub maintains a fairly comprehensive list of good `.gitignore` file examples for dozens of projects and languages at https://github.com/github/gitignore if you want a starting point for your project.

Viewing Your Staged and Unstaged Changes

If the `git status` command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the `git diff` command. We'll cover `git diff` in more detail later, but you'll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although `git status` answers those questions very generally by listing the file names, `git diff` shows you the exact lines added and removed – the patch, as it were.

Let's say you edit and stage the `README` file again and then edit the `CONTRIBUTING.md` file without staging it. If you run your `git status` command, you once again see something like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
```

```
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

It's important to note that `git diff` by itself doesn't show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you've staged all of your changes, `git diff` will give you no output.

For another example, if you stage the `CONTRIBUTING.md` file and then edit it, you can use `git diff` to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

```
$ git add CONTRIBUTING.md
$ echo '# test line' &gt;&gt; CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Now you can use `git diff` to see what is still unstaged:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list]
(https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

and `git diff --cached` to see what you've staged so far (--staged and --cached are synonyms):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

NOTE

Git Diff in an External Tool

We will continue to use the `git diff` command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run `git difftool` instead of `git diff`, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run `git difftool --tool-help` to see what is available on your system.

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run `git add` on since you edited them – won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran `git status`, you saw that everything was staged, so you're ready to commit your changes. The simplest way to commit is to type `git commit`:

```
$ git commit
```

Doing so launches your editor of choice. (This is set by your shell's `$EDITOR` environment variable – usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor` command as you saw in [Getting Started](#)).

The editor displays the following text (this example is a Vim screen):

```
# 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:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing. (For an even more explicit reminder of what you've modified, you can pass the `-v` option to `git commit`. Doing so also puts the diff of your change in the editor so you can see exactly what changes you're committing.) When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the `commit` command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Now you've created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (`master`), what SHA-1 checksum the commit has (`463dc4f`), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Notice how you don't have to run `git add` on the "CONTRIBUTING.md" file in this case before you commit.

Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.
If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is, *unstaged*) area of your `git status` output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm &lt;file&gt;..." to update what will be committed)
  (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)

        deleted:    PROJECTS.md

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

Then, if you run `git rm`, it stages the file's removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    deleted:    PROJECTS.md
```

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

```
$ git rm --cached README
```

You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as:

```
$ git rm log/\*.log
```

Note the backslash (`\`) in front of the `*`. This is necessary because Git does its own filename expansion in addition to your shell's filename expansion. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like this:

```
$ git rm \*~
```

This command removes all files that end with `~`.

Moving Files

Unlike many other VCS systems, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact – we'll deal with detecting file movement a bit later.

Thus it's a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD &lt;file&gt;..." to unstage)

    renamed:    README.md -&gt; README
```

However, this is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the `mv` command. The only real difference is that `mv` is one command instead of three – it's a convenience function. More important, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

2.3 Git Basics - Viewing the Commit History

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

These examples use a very simple project called "simplegit". To get the project, run

```
$ git clone https://github.com/schacon/simplegit-progit
```

When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

A huge number and variety of options to the `git log` command are available to show you exactly what you're looking for. Here, we'll show you some of the most popular.

One of the more helpful options is `-p`, which shows the difference introduced in each commit. You can also use `-2`, which limits the output to only the last two entries:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
```

```
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with `git log`. For example, if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

As you can see, the `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is `--pretty`. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The `oneline` option prints each commit on a single line, which is useful if you're looking at a lot of commits. In addition, the `short`, `full`, and `fuller` options show the output in roughly the same format but with less or more information, respectively:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

The most interesting option is `format`, which allows you to specify your own log output format.

This is especially useful when you're generating output for machine parsing – because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Table 2-1 lists some of the more useful options that format takes.

Table 2-1. Useful options for `git log --pretty=format`

Option	Description of Output
`%H`	Commit hash
`%h`	Abbreviated commit hash
`%T`	Tree hash
`%t`	Abbreviated tree hash
`%P`	Parent hashes
`%p`	Abbreviated parent hashes
`%an`	Author name
`%ae`	Author email
`%ad`	Author date (format respects the --date=option)
`%ar`	Author date, relative
`%cn`	Committer name
`%ce`	Committer email

Table 2-1. Useful options for `git log --pretty=format`

Option	Description of Output
`%cd`	Committer date
`%cr`	Committer date, relative
`%s`	Subject

You may be wondering what the difference is between *author* and *committer*. The author is the person who originally wrote the work, whereas the committer is the person who last applied the work. So, if you send in a patch to a project and one of the core members applies the patch, both of you get credit – you as the author, and the core member as the committer. We'll cover this distinction a bit more inDistributed Git.

The oneline and format options are particularly useful with another `log` option called `--graph`. This option adds a nice little ASCII graph showing your branch and merge history:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

This type of output will become more interesting as we go through branching and merging in the next chapter.

Those are only some simple output-formatting options to `git log` – there are many more. Table 2-2lists the options we've covered so far, as well as some other common formatting options that may be useful, along with how they change the output of the log command.

Table 2-2. Common options to `git log`

Option	Description
`-p`	Show the patch introduced with each commit.

Table 2-2. Common options to `git log`

Option	Description
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the --stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).

Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options – that is, options that let you show only a subset of commits. You've seen one such option already – the `-2` option, which show only the last two commits. In fact, you can do `-<n>`, where `n` is any integer to show the last `n` commits. In reality, you're unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

However, the time-limiting options such as `--since` and `--until` are very useful. For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

This command works with lots of formats – you can specify a specific date like `"2008-01-15"`, or a relative date such as `"2 years 1 day 3 minutes ago"`.

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages. (Note that if you want to specify both author and grep options, you have to add `--all-match` or the command will match commits with either.)

Another really helpful filter is the `-S` option which takes a string and only shows the commits that introduced a change to the code that added or removed that string. For instance, if you wanted to find the last commit that added or removed a reference to a specific function, you could call:

```
$ git log -Sfunction_name
```

The last really useful option to pass to `git log` as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (`--`) to separate the paths from the options.

In Table 2-3 we'll list these and a few other common options for your reference.

Table 2-3. Options to limit the output of `git log`

Option	Description
`-(n)`	Show only the last n commits
`--since`, `--after`	Limit the commits to those made after the specified date.
`--until`, `--before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string
`-S`	Only show commits adding or removing code matching the string

For example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano and were not merged in the month of October 2008, you can run something like this:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
```

```
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Of the nearly 40,000 commits in the Git source code history, this command shows the 6 that match those criteria.