Mike McQuaid

Git

IN PRACTICE

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Git in Practice
Version 1

For more information on this and other Manning titles go to
www.manning.com

Welcome

Hi,

Thanks for purchasing the MEAP of *Git In Practice*. I hope that what you'll get access to will be of immediate use to you and, with your help, the final book will be great!

Git is a powerful distributed version control system but it can sometimes be difficult to understand what is going on behind the scenes or how you've found yourself in a particular state. In Git In Practice by the end of the book you should understand:

- The internals and confusing jargon Git will sometimes expose
- A wide array of commands for interacting with Git repositories
- Workflows for configuring, using Git and GitHub effectively for teams, products and open-source project

What you'll be getting immediately is Part 1 of the book with a chapter from Part 2. Next month, you'll get a chapter from 3.

Part 1 (Chapters 1-3) will fly through the basics of distributed version control using Git while teaching you the underlying concepts that are often misunderstood or omitted in beginners guides. You may know how to use Git already but I'd encourage you to persevere with this anyway; it's setting a good foundation which the rest of the book will build upon.

Chapter 4 (from Part 2) will show you various ways of interacting with the file system using Git in a problem/solution format.

Chapter 10 (from Part 3) discusses two approaches for team branch workflows using Git; rebasing and merging. These are compared by contrasting the approaches taken by two successful open-source projects: the CMake build system and the Homebrew OS X package manager.

Part 2 will continue to introduce more Git commands in a problem/solution format and Part 3 will introduce more high-level workflows for managing projects and teams when using Git.

Please let me know your thoughts on what's been written so far and what you'd like to see in the rest of the book. Your feedback will be invaluable in improving Git In Practice.

Thanks again for your interest and for purchasing the MEAP!

Mike McQuaid

brief contents

1 *Introduction to Distributed Version Control*

In this chapter you will about distributed control systems such as Git by covering the following topics:

- Why programmers use *version control systems* to keep track of changes to files over time.
- Adding files to a version control system (*committing*), seeing the differences between versions of files (*diffing*), creating independent tracks of changes (*branching*) and bringing the changes from one track back into another (*merging*).
- Differences between version control systems that store and share changes on a central server (*centralized version control systems*) and those that keep all changes on a local machine and can send and receive them from other machines (*distributed version control systems*).

You may not be familiar with version control concepts or why version control systems are useful for managing changes to text. Let's start off by asking why you should use version control.

1.1 Why use version control?

A common problem when dealing with information stored on a computer is handling *changes*. For example, after adding, modifying or deleting text you may want to undo that action (and perhaps redo it later). At the simplest level this might be done by clicking *undo* in a text editor (which reverts a previous action); after new words are added it may be necessary to undo these changes by pressing undo repeatedly until you return to the desired previous state.

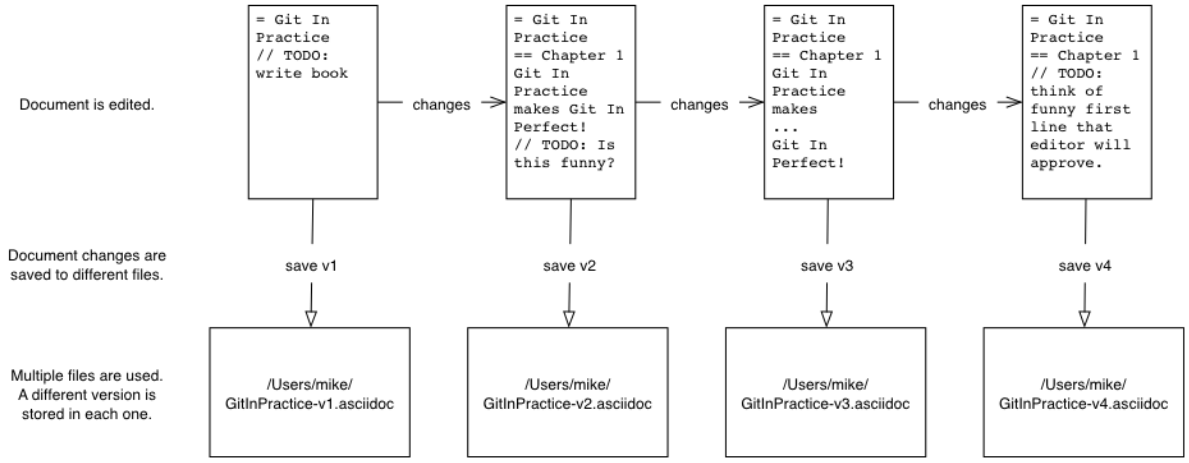

Figure 1.1 Versioning with multiple files

A naive method for handling multiple file versions is often simply creating duplicate files with differing filenames and contents (`Important Document V4 FINAL FINAL.doc` may sound sadly familiar). An example of this approach can be seen in Figure 1.1.

At a more advanced level you may be sharing a document with other people and, rather than just undoing and redoing changes, wish to know who made a change, why they made it, when they made it, what the change was and perhaps even store multiple versions of the document in parallel. A *version control system* (such as Git) allows all these operations and more.

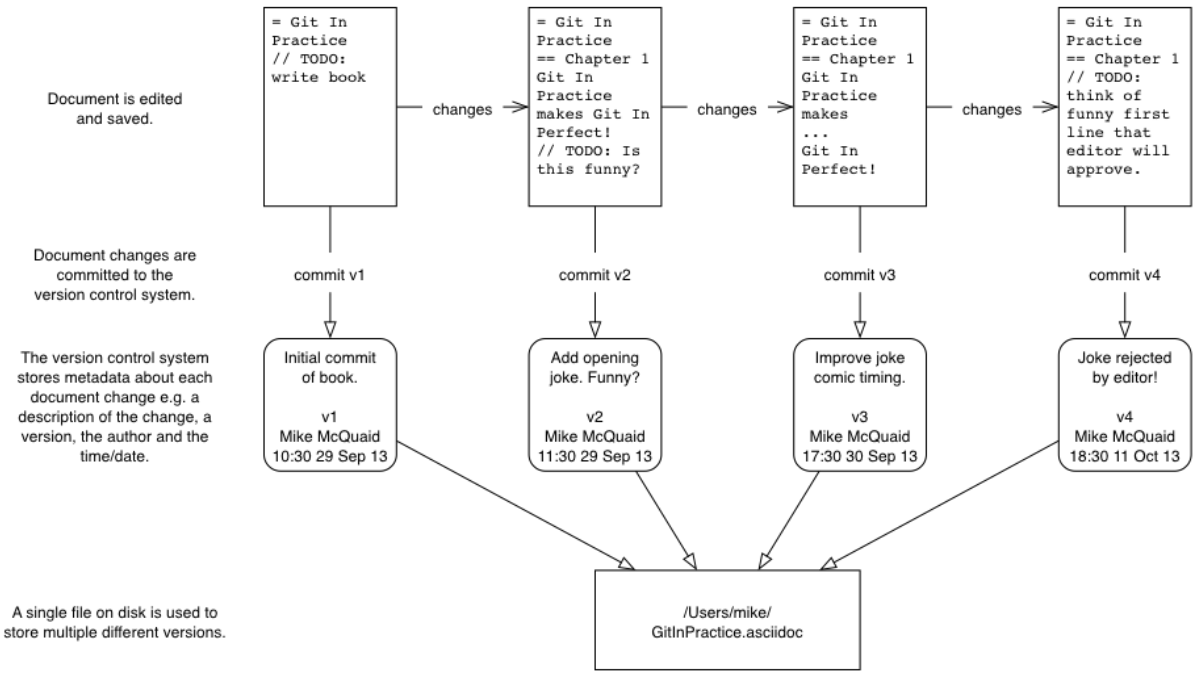

Figure 1.2 Versioning with a version control system

In a version control system instead of just saving a document after your changes have been made you would *commit* it. This involves a save-like operation commanding the version control system to store this particular version and specifying a message stating the reason for their change or what it accomplishes. When another commit is made then the previous version would remain in *history* where its changes can be examined at a later time. Version control systems can therefore solve the problem of reviewing and retrieving previous changes and allow single files to be used rather than duplicated. This workflow can be seen in Figure 1.2.

When editing a file in a version control system you will always edit/save/commit the same file on disk. It will not move location either manually or automatically (unless you wish to rename it, of course). When you wish to access previous versions of the file you can either view them through the version control system or restore the file on disk to a previous version. This allows you to see exactly what may have changed between versions. When using multiple files you would have to manually compare each of the files to see differences and keep track of multiple files on your disk.

1.1.1 Version control workflow

Version control systems work by maintaining a list of changes to files over time. Each time a file is modified and committed both the newly committed version of the file and the previous version of the file are stored in the *repository*; a centralized location where the version control system stores files for a particular project. Each commit corresponds to a particular version and stores references to the previously made commit, a description of the changes, time it was made, who made it and the contents of the files at this point. The files' state from a commit can be compared to a previous version and the difference between the versions' files (known as *diffs*) can be queried.

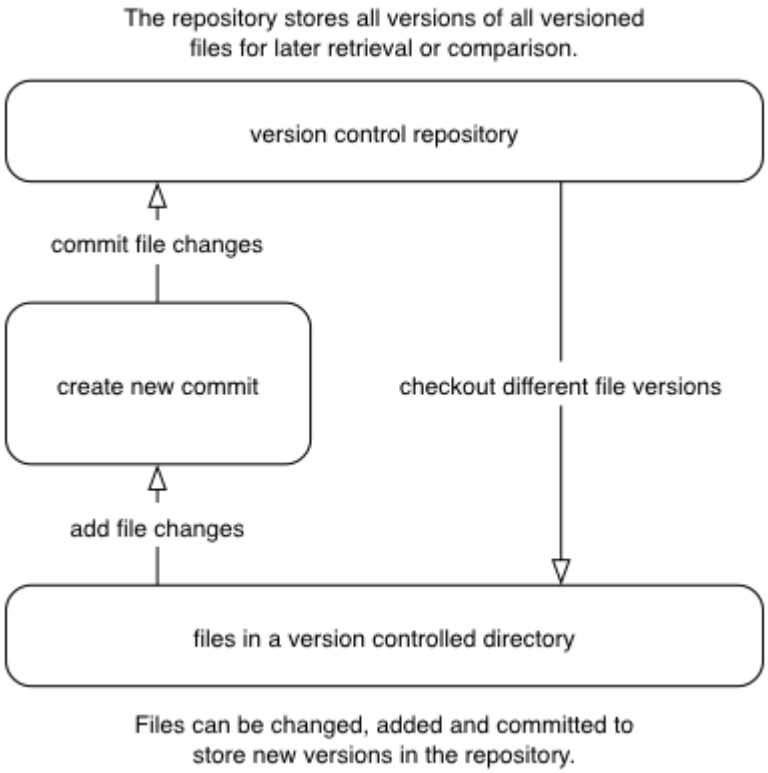

Figure 1.3 Git add/commit/checkout workflow

Figure 1.3 shows the workflow you will use when using a version control system. After adding new changes to versioned files you will create new commits containing these changes and commit the changes to the repository. At a later point you can checkout different versions of files. This allows you to have confidence that, no matter what you may add, modify or delete, all committed versions of your files will remain in the version control system if you need to check their contents later.

1.1.2 Version control for programmer collaboration

Programmers spend most of their jobs (and sometimes their lives) editing text. This text is typically source code which will be interpreted by a computer to perform some task (hopefully better than a human) but could also be software configuration files, documentation, emails or even a dreaded TPS report. As they typically work on independent units of work while in larger teams and can be distributed by time or geography it's important that they communicate explicitly to other programmers why a particular change was made. Additionally programmers inevitably write software which contains *errors* (or *bugs*, if you're feeling kind, or *errata* if you're an electronic engineer). When trying to work out why a bug occurred (and hopefully fix it) it's useful to see what changes were made, by whom and for what reason. Often programmers will need to fix bugs in sections of code they did not create so being able to record and recall the intent of the original author at a later point can help understand what faulty assumptions may have caused a bug. Combine these reasons with the sheer numbers of files programmers typically work with and it should become clear why most programming projects use version control systems to manage their source code.

When creating computer software it's also common to release new *versions* of a product. New versions are generally released when bugs are fixed and/or when new features have been created. However, a team may be half way through developing a new feature but have fixed a bug that they need to provide to users immediately (before the new feature has been completed). In this case two *branches* could be used to split the history and allow a version containing the bug fix to be immediately released and ensure that none of the changes made while creating a new feature end up in that release. Instead these changes would be made on an independent branch of the history to allow continuous work on the new feature rather than waiting for the bug fix's release to be made. This branch could be later *merged* into the main branch which would include all the changes into the main branch.

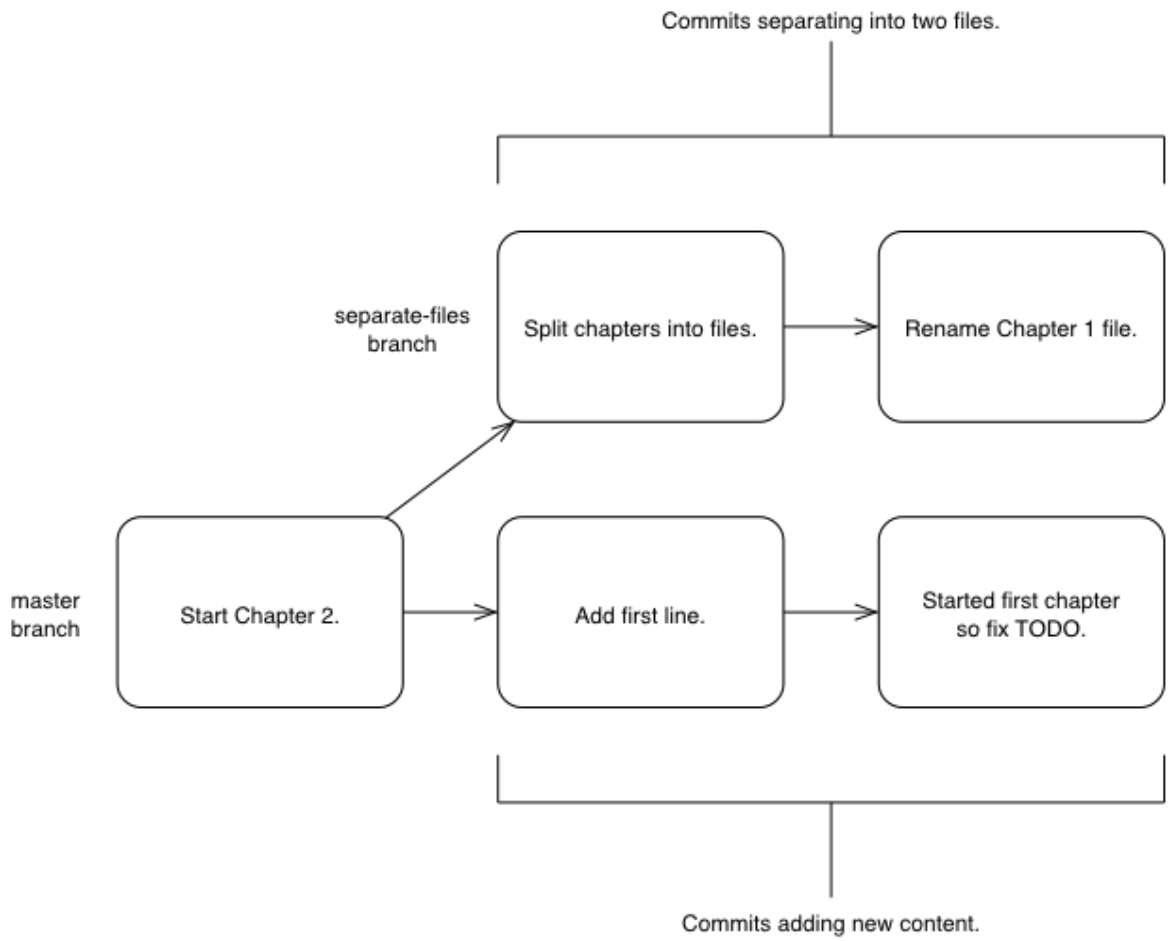

Figure 1.4 Committing on multiple branches

In Figure 1.4 you can see a simple example of using multiple branches when writing a book. We'll cover this in more detail in Section 1.4.1 but you should already be able to see how they allow multiple tracks of development in a project.

Now you've learnt why programmers use version control let's see why they use Git specifically.

1.2 Why do programmers use Git?

Git was created by a programmer to be used by programmers. Linus Torvalds, the creator of Git and the Linux kernel, started in 2008 with the goal of having a distributed, open-source, high-performance and hard to corrupt version control system for the Linux kernel project to use. Within a week Git was self-hosting (meaning Git's source code was hosted inside a Git repository) and within two and a half months the version 2.6.12 of the Linux kernel was made using Git.

From its initial creation for the Linux kernel Git is now used by all sizes of companies, many open source projects and large "social coding" Git hosting sites such as GitHub or Bitbucket.

NOTE **Why do people prefer Git?**

Git is my preferred method of software source code control. I also use it for versioning plain text files such as the text for this book. Git has many strengths over other methods of source control. Git stores all of the history of a repository, branches and commits locally. This means adding new versions or querying the history of a repository doesn't require a network connection. Git's history log viewing and branch creation is near-instant compared to e.g. Subversion's which is sluggish even on a fast network connection. As Git stores changes locally you are not constrained by the work of others when working in a team. For example, merge conflicts are solved at the time by the person doing the merge so you can continue your edit/commit workflow without interruptions. In Git you can modify the history of branches and even entire repositories. It's often useful to be able to make lots of small commits which are later turned into a single commit or make commit messages contain more information after the fact. Despite this flexibility with Git anything that is committed has a unique reference that survives rewriting or changes for at least 30 days. This means it's very hard to accidentally lose work.

Git's main downsides are that the command-line application's interface is often counterintuitive; it makes frequent use of jargon that can only be adequately explained by understanding Git's internals. Additionally Git's official documentation can be hard to follow; it also uses jargon and has to detail the large number of options for Git's commands. To the credit of the Git community both the UI and documentation around Git have improved hugely over the years. This book will help you understand Git's jargon and the Git's internal operations; this should help you to understand why Git does what it does when you run the various Git commands.

Despite these downsides the strengths of Git have proved too strong for many software projects to resist. Google, Microsoft, Twitter, LinkedIn and Netflix all use Git as well as open-source projects such as the Linux kernel (the first Git user), Perl, PostgreSQL, Android, Ruby on Rails, Qt, GNOME, KDE, Eclipse and X.org.

Many of the above projects and many users of Git have also been introduced to Git and use it regularly through a Git hosting provider. My favorite is GitHub but there are alternatives such as Gitorious, Bitbucket, SourceForge, Google Code and others.

Let's learn more how Git actually manages changes. Git's changes are known as *commits*.

1.3 Committing: recording changes to code

A *commit* is a collection of changes to one or more files in a version control system. Each commit contains a message entered by the author, details of the author of the commit, a unique commit reference (the format of which varies between version control systems but in Git looks like `86bb0d659a39c98808439fadb8dbd594bec0004d`), a pointer to the preceding commit (known as the *parent commit*), the date the commit was created and a pointer to the contents of files when the commit was made. The file contents are typically displayed as the *diff* (the differences between the files before and the files after the commit).

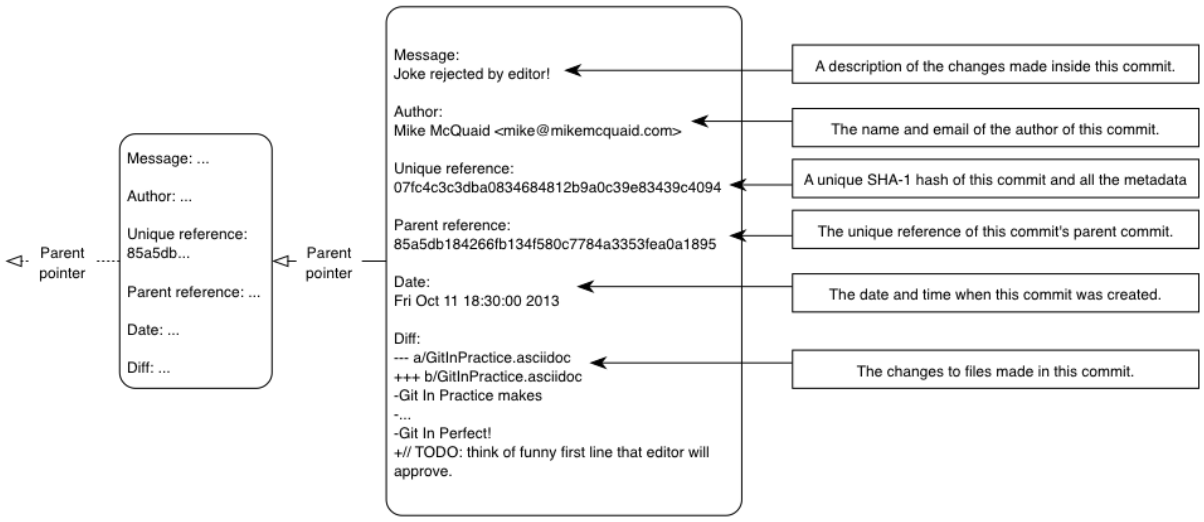

Figure 1.5 A typical commit broken down into its parts

As you may have noticed Figure 1.5 uses arrows pointing from commits to their previous commit. The reason for this is that commits contain a pointer to the *parent commit* and not the other way round; when a commit is made it has no idea what the next commit will be yet.

A *commit* is made up of the changes to one or more files on disk. The typical workflow is that you will change the contents of files inside a folder on disk which is managed by Git and, after making all necessary changes, review the *diffs* and add them to a new commit. Often all the *diffs* made will turn into one commit and then the cycle will repeat. Sometimes, however, it is desirable to pick only some changed files (or even some changed lines within files) to include in a commit and leave the other changes for adding in a future commit. This is often desirable

because commits should be the smallest possible units of work to make them easier to understand.

> **NOTE** **Why are small commits better?**
> Commits should be kept as small as possible. This allows their message to describe a single change rather than multiple changes that are unrelated but were worked on at the same time. Small commits keep the history readable; it's easier when looking at a small commit in future to understand exactly why the change was made. If a small commit was later found to be undesirable it can be easily reverted. This is much more difficult if many unrelated changes are clumped together into a single commit and you wish to revert a single change.

1.3.1 Commit storage in Git

Git is a version control system built on top of an *object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*. The repository is the local collection of all the files related to a particular Git version control system and is stored in a `.git` folder in the root of the project. If you were to explore under here (as we will do in Chapter 2) you would find objects, pointers/references to objects and configuration files.

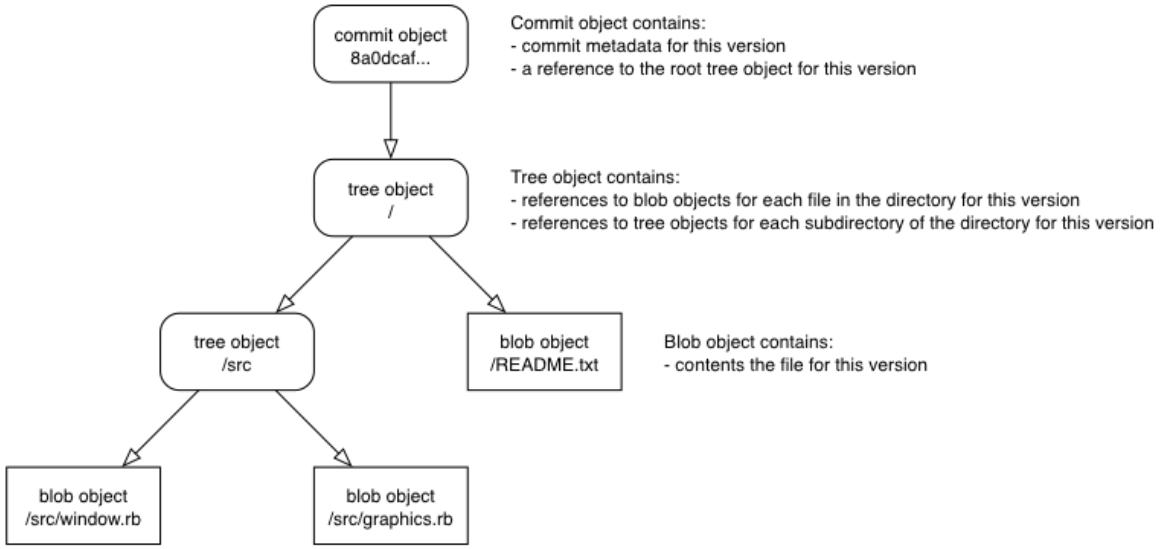

Figure 1.6 commit, blob and tree objects

In Figure 1.6 you can see the main Git objects we're concerned with: *commits*, *blobs* and *trees*. There is also a *tag* object but we'll leave tags until Section 1.4. We've already seen that commits store metadata and referenced file contents. The

file contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subfolders. A blob object stores the contents of a particular version of a particular single file in the Git repository.

NOTE **Should objects being interacted with directly?**
When using Git you should never need to interact with objects or object files directly. The terminology of *blobs* and *trees* are not used regularly in Git or in this book but it's useful to remember what these are so you can build a conceptual understanding of what Git is doing internally. When things go well this should be unnecessary but when we start to delve into more advanced Git functionality or Git spits out a baffling error message then remembering *blobs* and *trees* may help you work out what has happened.

Now we've peeked behind Git's abstraction to see how it stores things internally let's return to something very practical: how versions of code change over time.

1.4 History: how code changes over time

The *history* of a version control system is the complete list of all changes made since the repository was created and the initial commit was made. The history also contains the references to any *branches*, *tags* (a way of annotating a particular commit with, for example, a version) and *merges* made within the repository.

Without history version control would be a simple mechanism for file storage. History allows us to analyze the state of a repository at any specific date and time and recall the contents of every file, the person who changed the files, when they changed the files and (if a good commit message has been written) why they changed them.

When you are using version control you will find yourself regularly checking the history; sometimes to remind yourself of your own work, sometimes to see why other changes were made in the past and sometimes reading new changes than have been made by others. In different situations different pieces of data will be interesting but all pieces of data will always be available for every commit.

As you may have got a sense of already: how useful the history is relies very much on the quality of the data entered into it. If I made a commit once per year with huge numbers of changes and a commit message of "fixes" then it would be fairly hard to use the history effectively. Ideally commits are small and

well-described; follow these two rules and having a complete history becomes a very useful tool.

1.4.1 Commits point to their parent commits

Every commit points to its *parent commit*. The parent commit in a linear, branch-less history will be the one that immediately preceded it. The only commit that lacks a parent commit is the *initial commit*; the first commit in the repository. By following the parent commit, its parent, its parent and so on you will always be able to get back from the current commit to the initial commit. You can see an example of parent commit pointers in Figure 1.7.

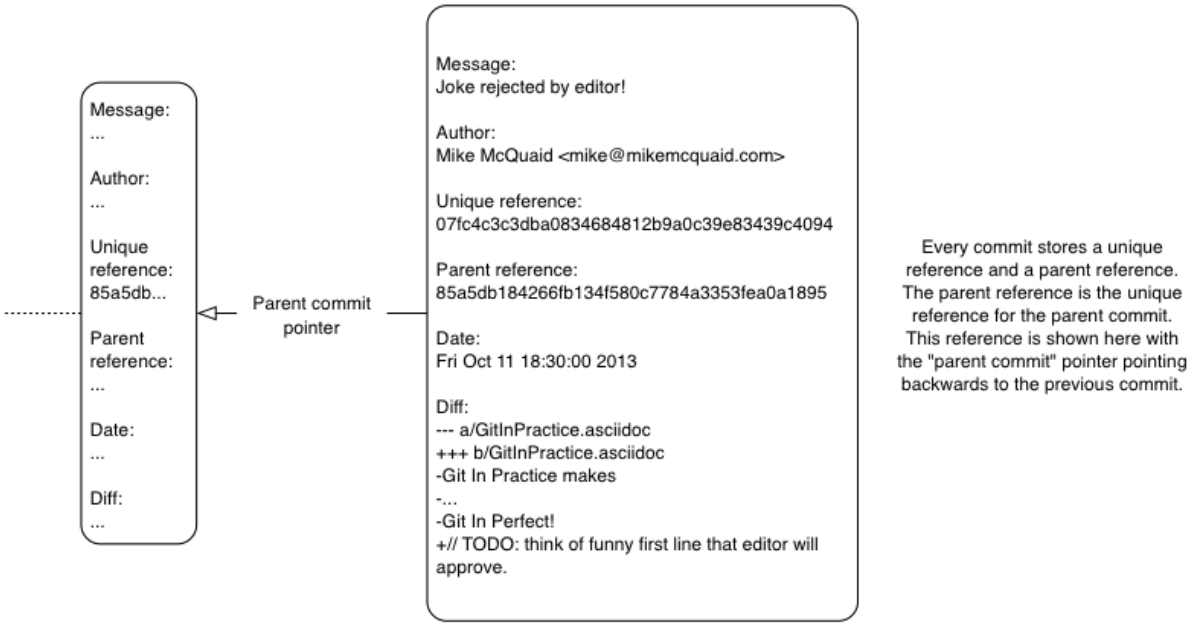

Figure 1.7 Parent commit pointers

1.4.2 Rewriting history

Git is unusual compared to many other version control systems in that it allows history to be rewritten. This may seen surprising or worrying; after all did I not just tell you that the history contains the entire list of changes to the project over time? Surely it is dangerous to modify this? The answer to this question is: sometimes. In a history book you may hear about the beginning and end of various historical transitions but not every detail of what occurred in between. Similarly sometimes you may want to highlight only broader changes to files in a version control system over a period of time rather than sharing ever single change that was made in reaching the final state.

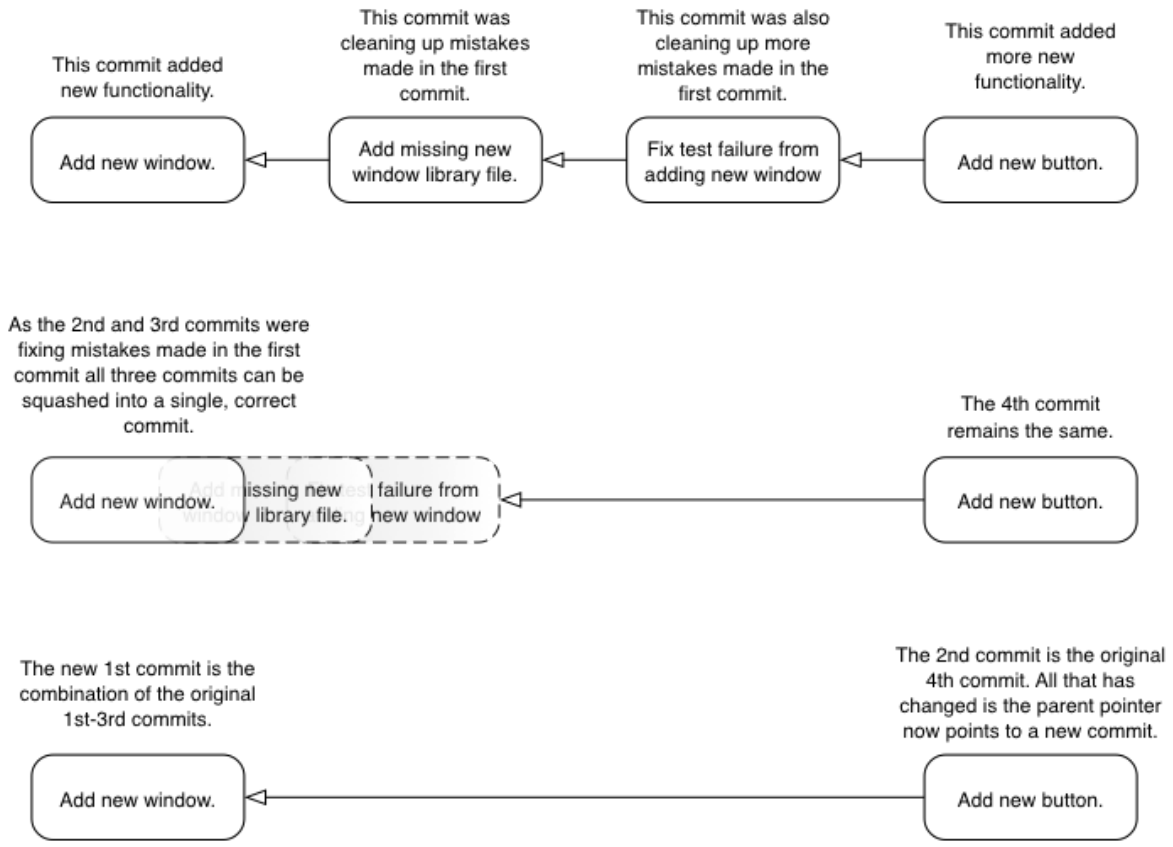

Figure 1.8 Squashing multiple commits into a single commit

In Figure 1.8 you see a fairly common use-case for rewriting history with Git. If you were working on some window code all morning and wanted your coworkers to see it later (or just include it in the project) then there's no need for everyone to see the mistakes you made along the way; why damage your good reputation unnecessarily? In Figure 1.8 the commits are being *squashed* together so instead of three commits and the latter two fixing mistakes in the first commit we have squashed these together to create a single commit for the window feature. We'd only rewrite history like this if working on a separate branch that hadn't had other work from other people relying on it yet as it has changed some parent commits (so, without intervention, other people's commits may point to commits that no longer exist). Don't worry too much about rebasing or squashing work for now; just this as a situation where you may want to rewrite history. In Chapter 7 we'll cover cases where history rewriting is useful such as rewriting an entire repository to change an email address or removing confidential information before making the history public.

What we're generally interested in when reading the history (and why we clean it up) is ensuring the changes between commits are relevant (for example don't

make changes only to revert then immediately in the next commit five minutes later), minimal and readable. These changes are known as *diffs*.

1.5 Diffs: the differences between commits

A *diff* (also known as a *change* or *delta*) is the difference between two commits. In a version control system you can typically request a diff between any two commits, branches or tags. It's often useful to be able to request the difference between two parts of the history for analysis. For example, if an unexpected part of the software has recently started misbehaving you may go back into the history to verify that it previously worked. If it did work previously then you may want to examine the diff between the the code in the different parts of the history to see what has changed. The various ways of displaying diffs in version control typically allow you to narrow them down per-file, folder and even committer.

1.5.1 Default diff format

Diffs are typically shown by version control systems in a format that is known as a *unified diff*.

Listing 1.1 Unified diff from Git

```
diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index 7bd3fb8..7230cbf 100644
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
@@ -1,5 +1,3 @@
 = Git In Practice
 == Chapter 1
-Git In Practice makes
-...
-Git In Perfect!
+// TODO: think of funny first line that editor will approve.
```

1. diff command
2. staging area SHA-1
3. old file version
4. new file version
5. file changes range
6. unchanged line
7. deleted line
8. inserted line

Listing 1.1 shows a change to a `GitInPractice.asciidoc` deleting multiple lines of text and replacing them with a comment.

The "diff command (1)" shows an example `diff` command that may have been used to output these changes (although in reality this is done by Git internally). The "staging area SHA-1 (2)" shows the changes that were made to the `index staging area` by this commit; in short the changes to the actual contents of files in this commit. The staging area will be explained more in Chapter 2. The "old file version (3)" and "new file version (4)" show virtual file names relating to the diff command; as if instead of comparing multiple versions in the

version control repository multiple versions of the files in different folders (`a` and `b` were compared). The "file changes range (5)" is used by the `diff` tool to find the locations in the file this diff refers to. In this example you can see the entire file and all the changes made to it. If this were a file with thousands of lines but only five lines were changed then the diff would only show the five lines that were changed with some surrounding context. These range markers would allow a `diff` tool to find what lines should be changed. Lines that are " unchanged (6)" are displayed as-is, "deleted (7)" lines are prefixed with a `-` and "inserted (8)" lines are prefixed with a `+`.

NOTE **How do diffs show changed lines?**
changed lines are displayed in a unified diff as a deletion of the previous line and insertion of the new one (even if the change is only a single character).

Diffs are used throughout version control systems to indicate changes to files; for example when navigating through history or viewing what you are about to commit. It's important to grasp the format as it will be used throughout this book and when using Git.

1.5.2 Different diff formats

Sometimes it is desirable to display diffs in slightly differing formats. Two common alternatives to a typical unified diff are a *diffstat* and *word diff*.

Listing 1.2 Diffstat from Git

```
GitInPractice.asciidoc | 4 +---
1 file changed, 1 insertion(+), 3 deletions(-)
```

❶ one file's changes
❷ all files' changes

Listing 1.2 is a diffstat for the same changes as the unified diff in Listing 1.1. Rather than showing the breakdown of exactly what has changed it indicates what files have changed and a brief overview of how many lines were involved in the changes. This can be useful when getting a quick overview of what has changed without needing all the detail of a normal unified diff.

The "one file's changes (1)" shows the filename that has been changed, the number of lines changed in that file and `+`/`-` characters summarizing the overall

changes to the file. If multiple files were changed this would show there would be multiple filenames listed and each would have the lines changed in that file and +/- characters.

The "all files' changes (2)" shows a summary of totals of the number of files changes and lines inserted/deleted across all files.

Listing 1.3 Word diff from Git

```
diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index 7bd3fb8..7230cbf 100644
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
@@ -1,5 +1,3 @@
= Git In Practice
== Chapter 1
[-Git In Practice makes-]
[-...-]
[-Git In Perfect!-]{+// TODO: think of funny first line that editor
will approve.+}
```

1 deleted line

2 modified line

A word diff is similar to a unified diff but shows modifications per-word rather than per-line. Listing 1.3 shows that most of the sentence remained the same except for a few changed words. This is particularly useful when viewing changes that are not to code but plain text; in README files we probably care more about individual word choices than knowing that an entire line has changed and the special characters (`[-]{+}`) are not used as often in prose than in code.

The "deleted line" is surrounded by `[-]` shows a line that was completed removed. The "modified line" has some characters that were removed surrounded by `[-]` and some lines that were added surrounded by `{+}`.

1.6 Branches: working on multiple versions of code in parallel

When committing to a version control system the history continues linearly; what was the most recent commit becomes the parent commit for the new commit. This parenting continues back to the initial commit in the repository. You can see an example of this in Figure 1.9.

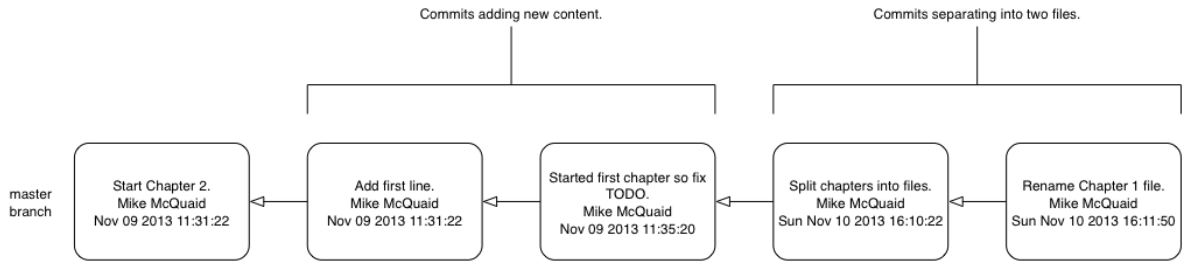

Figure 1.9 Committing without using branches

As discussed previously, sometimes this linear approach is not enough for software projects. Sometimes you may need to make new commits which are not yet ready for public consumption. Enter *branches*.

Branching allows two independent tracks through history to be created and committed to without either modifying the other. Programmers can happily commit to their independent branch without the fear of disrupting the work of another branch. This means that they can, for example, commit broken or incomplete features rather than having to wait for others to be ready for their commits. It also means they can be isolated from changes made by others until they are ready to integrate them into their branch. Figure 1.10 shows the same commits as Figure 1.9 if they were split between two branches instead for isolation.

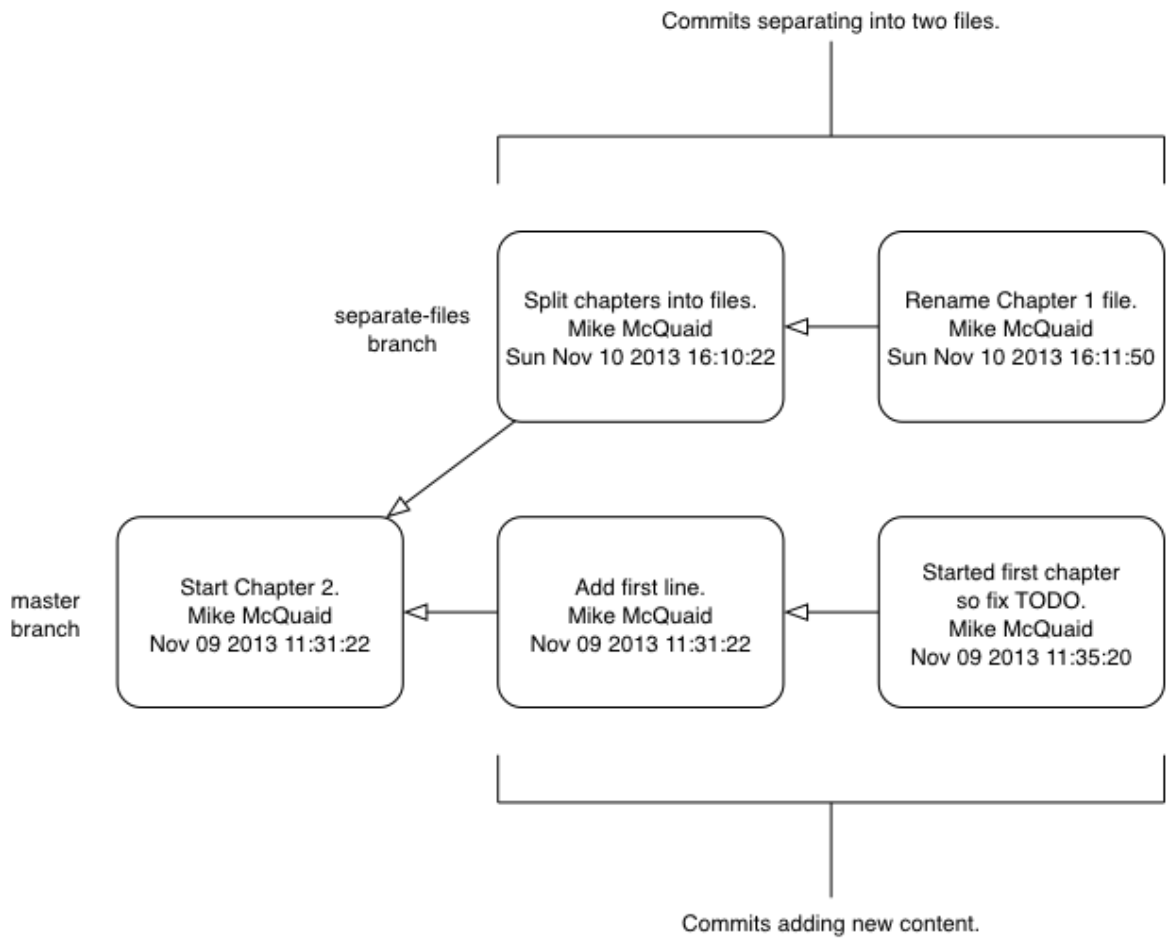

Figure 1.10 Committing to multiple branches

When a branch is created and new commits are made that branch advances forward to include the new commits. In Git a branch is actually no more than a pointer to a particular commit. The branch is pointed to a new commit when a new commit is made on that branch. A *tag* is quite similar to a branch but points to a single commit and remains pointing to the same commit even when new commits are made. Typically tags are used for annotating commits; for example, when you release version 1.0 of your software you may tag the commit used to built the 1.0 release with a "1.0" tag. This means you can come back to it in future, rebuild that release or check how certain things worked without fear that it will be somehow changed automatically.

1.6.1 Using branching

Branching allows two independent tracks of development to occur at once. In Figure 1.10, the `separate-files branch` was used to separate the content from a single file and split it into two new files. This allowed refactoring of the book structure to be done in the `separate-files` branch while the default branch (known as `master` in Git) could be used to create more content. In version control systems like Git where creating a branch is a quick, local operation branches may be used for every independent change.

Some programmers will create new branches whenever they work on a new bug fix or feature and then integrate these branches at a later point; perhaps after requesting review of their changes from others. This means even for programmers working without a team it can be useful to have multiple branches in use at any one point. For example, you may be working on a new feature but realize that a critical error in your application needs fixed immediately. You could quickly create a new branch based off the version used by customers, fix the error and switch branch back to the branch you had been committing the new feature to.

1.7 Merging: bringing the changes from one branch into another

At some point we have a branch that we're done with and we want to bring all the commits made on it into another branch. This process is known as a `merge`.

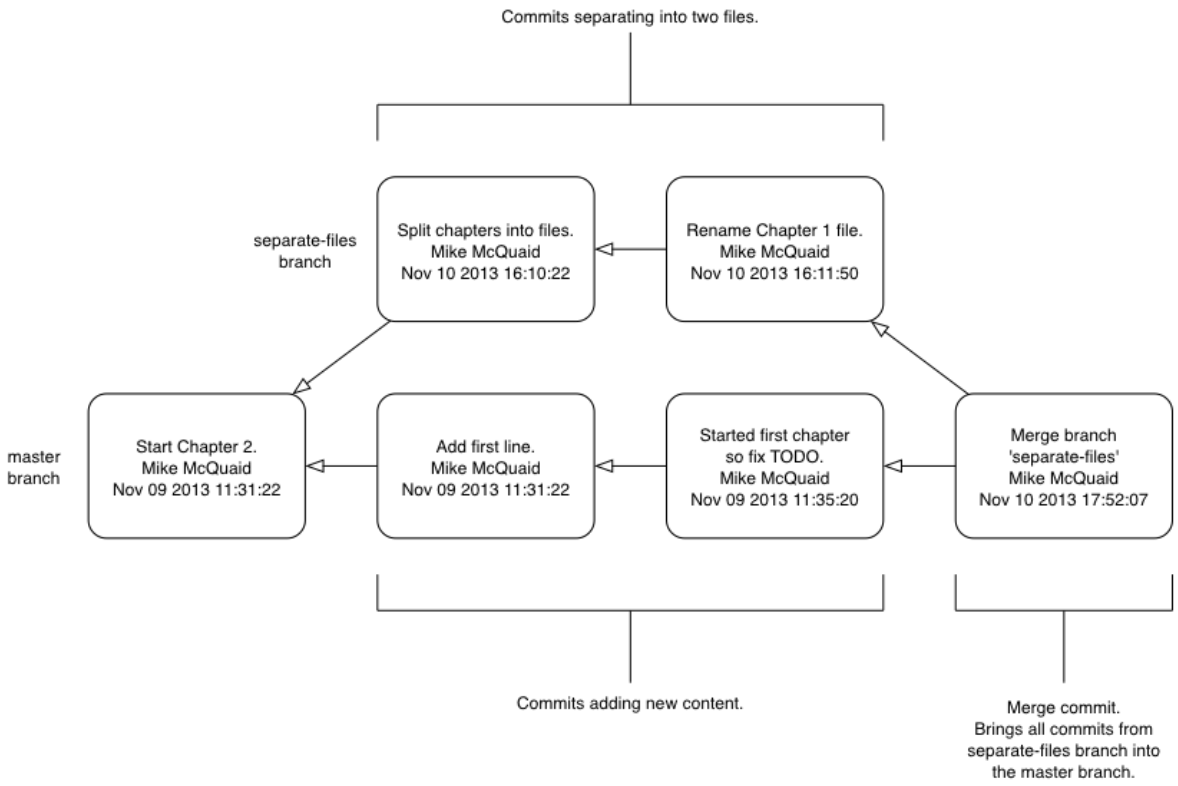

Figure 1.11 Merging a branch into master

When a merge is requested all the commits from another branch are pulled into the current branch. Those commits then become part of the history of the branch. Please note from Figure 1.11 the commit in which the merge is made has two parents commits rather than one; it is joining together two separate paths through the history back into a single one. After a merge you may decide to keep the existing branch around to add more commits to it and perhaps merge again at a later point (only the new commits will need to be merged next time). Alternatively, you may delete the branch and make future commits on the Git's default `master` branch and create another branch when needed in the future.

1.7.1 Merge conflicts

So far merges may have sounded too good to be true; you can work on multiple things in progress and combine them at any later point in any order. Not so fast my merge-happy friend; I haven't told you about merge conflicts yet.

A *merge conflict* occurs when both branches involved in the merge have changed the same file (or the same part of the same file, depending on how smart your version control system is). The version control system will try and automatically resolve these conflicts but sometimes is unable to do so without human intervention. Git can typically merge without conflicts as long as the

changes were not too near each other in the same file. If the version control system fails to perform the merge without human intervention it produces a merge conflict.

Listing 1.4 Merge conflict resolution with Git

```
= Git In Practice  (1)
<<<<<<< HEAD  (2)
== Chapter 1  (3)
It is a truth universally acknowledged, that a single person in
possession of good source code, must be in want of a version control
system.

== Chapter 2
// TODO: write second chapter.
=======  (4)
>>>>>>> separate-files  (5)
```

(1) unchanged line
(2) previous changes marker
(3) previous line version
(4) changes separator
(5) new changes marker

When a merge conflict occurs the version control system will go through any files that have conflicts and insert something similar to the above markers. These markers indicate the versions of the file on each branch.

The "unchanged line (1)" is one that, like in the Listing 1.1 unified diff, is one that is provided only for context in this example. The "previous changes marker (2)" shows the beginning of where new the lines from the previous commit (referenced by `HEAD` here; `HEAD` will be explained more in Chapter 2). The "previous line version (3)" shows a line that was from the previous commit. The "changes separator (4)" separates the previous and new changes. The "new changes (5)" marker shows the end of the new changes and the name of the branch that has been merged in; `separate-files` in this case.

NOTE **How can conflict markers be found quickly?**
When searching a large file for the merge conflict markers you should enter <<<< into your text editor's find tool to quickly locate them.

The person performing the merge will need to manually edit the file to produce the correctly merged output, save it and mark the commit as resolved. Sometimes the correct output will picking a single side of the markers and sometimes it will be a combination of the two. In cases where other files have been edited (like this example) it may also involve putting some of these lines into other files.

When conflicts have been resolved a *merge commit* can be made. This will store the two parent commits and the conflicts that were resolved so they can be inspected in the future. Unfortunately sometimes people will pick the wrong option or merge incorrectly so it's good to be able to later see what conflicts they had to resolve.

1.7.2 Rebasing

A *rebase* is a method of history rewriting in Git that is similar to a merge. A rebase involves changing the parent of a commit to point to another.

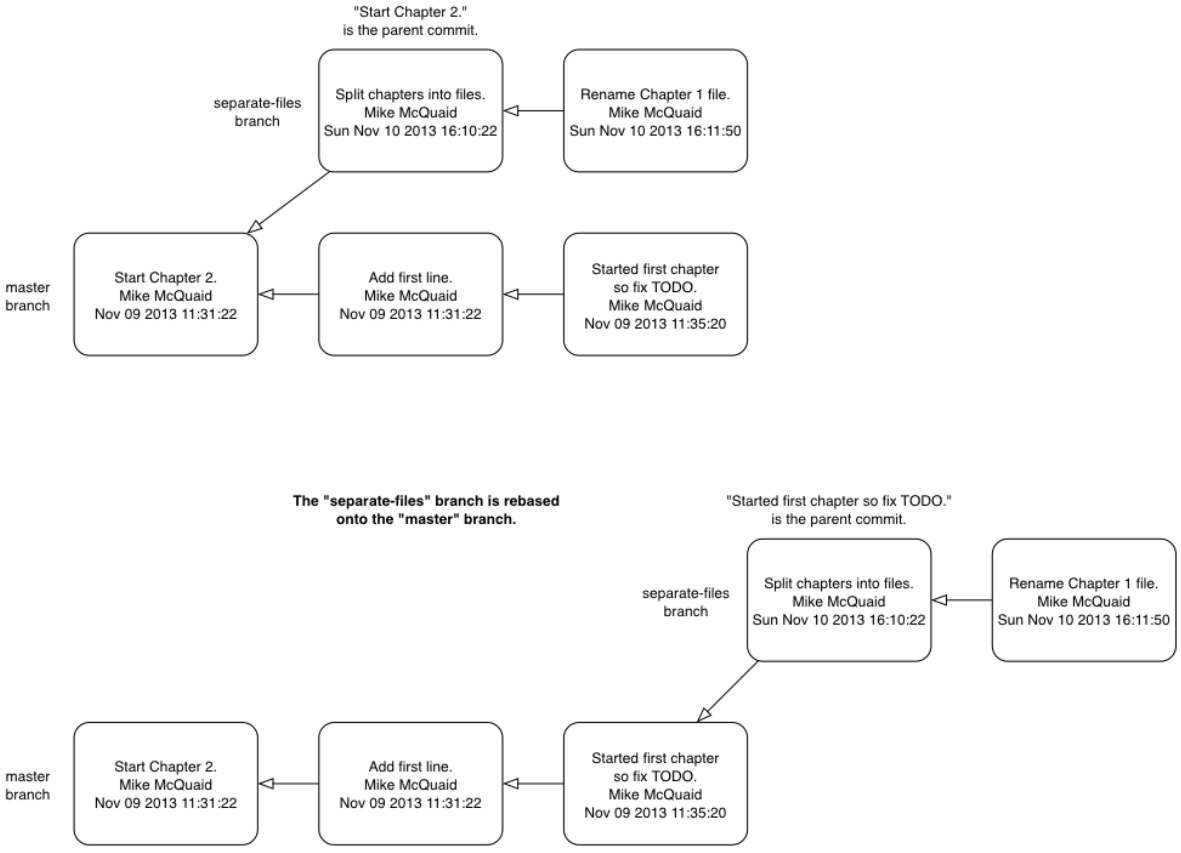

Figure 1.12 Rebasing a branch on top of master

Figure 1.12 shows a rebase of the `seperate-files` branch onto the `master` branch. The rebase operation has changed the parent of the first commit in the `separate-files` branch to be the last commit in the `master` branch.

This means all the content changes from the `master` branch are now included in the `separate-files branch` and any conflicts were manually resolved but were not stored (as they would be in a merge conflict).

We'll cover rebasing in more detail later in the book. All that's necessary to remember for now is that it's a different approach to a merge that can be used for a similar outcome (pulling changes from one branch into another).

1.8 Remote Repositories: exchanging commits with another computer

Typically when using version control you will want to share your commits (or branches) with other people using other computers. With a traditional, *centralized version control system* (such as Subversion or CVS) the repository is usually stored on another machine. As you make a commit it is sent over the network, checked that it can apply (there may be other changes since you last checked) and then committed to the version control system where others can see it.

With a *distributed version control system* like Git every user has a complete repository on their own computer. While there may be a centralized repository that people send their commits to it will not be accessed unless specifically requested. All commits, branches and history are stored offline unless users choose to send or receive commits from another repository.

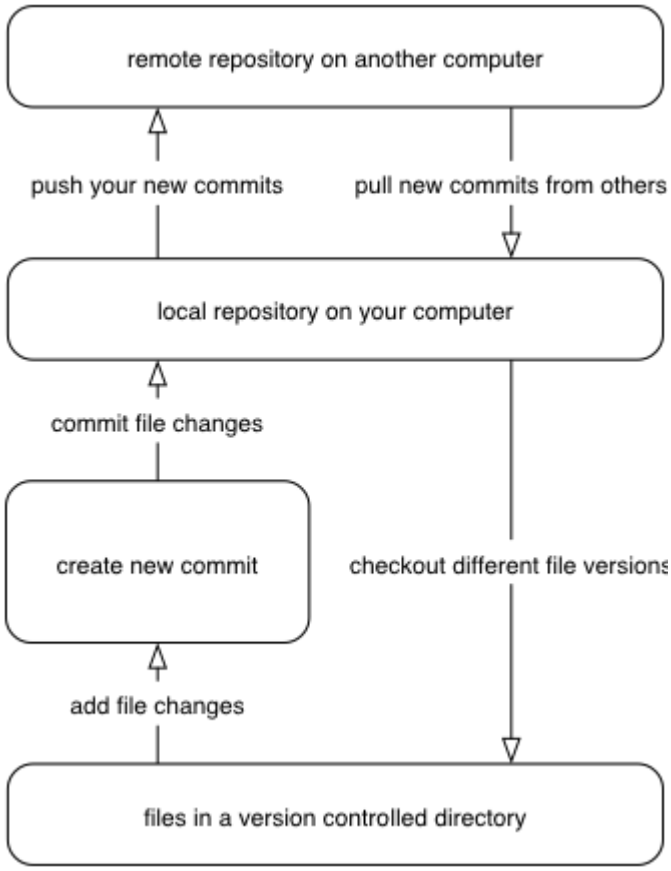

Figure 1.13 Git add/commit/push/pull/checkout workflow

A repository you send or receive commits to is known as a *remote repository*. You control when these changes are sent or received. Figure 1.13 shows the workflow for making commits in your local repository and then pushing them to a remote repository and pulling those made by others back to your local repository.

1.8.1 Communicating with a remote repository

Changes are sent to a remote repository in a *push* operation and received in a *pull* (or *fetch*) operation. When either of these occur your repository talks to the other repository, finds out what you know in common and sends only the differences between the two repositories (obviously with large repositories to do otherwise would be very slow).

1.8.2 Authoritative version storage

With centralized version control systems the central server always stores the authoritative version of the code. Clients to this repository will typically only store a small proportion of the history and require access to the server to perform most tasks. With distributed version control system like Git every local repository has a complete copy of the data. Which repository stores the authoritative version in this case? It turns out that this is merely a matter of convention; Git itself does not deem any particular repository to have any higher priority than another. Typically in organizations there will be a central location (like with a centralized version control) which is treated as the authoritative version and people are encouraging to push their commits and branches to.

The lack of authority for a particular repository with distributed version control systems is sometimes seen as a liability but can actually be a strength. The Linux kernel project (for which Git was original created) makes use of this to provide a network of trust and a more manageable way of merging changes. When Linus Torvalds, the self-named "benevolent dictator" of the project, tags a new release this is generally considered a new release of Linux. What is in his repository (well, his publicly accessible one; he will have multiple repositories between various personal machines that he does not make publicly accessible) is generally considered to be what is in Linux. Linus has trusted lieutenants from who he can pull and merge commits and branches. Rather than every single merge to Linux needing to be done by Linus he can leave some of it to his lieutenants (who leave some to their sub-lieutenants and so on) so everyone can needs only worry about verifying and including the work of a small number of others. This particular workflow may not make sense in many organizations but it demonstrates how distributed version control systems can allow different ways of managing merges to centralized version control.

1.9 Summary

In this chapter you hopefully learned:

- *Version control systems* exist to manage a series of changes over time to files in a project. They commonly used by programmers and provide a more robust alternative than manually renaming files to a form like `Document FINAL V4.txt`.
- A *commit* is a particular change to one or more files. As well as the changed file contents they also store the author, date and time, a unique reference, a explanatory *commit message* and a reference to their *parent commit*.
- *History* is the series of *commits* to a version control system over time. It tracks from the

current commit through the *parent commit* pointers all the way back to the *initial commit*. In Git past actions in the history can be *rewritten*.

- A *diff* is the difference between any two commits or parts of the history. There are various formats and they display how the text was changed and allow analysis of past changes to the history.
- *Branches* are independent paths of history. They allow commits to be made that are separate from changes made in another branch so incomplete work can be left and returned to later.
- A *merge* is when a branch's commits are brought into another branch. The *merge commit* joining the two branches has two *parent commits*. Sometimes merges cannot be done automatically and the version control system creates a *merge conflict*. When resolved this conflict is stored in the *merge commit*.
- A *remote repository* is a repository that is not stored on the current machine. Commits may be sent to or received from a remote repository to share work with others. *Centralized version control* sends new commits to a remote repository immediately. *Distributed version control* only sends new commits to a remote repository on request.

Now let's learn how to use these concepts to create and interact with a Git repository on your local machine.

2 Introduction to local Git

In this chapter you will learn how and why to use Git as a local version control system by covering the following topics:

- How to install Git
- How to install and use gitk/GitX
- How to create a new local Git repository
- How to commit to a Git repository
- How to view the history of a Git repository
- How to view the differences between Git commits/branches/tags

Git repositories store all their data on your local machine. This means when you create a new Git repository it does not need to send or receive any data from other computers in order to be useful. When you make commits, view history or request diffs these are all local operations that do not require a network connection. For these reasons Git is fast and is useful for storing versions of files on your local machine. Let's start using Git by getting it installed.

2.1 Installation

Let's see if Git is already installed on your local machine and install it if needed. The method to do so varies depending on your operating system of choice.

NOTE **Why do I need to install Git?**
Git does not come pre-installed on many operating systems as it is a tool typically used by programmers rather than non-technical computer users.

NOTE	**Why are there different versions of Git in this section?** The different installation methods and operating systems install different versions of Git. Do not worry about this; the main differences between newer Git versions and older ones are the helpfulness of the output messages. Version 1.7 or above should be fine for the needs of this book.

2.1.1 How to install and run Git on Apple OS X

To verify if Git is already installed open a Terminal (either the default OS X `/Applications/Utilities/Terminal.app` or an alternative such as `iTerm.app`) and run `git --version`. If Git is already installed the output should resemble Figure 2.1:

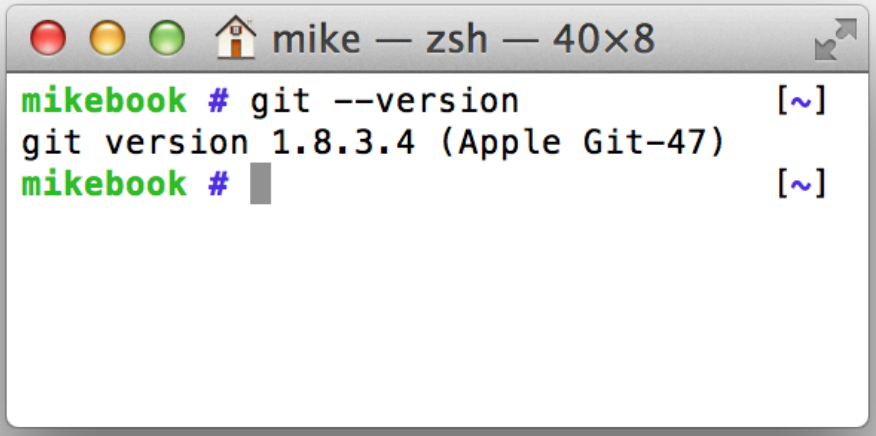

Figure 2.1 `git --version` in Terminal.app on OS X Mavericks

INSTALLING GIT ON APPLE OS X MAVERICKS OR NEWER

If you are running OS X Mavericks (10.9) or newer and Git was not already installed when you ran `git --version` it will prompt to download and install Git similarly to to Figure 2.2:

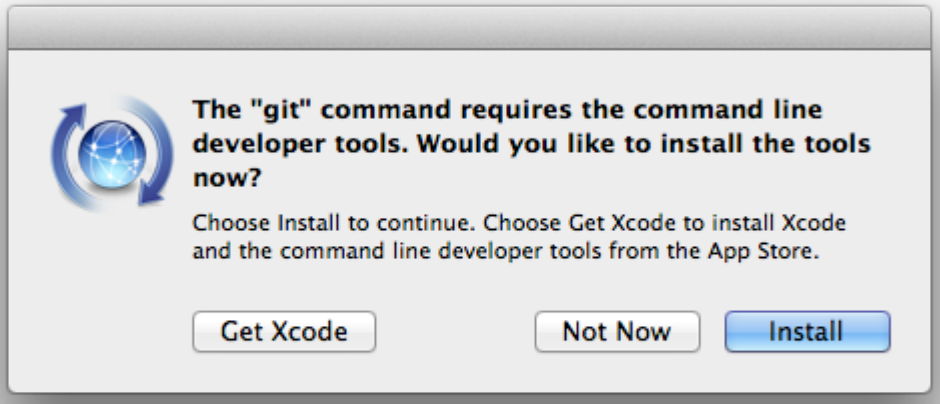

Figure 2.2 OS X Mavericks Git installation

INSTALLING GIT ON APPLE OS X MOUNTAIN LION OR OLDER

If you are running OS X Mountain Lion (10.8) or older and you have a package manager installed you can install Git using one of the options below:

- Homebrew/Tigerbrew (recommended): `brew install git`
- MacPorts: `sudo port install git-core +svn`
- Fink: `fink install git`

If you do not wish to install or use a package manager you can install Git using a graphical installer from the official Git site at http://git-scm.com/download/mac.

2.1.2 How to install and run Git on Linux or Unix

To verify if Git is already installed open a Terminal application or console and run `git --version`. If Git is already installed the output should resemble Figure 2.3:

Figure 2.3 `git --version` in XFCE Terminal on Debian 7.2 (Wheezy)

On Linux or Unix you can install Git directly from your package manager. How to do this varies from system to system but some of the popular options are below:

- Debian/Ubuntu: `apt-get install git`
- Fedora: `yum install git`
- Gentoo: `emerge --ask --verbose dev-vcs/git`
- Arch Linux: `pacman -S git`
- FreeBSD: `cd /usr/ports/devel/git && make install`
- Solaris 11 Express: `pkg install developer/versioning/git`
- OpenBSD: `pkg_add git`

2.1.3 How to install and run Git on Microsoft Windows

To verify if Git is already installed look for "Git Bash" links in your Start Menu or on your Desktop.

Git for Windows can be downloaded from the official Git site at http://git-scm.com/download/win. Download and click through the installer. When it has completed it will provide Start Menu links to run Git Bash.

As Git is a Unix program running Git on Windows will run a Unix shell which allows access to Git commands. This may be slightly scary but don't worry; this book will show any commands you'll need to use.

To run Git commands open the *Git Bash* shortcut from the Start Menu. This will open a Unix shell in a Windows Command Prompt.

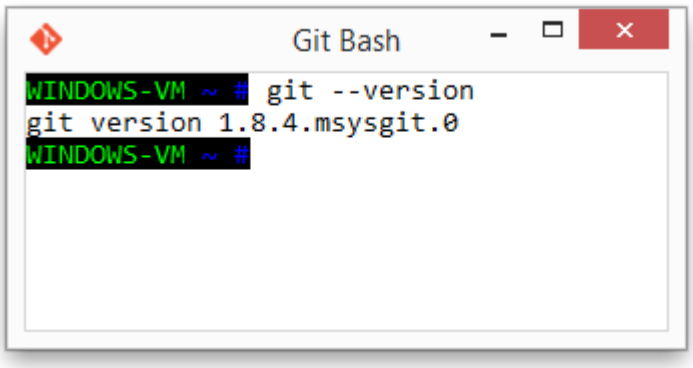

Figure 2.4 `git --version` in Git Bash on Windows 8.1

With the Git shell open you can type in Git commands. To see what Git version you have installed type `git --version`. The output should resemble Figure 2.4.

2.1.4 Verifying Git has installed correctly.

To run Git commands you will need to open a Terminal application, console or command-prompt (depending on your platform). To verify that Git has installed correctly run `git --version` which should output `git version 1.8.4.3` (or another version).

2.1.5 Gitk/GitX tools

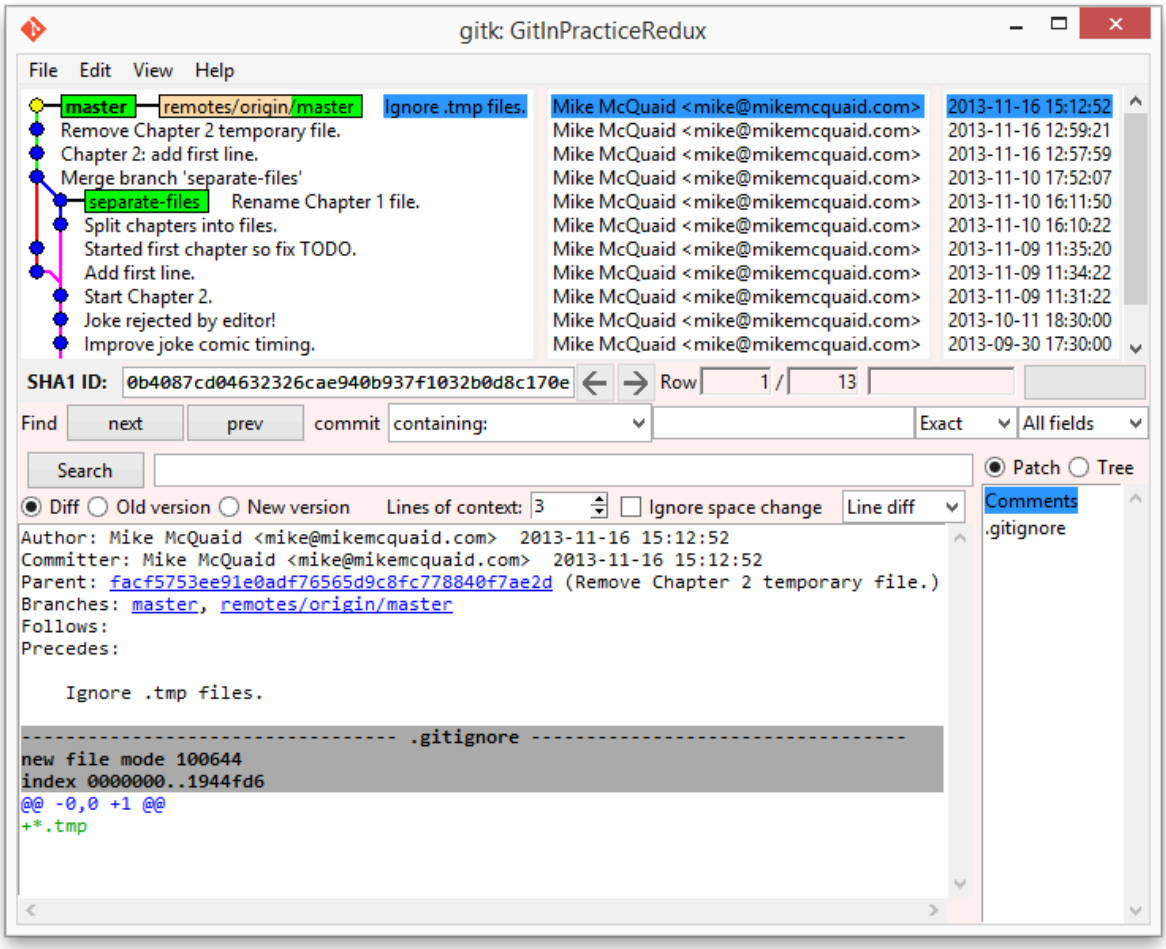

Figure 2.5 `gitk` on Windows 8.1

`gitk` is Git tool for viewing the history of Git repositories. It is usually installed with Git but may need installed by your package manager or separately. It's ability to graphically visualize Git's history is particularly helpful when history becomes more complex (e.g. with merges and remote branches). It can be seen running on Windows 8.1 in Figure 2.5.

There are attractive, up-to-date and platform-native alternatives to `gitk`. On Linux/Unix there are tools such as `gitg` for gtk+/GNOME integration and `QGit` for Qt/KDE integration. These can be installed using your package manager.

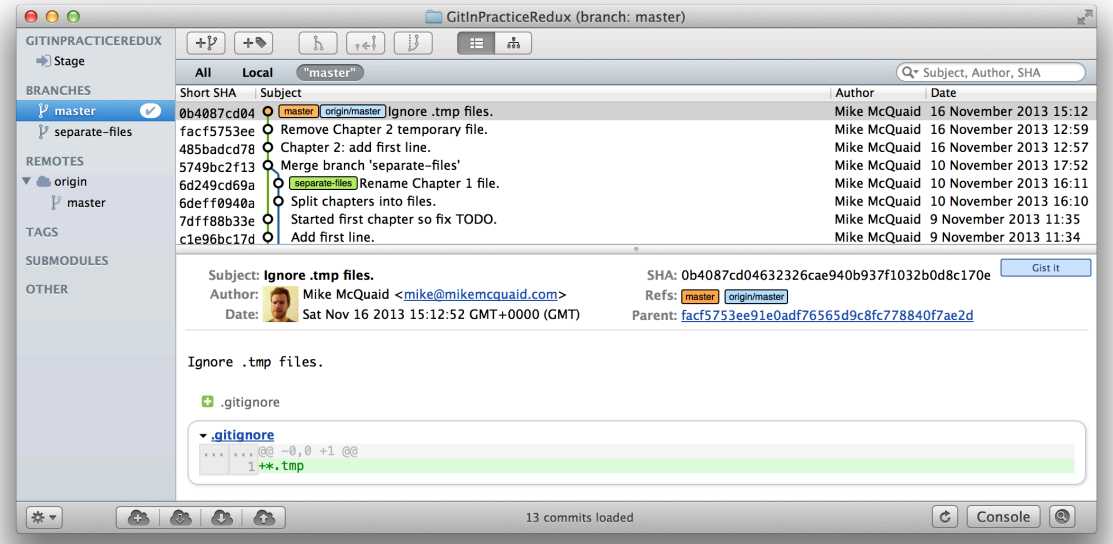

Figure 2.6 GitX-dev on OS X Mavericks

On OS X there are tools such as `GitX` (and various forks of the project). As OS X is my platform of choice I'll be using screenshots of the `GitX-dev` fork of `GitX` to discuss history in this book and would recommend you use it too if you use OS X. `GitX-dev` is available at https://github.com/rowanj/gitx and can be seen in Figure 2.6.

2.2 Creating a repository

Once you've installed Git the first thing you need to do to use it on your local machine is to create a Git repository. The Git repository is a folder on disk where Git keeps track of the state of the files within it.

Typically you create a new repository to do this by downloading (known as *cloning* by Git and introduced in Chapter 3) another repository that already exists but let's start by creating an empty, new local repository. Remember to run any Git commands requires an open Terminal/console/Git Bash so open one and let's create a repository.

2.2.1 The git init command

A Git repository must be initialized before any files can be added, commits made or pushed elsewhere. When `git init` is run it creates a named directory (if passed; otherwise uses the current directory).

To create a new local Git repository in a new subdirectory named `GitInPracticeRedux:

1. Change to the directory you wish to contain your new repository directory e.g. `cd /Users/mike/`.
2. Run `git init GitInPracticeRedux`.

You have created a new local Git repository named `GitInPracticeRedux` accessible at e.g. `/Users/mike/GitInPracticeRedux`.

Under this directory a subdirectory at e.g `/Users/mike/GitInPracticeRedux/.git/` which is created with various files and directories.

NOTE **Why is the `.git` directory not visible?**

On some operating systems by default directories starting with a `.` such as `.git` will be hidden by default. They can still be accessed in the console using their full path (e.g. `/Users/mike/GitInPracticeRedux/.git/`) but will not show up in file listings in file browsers or by running e.g. `ls /Users/mike/GitInPracticeRedux/`.

Let's view the contents of the new Git repository by changing to directory containing the Git repository directory and running the `find` command e.g. `cd /Users/mike/ && find GitInPracticeRedux`

Listing 2.1 Files created in a new Git repository

```
GitInPracticeRedux/.git/config
GitInPracticeRedux/.git/description
GitInPracticeRedux/.git/HEAD
GitInPracticeRedux/.git/hooks/applypatch-msg.sample
GitInPracticeRedux/.git/hooks/commit-msg.sample
GitInPracticeRedux/.git/hooks/post-update.sample
GitInPracticeRedux/.git/hooks/pre-applypatch.sample
GitInPracticeRedux/.git/hooks/pre-commit.sample
GitInPracticeRedux/.git/hooks/pre-push.sample
GitInPracticeRedux/.git/hooks/pre-rebase.sample
GitInPracticeRedux/.git/hooks/prepare-commit-msg.sample
GitInPracticeRedux/.git/hooks/update.sample
GitInPracticeRedux/.git/info/exclude
GitInPracticeRedux/.git/objects/info
GitInPracticeRedux/.git/objects/pack
GitInPracticeRedux/.git/refs/heads
GitInPracticeRedux/.git/refs/tags
```

1. **local configuration**
2. **description file**
3. **HEAD pointer**
4. **event hooks**
5. **excluded files**
6. **object information**
7. **pack files**
8. **branch pointers**
9. **tag pointers**

The purpose of some of these files (seen in Listing 2.1) may be obvious to you if you have prior experience of version control. Git has created files for:

- "local configuration (1)" of the local Git repository
- "description file (2)" to describe the repository for those created for use on a server
- "HEAD pointer (3)", "branch pointers (8)" and "tag pointers (9)" which point to commits
- "*event hooks* (4)" samples; scripts that run on defined events e.g. pre-commit is run before every new commit is made
- "excluded files (5)" which manages files which should be excluded from the repository
- "object information (6)" and "pack files (7)" which are used for object storage and reference

You shouldn't edit any of these files directly until you have a more advanced understanding of Git (or perhaps never at all). You will instead modify these files and folders by interacting with the Git repository through Git's filesystem commands introduced in Chapter 4.

2.3 Committing changes to files

Like other version control systems to do anything useful in Git we first need one or more commits in our repository. To do this first requires adding files to Git's *index* .

2.3.1 Git's index: a staging area for new commits

Git's index is a staging area used to build up new commits. Rather than requiring all changes in the working tree make up the next commit Git allows files (and even lines within files) to be added incrementally to the index. The add/commit/checkout workflow can be seen in Figure 2.7.

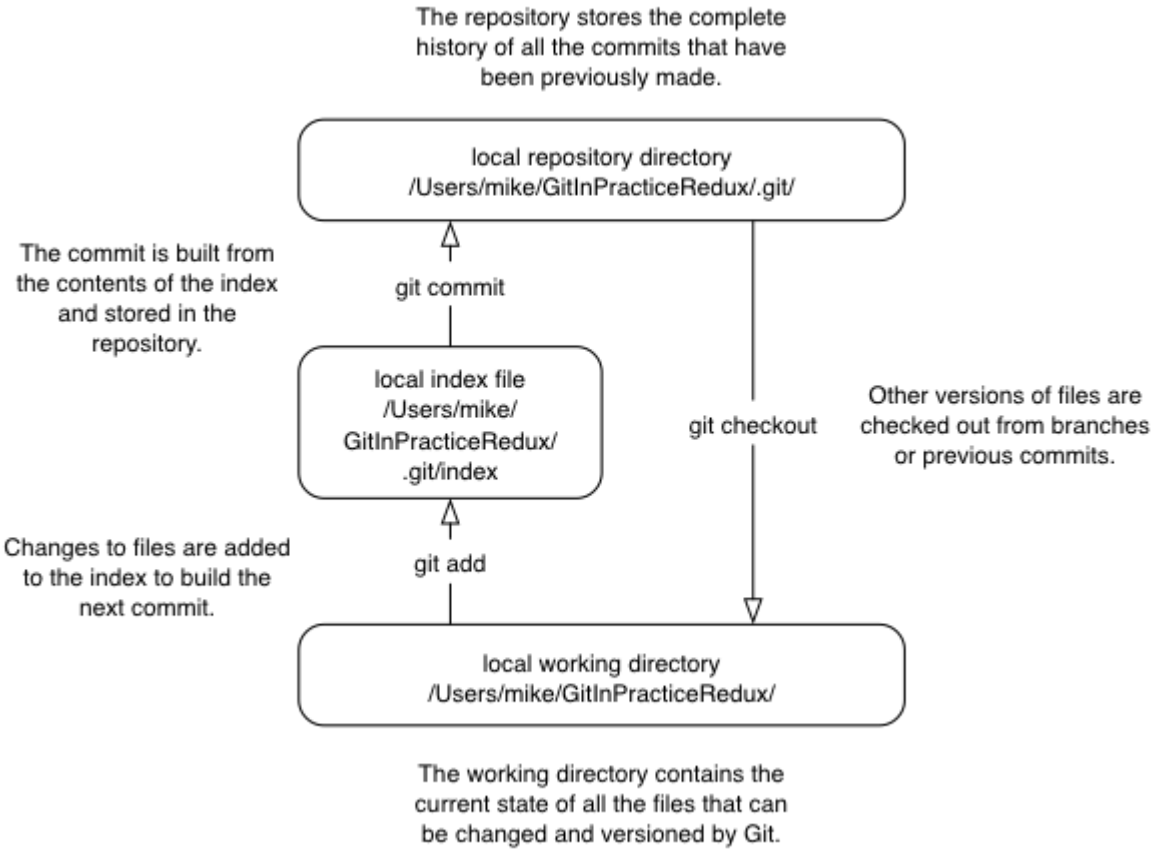

Figure 2.7 Git add/commit/checkout workflow

Git does not add anything to the index without your instruction. As a result, the first thing you have to do with a file we want to include in a Git repository is request Git to add it to the index.

2.3.2 The `git add` command: adding files to the index

To add an existing file `GitInPractice.asciidoc` to the index:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Ensure the file `GitInPractice.asciidoc` is in the current directory.
3. Run `git add GitInPractice.asciidoc`. There will be no output.

You have added the `GitInPractice.asciidoc` to the index. If this has been successful then the output of running `git status` should resemble:

Listing 2.2 Adding a file to the index

```
# git add GitInPractice.asciidoc

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   GitInPractice.asciidoc
#
```

1. **master is default branch**
2. **this is the first commit**
3. **new file added to index**

When a file is added to the index a file named `.git/index` is created (if it does not already exist). The added file contents and metadata are then added to the index file. You have requested two things of Git here:

1. for Git to track the contents of the file as it changes (this is not done without an explicit `git add`)
2. the contents of the file when `git add` was run should be added to the index, ready to create the next commit.

NOTE **Why do you need to keep running `git add`?**
As the file is changed the contents of the commit will not be updated to reflect these changes without another `git add`. This may appear strange; why would you not want to add new changes to the next commit? In Chapter 7 this approach of incrementally and explicitly constructing new commits will be used to create a more readable version control history.

Now that the contents of the file have been added to the index you're ready to make a new commit.

2.3.3 The `git commit` command: adding a new commit to the repository

To commit the contents of an existing file `GitInPractice.asciidoc`:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Ensure the file `GitInPractice.asciidoc` is in the current directory.
3. Run `git add GitInPractice.asciidoc`. There will be no output.
4. Run `git commit --message 'Initial commit of book.'`. The output should resemble:

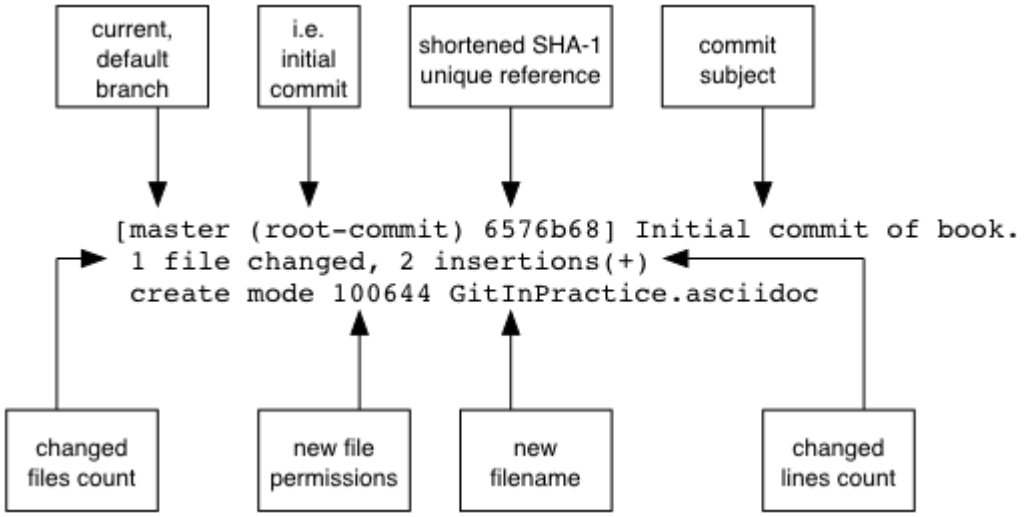

Figure 2.8 `git commit` output

You have made a new commit containing `GitInPractice.asciidoc`.

The output of `git commit` can be seen in Figure 2.8. To expand on the annotations in this diagram:

- *current, default branch*. The branch on which the commit was made. The default branch in Git is master so that is what is shown here (as you never explicitly created a branch).
- *i.e. initial commit*. As this was the first commit in the repository it is known as the *root-commit* or *initial commit*. This means it has no parent commit. This part of the `git commit` output is only shown for the first commit.
- *shortened SHA-1 unique reference*. Every commit in Git is given a unique 40 hexadecimal character SHA-1 hash of the contents and metadata of that commit. As these are rather unwieldy Git will often show shortened versions (as long as they are unique in the repository). Anywhere that Git accepts a SHA-1 unique commit reference it will also accept the shortened version.
- *commit subject*. The commit message you entered is structured like an email. The first line of it is treated as the subject and the rest as the body. The commit subject will be used as a summary for that commit when only a single line of the commit message is shown.
- *changed files count*. On a new commit Git will always show how many files were added, modified or deleted in the commit. In this case we added one file (`GitInPractice.asciidoc`).
- *new file permissions*. This is the file mode for the newly created file. These are related to Unix file permissions and the `chmod` command but are not important in understanding how Git works so can be safely ignored.
- *new filename*. This shows what filenames have been added or deleted in this commit.
- *changed lines count*. On a new commit Git will also show how many lines were added, modified or deleted across all the files in the commit. In this case I added one new file with two new lines.

NOTE **What is a SHA-1 hash?**
A "SHA-1 hash" is a secure hash digest function that is used extensively inside of Git. It outputs a 160-bit (20-byte) hash value which is usually displayed as a 40 character hexadecimal string. The hash is used to uniquely identify commits by Git instead of e.g. incremental revision numbers like Subversion. Git will also accept shortened versions of SHA-1 hashes as long as the shortened version is also unique inside the repository.

Let's see the output after modifying the contents of the `GitInPractice.asciidoc` file, running `git add` and `git commit`:

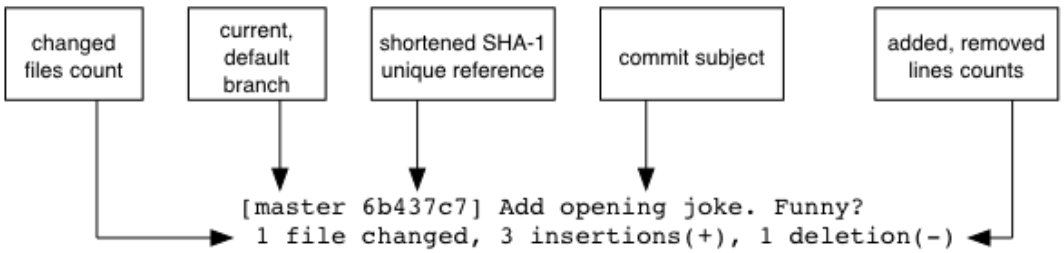

Figure 2.9 second `git commit` output

There are a few changes in Figure 2.9 from Figure 2.8:

- No *root commit* is shown as this is the second, non-initial commit which has the first commit as its parent.
- *shortened SHA-1*. As this is a new commit with different contents and metadata the SHA-1 differs from the initial commit.
- *added, removed lines count*. Two new lines were inserted and one was modified. This shows three insertions and one deletion because Git treats the modification of a line as the deletion of an old line and insertion of a new one.

Now that we have two commits we can start looking at Git's history.

2.4 History

Git's history stores the graph of all commits in the repository. Viewing it is useful for working out where you are in terms of branches and previous commits.

The first command you will use to navigate history is `git log`.

2.4.1 The git log command: viewing the history

To view the commit history (also known as log):

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.

2. Run `git log`. The output should resemble:

Listing 2.3 History output

```
# git log

commit 6b437c7739d24e29c8ded318e683eca8f03a5260
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Sun Sep 29 11:30:00 2013 +0100

    Add opening joke. Funny?

commit 6576b6803e947b29e7d3b4870477ae283409ba71
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Sun Sep 29 10:30:00 2013 +0100

    Initial commit of book.
```

1. **unique SHA-1**
2. **commit author**
3. **committed date**
4. **full commit message**

The `git log` output lists all the commits that have been made on the current branch in reverse chronological order i.e. the most recent commit comes first. You can see the two commits that were made in the previous section and how they are represented by Git. The *commit* lists the full 40 character "unique SHA-1 (1)" (that is sometimes shown abbreviated). The "commit author (2)" name and email address set by the person who created the commit. The "committed date" is the date and time the commit was created. The additional text is the "full commit message"; the first line is the commit message subject and the rest the commit message body.

It’s also helpful to visualize the history graphically.

2.4.2 Viewing history with gitk/GitX tools

To view the commit history with gitk or GitX:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `gitk` or `gitx`.

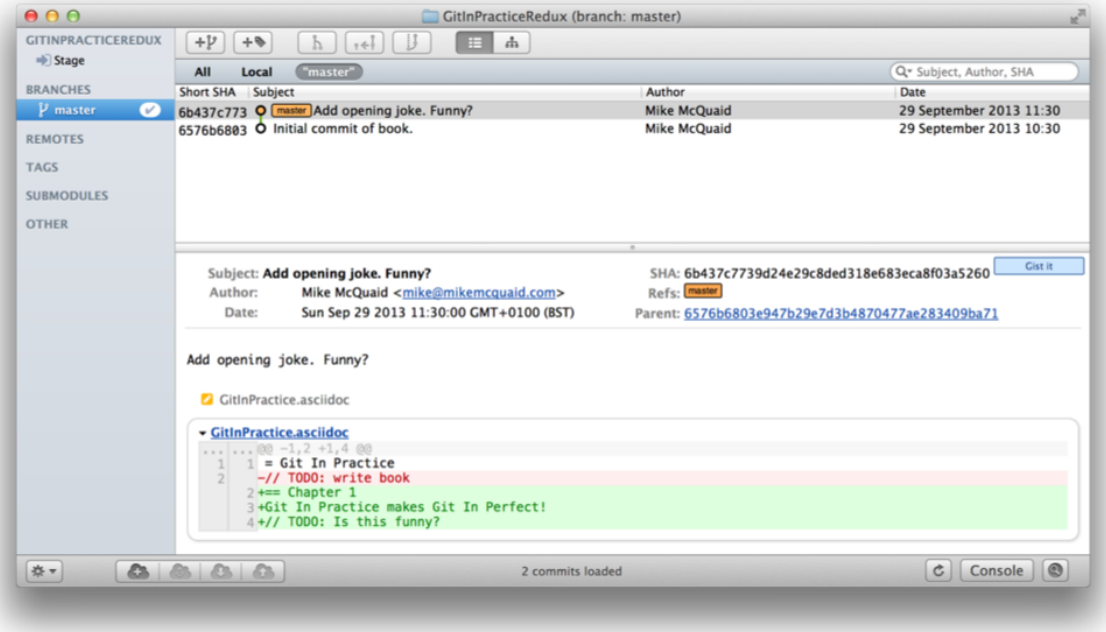

Figure 2.10 GitX history output

The GitX history (seen in Figure 2.10) shows similar output to `git log` but in a different format. You can also see the current branch and the contents of the current commit including the diff and parent SHA-1. There's a lot of information that doesn't differ between commits, however.

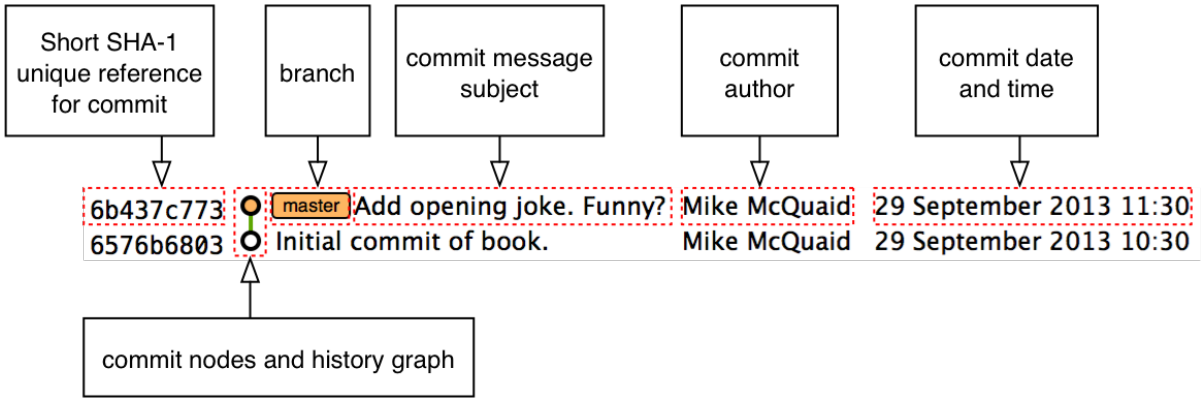

Figure 2.11 GitX history graph output

In Figure 2.11 you can see the GitX history graph output. This format will be used throughout the book to show the current state of the repository and/or the previous few commits. It concisely shows the unique SHA-1, all branches (only `master` in this case), the current local branch (shown with an orange "node" and label), the commit message subject (the first line of the commit message) and the commit's author, date and time.

The history can give us a quick overview of all the previous commits. However, querying the differences between any two arbitrary commits can also sometimes be useful so let's learn how to do that.

2.5 Diffs: differences between commits

You learnt in the previous chapter that diffs are the differences between two commits. In Git we are able to reference commits using various references (known by Git as *refs*).

2.5.1 Git refs: different references for individual commits

In Git *refs* are the possible ways of addressing individual commits. They are an easier method to type a reference to a specific commit or branch when querying the difference between commits (or other techniques we'll learn later)

Remember that a SHA-1 (shortened or the full 40 characters) is a unique reference to a commit. What about other ways of referencing a commit?

The first ref you have already seen is by the branch (which is `master` by default if you haven't created any other branches). If you remember from the previous chapter, branches are actually pointers to a specific commit. Therefore referencing the SHA-1 of commit at the top of the master branch (the short version from the last example being `6b437c`) is the same as referencing the branch name `master`. In this case whenever you might type `6b437c` to a command (such as `git diff` as we will see later this section) you could instead type `master`. Using branch names is quicker and easier to remember for referencing commits than always using SHA-1s.

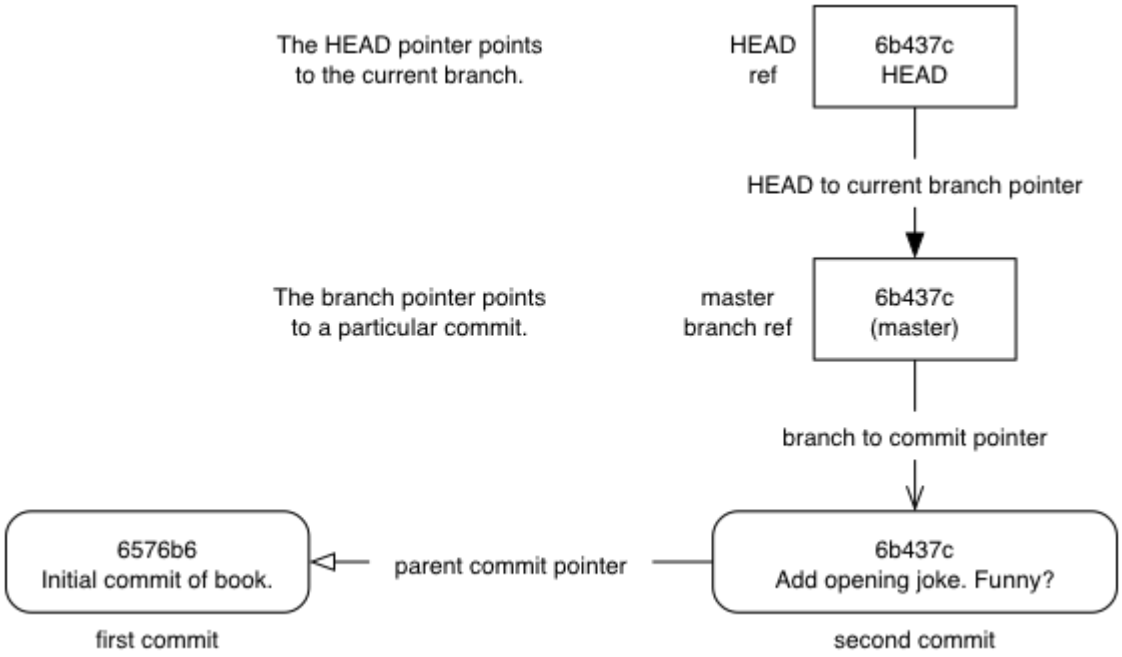

Figure 2.12 HEAD

The second ref is `HEAD`. The `HEAD` always points to the top of whatever you have currently checked out so almost always be the top commit of the current branch you are on. Therefore if you have the `master` branch checked out then `master` and `HEAD` (and `6b437c7` in the last example) are equivalent. See the `master`/`HEAD` pointers demonstrated in Figure 2.11.

The third ref is a tag. Tags are very similar to branches in Git but don't update as branches do when you make more commits on top of them. We'll discuss tags more in the next chapter.

There are more types (such as remote references) but you don't need to worry about them just now; they will be introduced in Chapter 3.

Refs can also have modifiers appended. Suffixing a ref with `^` is the same as saying *the commit before that ref*. For example `HEAD^` is the commit before the currently checked out commit and `master^` is the penultimate commit on the master branch. Another modification allows you to specify the number of commits to look before. `HEAD~2` is two commits before the currently checked out branch. Note that `HEAD^` and `HEAD~1` are equivalent.

Now that you know various ways to reference commits lets see how to query the differences between two commits.

2.5.2 The git diff command

The `git diff` command allows you to query the differences between two commits (or refs).

To see the diff between the current state of the working directory and the penultimate commit:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git diff HEAD^`. The output should resemble:

Listing 2.4 The differences to the commit before HEAD output

```
# git diff HEAD^

diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc   (1)
index 48f7a8a..b14909f 100644   (2)
--- a/GitInPractice.asciidoc   (3)
+++ b/GitInPractice.asciidoc   (4)
@@ -1,2 +1,4 @@   (5)
 = Git In Practice
-// TODO: write book   (6)
+== Chapter 1   (7)
+Git In Practice makes Git In Perfect!   (8)
+// TODO: Is this funny?
```

(1) virtual diff command
(2) index SHA-1 changes
(3) old virtual path
(4) new virtual path
(5) diff offsets
(6) modified/deleted line
(7) modified/inserted line
(8) inserted line

The `git diff` output (seen in Listing 2.2) contains some similar elements to the `git commit` or `gitx` output we looked at earlier. The "virtual diff command (1)" is the invocation of the Unix `diff` command that Git is simulating. Git pretends that it is actually diffing the contents two folders, the "old virtual path (3)" and the "new virtual path (4)" and the "virtual diff command (1)" represents that. The "index SHA-1 changes" show the difference in the contents of the working tree between these commits. This can be safely ignored other than noticing that these SHA-1s do not refer to the commits themselves. The "diff offsets (5)" can

also be ignored; they are used by the Unix `diff` command to identify what lines the diff relates to for files that are too large for `diff` to show the entire file. The "modified/deleted (6)" line and "modified/inserted (7)" line relate to the single line that was modified and the "inserted line (8)" is one of the two lines that was inserted in this commit.

These changes indicate the differences between the two states we requested: the commit before HEAD and the (implicitly requested) current state of the working tree.

We could request the difference between the last committed revision and the previous revision by providing two arguments to `git diff`:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git diff HEAD HEAD^`. The output should resemble:

Listing 2.5 The difference from `HEAD` to the commit before `HEAD` output

```
# git diff HEAD HEAD^

diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index b14909f..48f7a8a 100644
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
@@ -1,4 +1,2 @@
 = Git In Practice
-== Chapter 1
-Git In Practice makes Git In Perfect!
-// TODO: Is this funny?
+// TODO: write book
```

This time because we specified `HEAD` followed by `HEAD^` we see the changes in the `git diff HEAD^` but applied in reverse. If you remember from the refs earlier this would also be equivalent to `git diff master HEAD~1` or `git diff 6b437c7 6576b68` and will produce identical output.

2.6 Summary

In this chapter you hopefully learned:

- How to install and run Git on Apple OS X, Linux/Unix and Microsoft Windows.
- How to create a new local repository using `git init`.
- How to add files to Git's index staging area using `git add`.
- How to commit files to the Git repository using `git commit`.
- How to view history using `git log` and `gitk`/`gitx`.

- How to use refs to reference commits and their ancestors.
- How to see the differences between commits using `git diff`.

Now let's learn how to use these concepts to interact with repositories that are not stored on your local machine.

3 Introduction to remote Git

In this chapter you will learn how to retrieve and share changes with other users' Git repositories by learning the following topics:

- How to create a GitHub repository and download it
- How to send/receive changes from a remote repository
- How to create and receive branches
- How to merging commits from one branch to another

As you learned in Chapter 2 it’s possible to work entirely with Git as a local version control system and never share changes with others. Usually, however, if you are using a version control system you will want to share changes with others; from simply sending files to a remote server for backup to collaborating as part of a large development team. Team collaboration will also require knowledge of how to create and interact with branches for working on different features in parallel. Let’s create a remote repository on GitHub and a GitHub account to be able to store changes on another computer.

3.1 Creating a GitHub repository

GitHub is a website that provides Git repository hosting as well as issue trackers, Git-backed wikis and a workflow to request a merge of the commits in a branch (which is known as a *pull request* and will be shown in Chapter 12). You can create free accounts for public remote repositories which are where everyone can see your code and commits. Typically these are used by open-source projects but it will also prove useful for your learning and experimentation. For private projects GitHub offers paid accounts.

As mentioned in Chapter 2 there are free and paid alternatives to GitHub. I’ve

picked GitHub to walkthrough because, at the time of writing, it is the most popular hosted version control system for open-source projects and is probably the most popular Git hosting provider. Learning to use GitHub will bring immediate benefits in terms of facilitating open-source access and contributions. While the GitHub UI may differ from the examples here or from other Git repository hosts the Git commands used will remain the same.

3.1.1 Signing up for a GitHub account

Let's sign up for a new GitHub account. Please browse to https://github.com/join where you should see something like Figure 3.1:

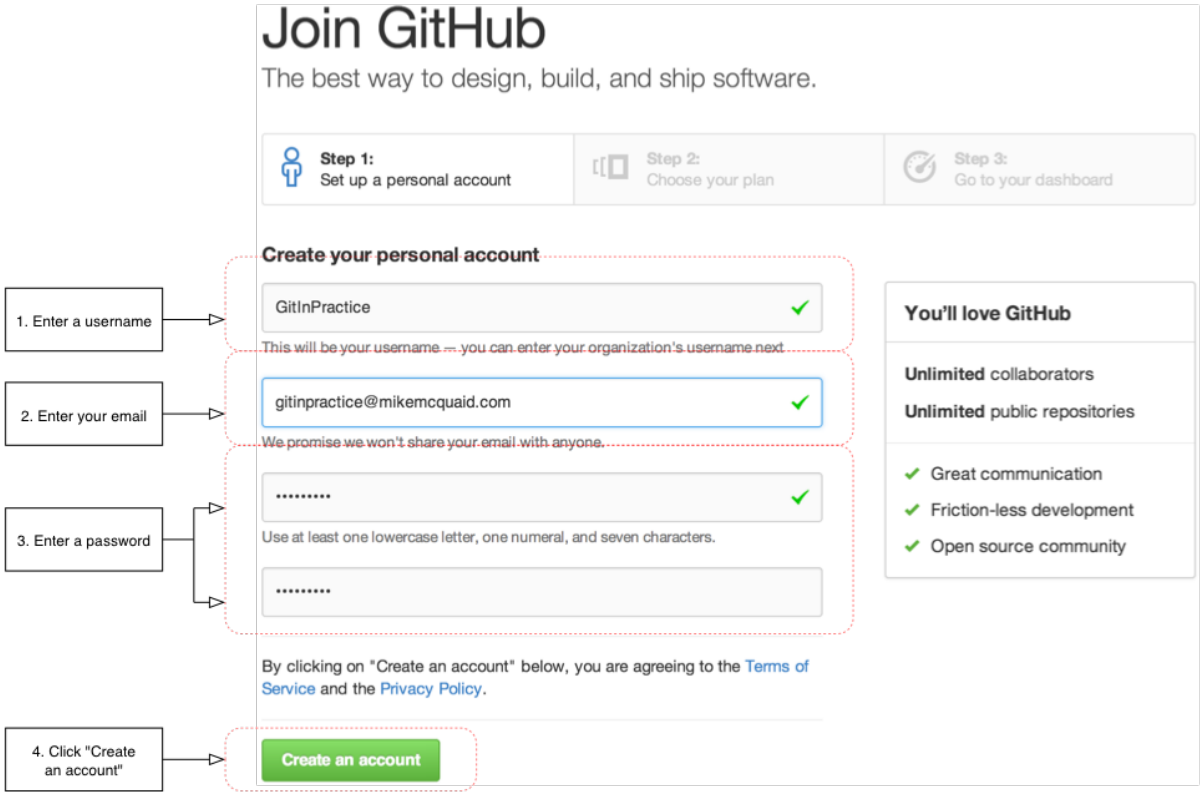

Figure 3.1 Join GitHub form

This form allows you to create a new GitHub account which will allow you to access the service and create new repositories. The username you pick will determine the URL of your GitHub account page and will be part of the URL for every repository you create so choose it carefully. It can be renamed in future but this may cause problems when updating existing local repositories without manually changing the URL.

Enter your username, email and password and click the create button to advance to the next screen.

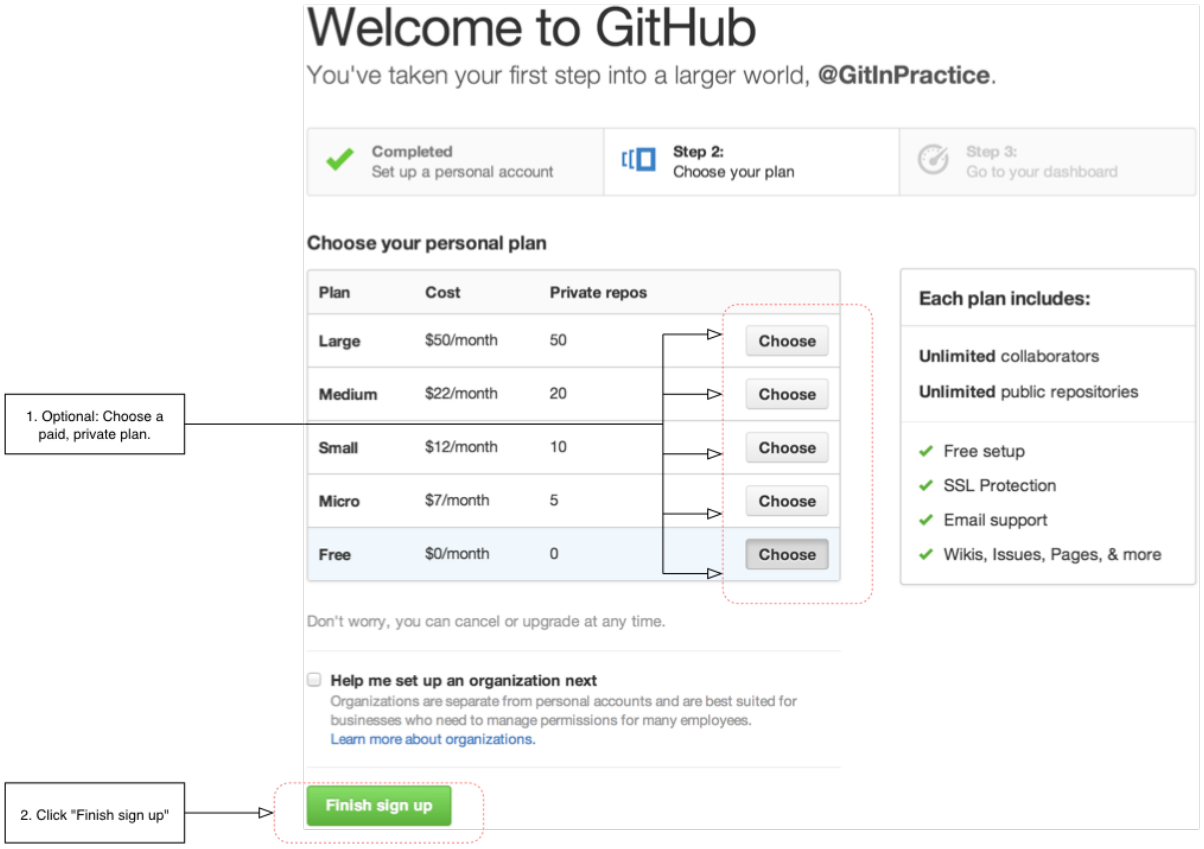

Figure 3.2 Choose GitHub plan

The form in Figure 3.2 allows you to select your GitHub payment plan. The only differences between plans are the number of private repositories you can create. Private repositories mean that none of your commits or files committed to the repository can accessed by others without your explicit approval. In this book you will never have to commit anything private to a repository so you do not need to choose a paid plan. After you have selected a plan click the finish button to advance to the next screen.

You have created a GitHub account and the next step is to create a new repository.

3.1.2 Creating a new repository on GitHub

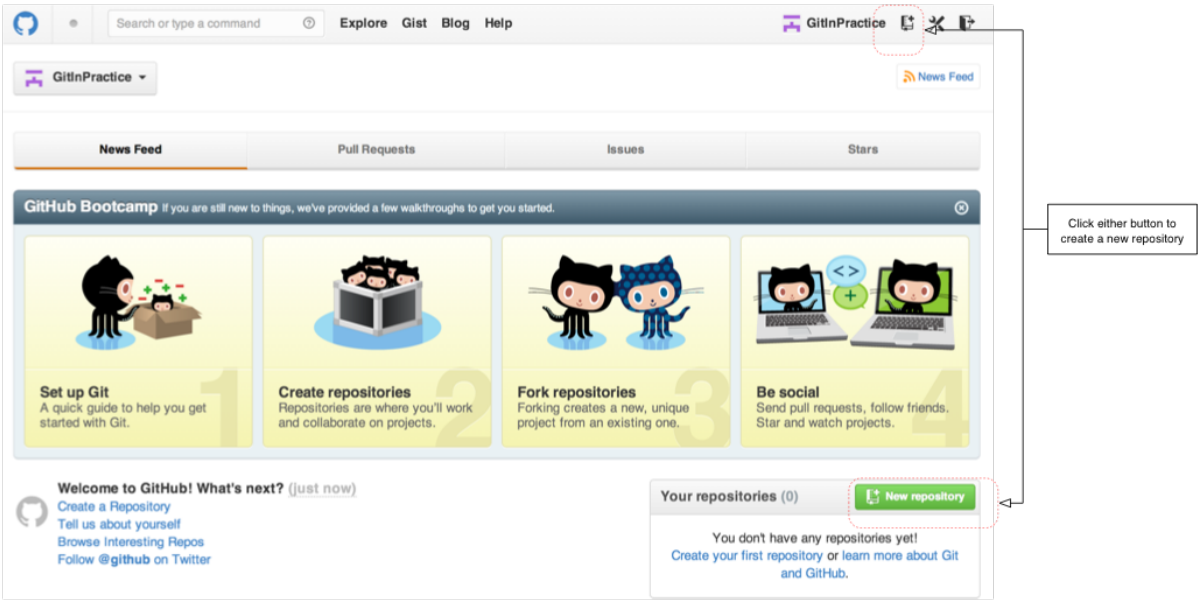

Figure 3.3 Dashboard buttons to create a new GitHub repository

After signing up for your new GitHub account you should see your dashboard which should resemble Figure 3.3. From the dashboard there are two buttons you can click to create a new GitHub repository. Click either of them to advance to the next screen.

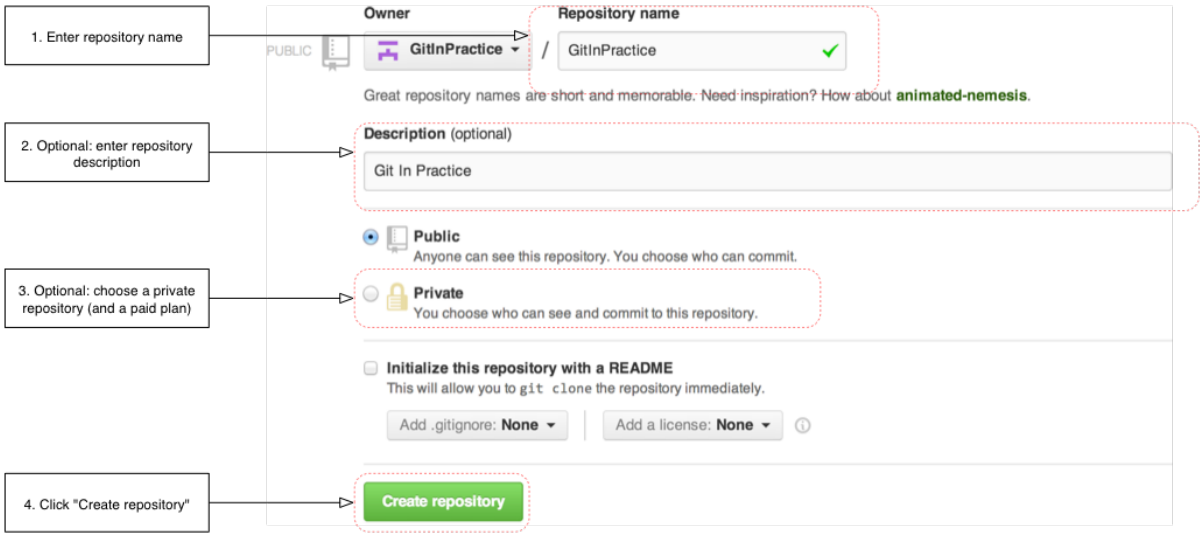

Figure 3.4 Create a new GitHub repository

Creating a new repository requires you to pick a name and optionally a description as in Figure 3.4. This name will be combined with the username you chose earlier to make the URL for your repository so choose it carefully. It can be

renamed in future but this may cause problems when updating existing local repositories without manually changing the URL. You may also choose for the repository to be private which requires purchasing a paid GitHub plan. After entering the repository details click the create button to advance to the next screen.

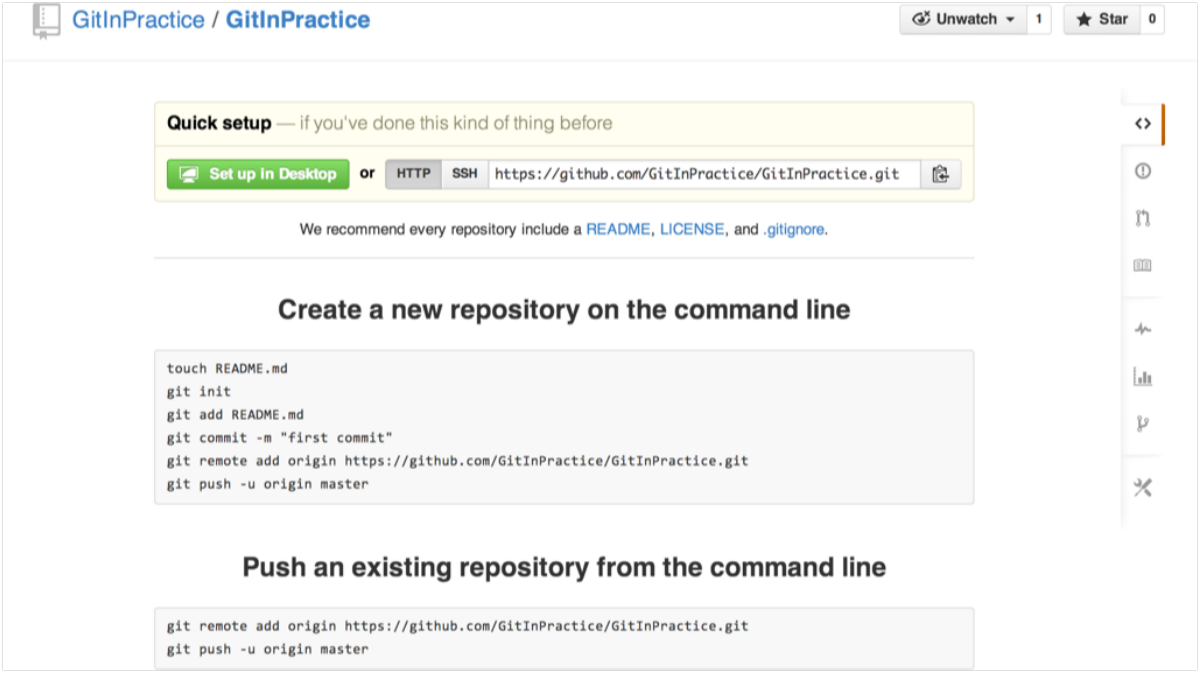

Figure 3.5 A new GitHub repository

You have created a GitHub repository and should see something similar to Figure 3.5. Now let's push the repository you created in Chapter 2 onto your local machine.

3.2 Remote Repositories

As discussed in Chapter 1 and 2 Git is a distributed version control system so any repository you have on your machine is a local repository. It can be committed to and the history queried without requiring a network connection to other repositories on other machines.

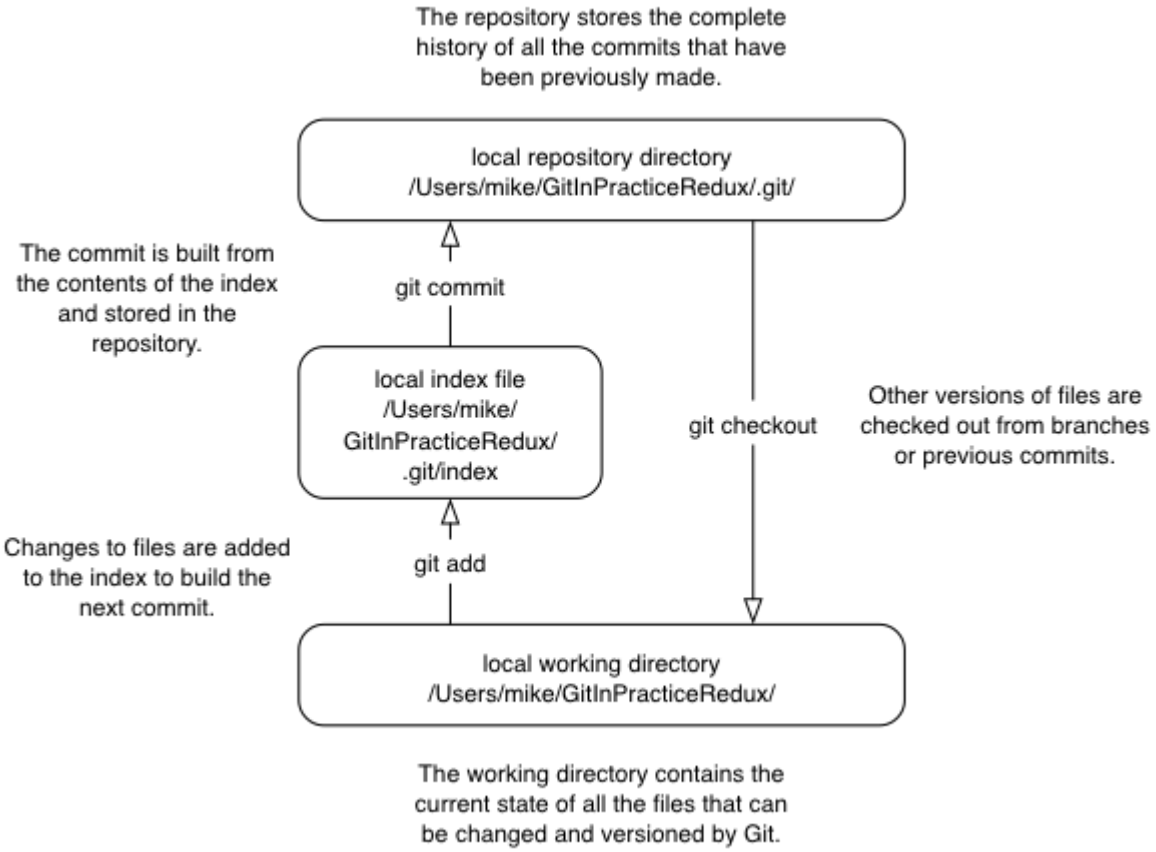

Figure 3.6 Git add/commit/checkout workflow

Figure 3.6 shows the local Git workflow we used in Chapter 2. Files in the local working directory are modified and added with `git add` to the index staging area. The contents of the index staging area is committed with `git commit` to form a new commit which is stored in the local repository directory. Later, this repository can be queried to view the differences between versions of files using `git diff`. In Section 3.3 you will also see how to use `git checkout` to change to different local branches' versions of files.

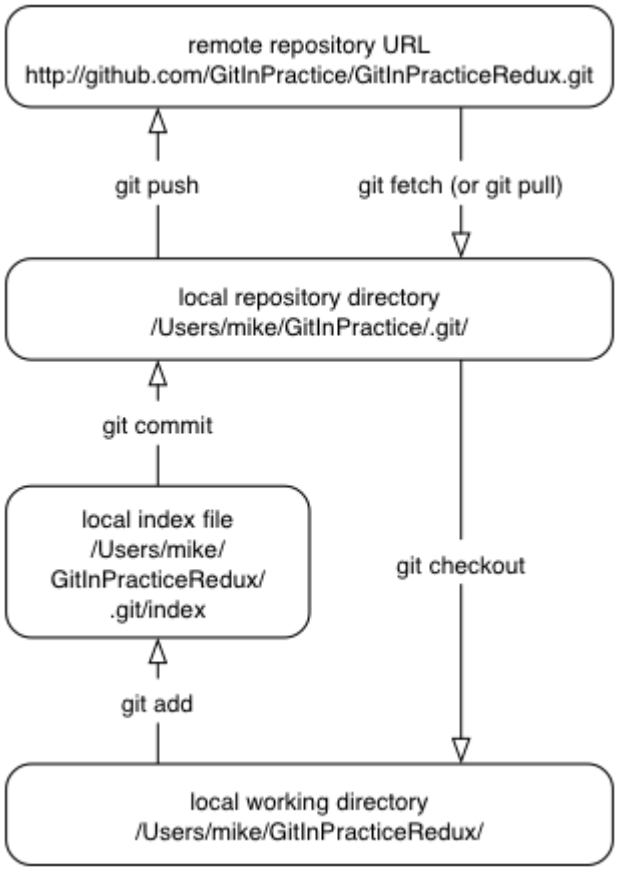

Figure 3.7 Git add/commit/push/pull/checkout workflow

Figure 3.7 shows the remote Git workflow we will look at in this Chapter. As in the local workflow files are modified, added, committed and can be checked out. However there are now two repositories: a local repository and *remote repository*.

If your local repository needs to send or receive data to a repository on another machine it will need to add a *remote repository*. A remote repository is one that is typically stored on another computer and `git push` sends your new commits to it and `git fetch` retrieves any new commits made by others from it.

In Chapter 2 you created a local repository on your machine and in Section 3.1 you've created a remote repository on GitHub so now let's connect the two together.

3.2.1 `git remote add` to add a new remote repository

The `git remote` command is used to perform actions around remote repositories (which are also known as *remotes*). The first action you're concerned with is adding a reference for your newly-created remote repository on GitHub in your previous local repository so you can push and fetch commits. To do this you'll use the `git remote add` subcommand.

NOTE **What is the default name for a remote?**
You can have multiple remote repositories connected to your local repository so the remote repositories are named. Typically if you have a single remote repository it will be named `origin`. This will be explained more in section 4.2.3.

To add the new GitInPractice remote repository to your current repository:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git remote add origin` with your repository URL appended. e.g. if your username is `GitInPractice` and repository is named `GitInPracticeRedux` run `git remote add origin https://github.com/GitInPractice/GitInPracticeRedux.git`. There will be no output.

You can verify this remote has been created successfully by running `git remote --verbose`. The output should resemble:

Listing 3.1 Remote repositories output

```
# git remote --verbose

origin  https://github.com/GitInPractice/GitInPracticeRedux.git (fetch)  (1)
origin  https://github.com/GitInPractice/GitInPracticeRedux.git (push)   (2)
```

(1) fetch URL
(2) push URL

The "fetch URL (1)" specifies the URL that `git fetch` uses to fetch new remote commits. The "push URL (2)" specifies the URL that `git push` uses to send new local commits.

You have added a remote named `origin` that points to the remote `GitInPracticeRedux` repository belonging to the `GitInPractice` user on GitHub.

You can now send and receive changes from this remote. Nothing has been sent or received yet; the new remote is effectively just a named URL pointing to the remote repository location. If you recall when we created the GitHub remote repository it was empty and told us to push changes to it so let's do that now.

3.2.2 `git push` to push changes to a remote repository

The `git push` command is used to send commits made in the local repository to a remote. Only changes specifically requested will be sent and the Git (which can operate over HTTP, SSH or it's own protocol (`git://`)) will ensure that only the differences between the repositories are sent. As a result you can push small changes from a large local repository to a large remote repository very quickly as long as they have most commits in common.

Let's push the changes you made in our repository in Chapter 2 to the newly created remote you made in the previous section.

To push the changes from the local `GitInPracticeRedux` repository to the `origin` remote on GitHub:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git push --set-upstream origin master` and enter your GitHub username and password when requested. The output should resemble:

Listing 3.2 Push and set upstream branch output

```
# git push --set-upstream origin master

Username for 'https://github.com': GitInPractice   (1)
Password for 'https://GitInPractice@github.com':   (2)
Counting objects: 6, done.   (3)
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 602 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git   (4)
 * [new branch]      master -> master   (5)
Branch master set up to track remote branch master from origin.   (6)
```

(1) username entry
(2) password entry
(3) object preparation/transmission
(4) remote URL
(5) local/remote branch
(6) set tracking branch

You have pushed your `master` branch's changes to the `origin` remote's `master` branch.

The "username entry (1)" and "password entry (2)" are those for your GitHub

account. They may only be asked for the first time you push to a repository depending on your operating system of choice (which may decide to save the password for you). They are always required to `push` to repositories but are only required for `fetch` when fetching from private repositories.

You can safely ignore the "object preparation/transmission (3)" section in this or future figures; it is simply Git communicating details on how the files are being sent to the remote repository and isn't worth understanding beyond basic progress feedback.

The "remote URL (4)" matches the push URL from the `git remote --verbose` output earlier. It is where Git has sent the local commits to.

The "local/remote branch (5)" line indicates that this was a new branch on the remote. This is because the remote repository on GitHub was empty until we pushed this; it had no commits and thus no `master` branch yet. This was created by the `git push`. The `master -> master` refers to the local master branch (the first of the two) has been pushed to the remote `master` branch (the second of the two). This may seem redundant but it is shown as it is possible (but ill-advised due to the obvious confusion it causes) to have local and remote branches with different names. Don't worry about local or remote branches for now as these will be covered in Section 3.3.

The "set tracking branch (6)" is shown because the `--set-upstream` option was passed to `git push`. By passing this option you have is told Git that you want the local `master` branch you have just pushed to *track* the `origin` remote's branch `master`. The `master` branch on the `origin` remote (which is often abbreviated as `origin/master`) is now known as the *tracking branch* (or *upstream*) for your local `master` branch.

A *tracking branch* is the default push or fetch location for a branch. This means in future you could run `git push` with no arguments on this branch and it will do the same thing as running `git push origin master` i.e. push the current branch to the `origin` remote's `master` branch.

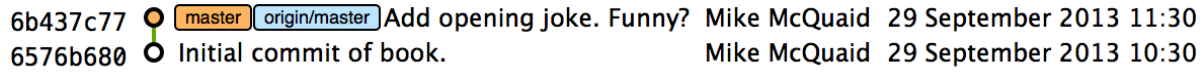

Figure 3.8 Local repository after `git push`

Figure 3.8 shows the state of the repository after the `git push`. There is one addition since we last looked at it in Figure 2.10: the blue, `origin/master`

label. This is attached to the commit which matches the currently known state of the `origin` remote's `master` branch.

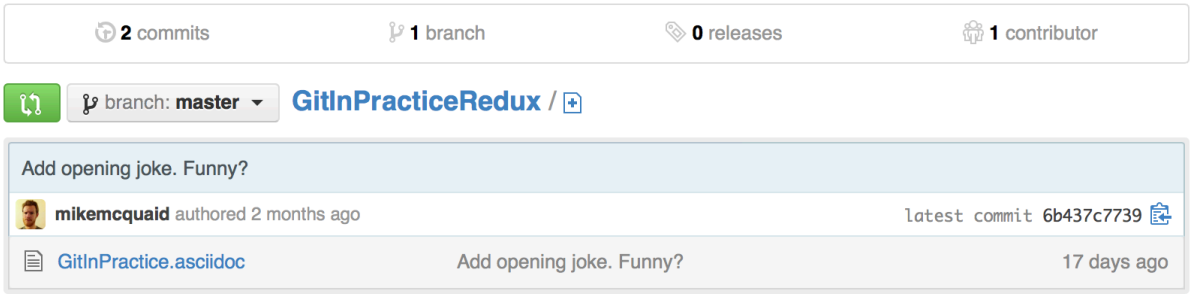

Figure 3.9 GitHub repository after `git push`

Figure 3.9 shows the remote repository on GitHub after the `git push`. The latest commit SHA-1 there matches your current latest commit on the `master` branch seen in Figure 3.8 (although they are different lengths; remember SHA-1s can always be shortened as long as they remain unique). To update this in future you would run `git push` again to push any local changes to GitHub.

3.2.3 Cloning a remote/GitHub repository onto your local machine

It is useful to learn how to create a new Git repository locally and push it to GitHub. However, you will usually be downloading an existing repository to use as your local repository. This process of creating a new local repository from an existing remote repository is known as *cloning* a repository.

Some other version control systems (such as Subversion) will use the terminology of *checking out* a repository. The reasoning for this is that Subversion is a centralized version control system so when you download a repository locally you are only actually downloading the latest revision from the repository. With Git it is known as *cloning* because you are making a complete copy of that repository by downloading all commits, branches, tags; the complete history of the repository onto your local machine.

As you just pushed the entire contents of the local repository to GitHub let's remove the local repository and recreate it by cloning the repository on GitHub.

To remove the existing `GitInPracticeRedux` local repository and recreate it by cloning from GitHub:

1. Change to the directory where you want the new `GitInPracticeRedux` repository to be created e.g. `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeRedux`.
2. Run `rm -rf GitInPracticeRedux` to remove the existing `GitInPracticeRedux`

repository.

3. Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git`. The output should resemble:

Listing 3.3 Cloning a remote repository output

```
# git clone https://github.com/GitInPractice/GitInPracticeRedux.git

Cloning into 'GitInPracticeRedux'...   (1)
remote: Counting objects: 6, done.   (2)
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 6 (delta 0)
Unpacking objects: 100% (6/6), done.
Checking connectivity... done
```

(1) destination directory
(2) object preparation/transmission

The "destination directory (1)" is the folder in which the new `GitInPracticeRedux` local repository was created. The "object preparation/transmission (2)" can be safely ignored again.

You have cloned the `GitInPracticeRedux` remote repository and created a new local repository containing all its commits in `/Users/mike/GitInPracticeRedux`.

Cloning a repository has also created a new remote called `origin`. `origin` is the default remote and references the repository that the clone originated from (which is https://github.com/GitInPractice/GitInPracticeRedux.git in this case).

You can verify this remote has been created successfully by running `git remote --verbose`. The output should resemble:

Listing 3.4 Remote repositories output

```
# git remote --verbose

origin  https://github.com/GitInPractice/GitInPracticeRedux.git (fetch)   (1)
origin  https://github.com/GitInPractice/GitInPracticeRedux.git (push)   (2)
```

(1) fetch URL
(2) push URL

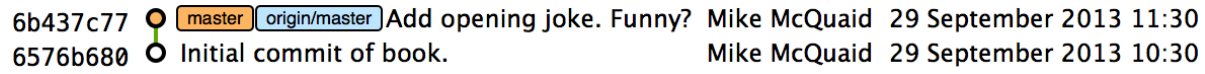

Figure 3.10 Local repository after `git clone`

Figure 3.10 shows the state of the repository after the `git push`. It is identical to the state after the `git push` in Figure 3.8. This shows that the clone was successful and the newly created local repository has the same contents as the deleted old local repository.

Now let's learn how to pull new commits from the remote repository.

3.2.4 `git pull` to obtain changes from another repository

`git pull` downloads the new commits from another repository and merges the remote branch into the current branch.

If you run `git pull` on the local repository you just see a message stating `Already up-to-date.`. `git pull` in this case contacted the remote repository, saw that there were no changes to be downloaded and let us know that it was up to date. This is expected as this repository has been pushed to but not updated since.

To test `git pull` let's create another clone of the same repository, make a new commit and `git push` it. This will allow downloading new changes with `git pull` on the original remote repository.

To create another cloned, local repository and push a commit from it:

1. Change to the directory where you want the new `GitInPracticeRedux` repository to be created e.g. `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxPushTest`.
2. Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git GitInPracticeReduxPushTest` to clone into the `GitInPracticeReduxPushTest` directory.
3. Change directory to the new Git repository e.g. `cd /Users/mike/GitInPracticeReduxPushTest/`.
4. Modify the `GitInPractice.asciidoc` file.
5. Run `git add GitInPractice.asciidoc`.
6. Run `git commit --message 'Improve joke comic timing.'`.
7. Run `git push`.

Now that you've pushed a commit to the `GitInPracticeRedux` remote on GitHub you can change back to your original repository and `git pull` from it. Keep the `GitInPracticeReduxPushTest` directory around as we'll use it later.

To pull new commits into the current branch on the local `GitInPracticeRedux` repository from the remote repository on GitHub:

1. Change directory to the original Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git pull`. The output should resemble:

Listing 3.5 Pulling new changes output

```
# git pull

remote: Counting objects: 5, done.    (1)
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux    (2)
   6b437c7..85a5db1  master     -> origin/master    (3)
Updating 6b437c7..85a5db1    (4)
Fast-forward    (5)
 GitInPractice.asciidoc | 5 +++--    (6)
 1 file changed, 3 insertions(+), 2 deletions(-)    (7)
```

(1) object preparation/transmission
(2) remote URL
(3) remote branch update
(4) local branch update
(5) merge type
(6) lines changed in file
(7) diff summary

The "object preparation/transmission (1)" can be safely ignored again. The "remote URL (2)" matches the remote repository URL we saw used for `git push`.

The "remote branch update (3)" shows how the state of the `origin` remote's `master` branch was updated and that this can be seen in `origin/master`. `origin/master` is a valid ref that can be used with tools such as `git diff` so `git diff origin/master` will show the differences between the current working tree state and the `origin` remote's `master` branch.

After `git pull` downloaded the changes from the other repository it merges the changes from the tracking branch into the current branch. In this case your `master` branch had the changes from the `master` branch on the remote `origin`

merged in. The "local branch update (4)" shows the changes that have been merged into the local `master` branch. You can see in this case the SHA-1s match those in the "remote branch update (3)". It has been updated to include the new commit (`85a5db1`). The "merge type (5)" was a *fast-forward merge* which means that no merge commit was made. Fast-forward merges will be fully explained in section 3.4.1.

NOTE **Why did a merge happen?**
It may be confusing that a merge has happened here. Didn't you just ask for the updates from that branch? You haven't created any other branches so why has a merge happened? In Git all remote branches (which includes the default `master` branch) are only linked to your local branches if the local branch is tracking the remote branch. The actual contents of the remote branches will always match the last seen state from the remote repository. If you want to just update the remote branches without merging to your local branches then you will use `git fetch`.

The "lines changed in file <6>" and "diff summary <7>" are similar to the output of `git commit` or `git diff` seen in Chapter 2. They are showing a summary of the changes that have been pulled into your `master` branch.

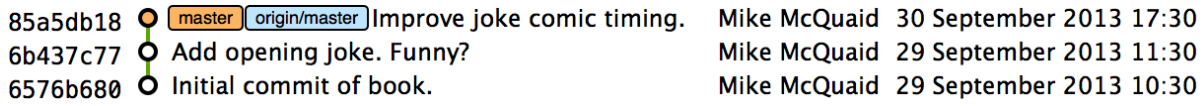

Figure 3.11 Local repository after `git pull`

You can see from Figure 3.11 that a new commit has been added to the repository and that both `master` and `origin/master` have been updated.

You have pulled the new commits from the `GitInPracticeRedux` remote repository into your local repository and Git has merged them into your `master` branch. Now let's learn how to download changes without apply them onto your master branch.

3.2.5 `git fetch` to get changes from a remote without modifying local branches

Remember that `git pull` does two actions: fetches the changes from a remote repository and merges them into the current branch. Sometimes you may wish to download the new commits from the remote repository without merging them into your current branch (or without merging them yet). To do this you can use the `git fetch` command. `git fetch` performs the fetching action of downloading the new commits but skips the merge step (which you can manually perform later).

To test `git fetch` let's use the `GitInPracticeReduxPushTest` local repository again to make another new commit and `git push` it. This will allow downloading new changes with `git fetch` on the original remote repository.

To push another commit from the `GitInPracticeReduxPushTest` repository:

1. Change directory to the `GitInPracticeReduxPushTest repository e.g. `cd /Users/mike/GitInPracticeReduxPushTest/`.
2. Modify the `GitInPractice.asciidoc` file.
3. Run `git add GitInPractice.asciidoc`.
4. Run `git commit --message 'Joke rejected by editor!'`.
5. Run `git push`.

Now that you've pushed another commit to the `GitInPracticeRedux` remote on GitHub you can change back to your original repository and `git fetch` from it. If you wish you can now delete the `GitInPracticeReduxPushTest` repository by running e.g. `rm -rf /Users/mike/GitInPracticeReduxPushTest/`

To fetch new commits to the local `GitInPracticeRedux` repository from the `GitInPracticeRedux` remote repository on GitHub:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git fetch`. The output should resemble:

Listing 3.6 Fetching new changes output

```
# git fetch

remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux
   85a5db1..07fc4c3  master     -> origin/master
```

1. object preparation/transmission
2. remote URL
3. remote branch update

The `git fetch` output is the same as the first part of the `git pull` output. The only difference here is the SHA-1s are different again as a new commit was downloaded. This is because `git fetch` is effectively half of what `git pull` is doing. If your `master` branch is tracking the `master` branch on the remote `origin` then `git pull` is directly equivalent to running `git fetch && git merge origin/master`.

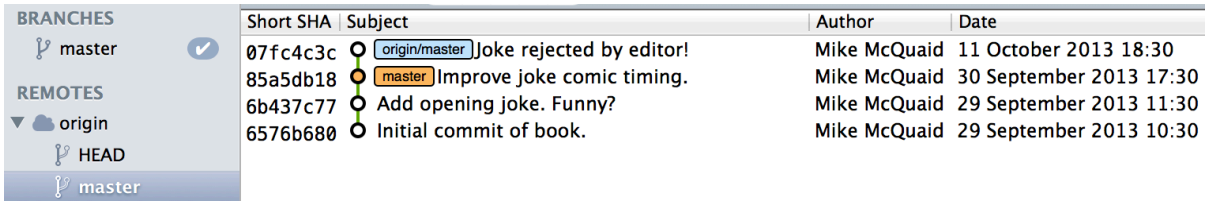

Figure 3.12 Remote repository after `git fetch`

You can see from Figure 3.12 that another new commit has been added to the repository but this time only `origin/master` has been updated but `master` has not. To see this you may need to select the `origin` remote and `master` remote branch in the GitX sidebar. This functionality is sadly not available in `gitk`.

You've fetched the new commits from the remote repository into your local repository but Git has not merged them into your `master` branch.

To clean up our local repository let's do another quick `git pull` to update the state of the `master` branch based on the (already fetched) `origin/master`.

To pull new commits into the current branch on the local `GitInPracticeRedux` repository from the remote repository on GitHub:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git pull`. The output should resemble:

Listing 3.7 Pull after fetch output

```
# git pull

Updating 85a5db1..07fc4c3   (1)
Fast-forward   (2)
 GitInPractice.asciidoc | 4 +---   (3)
 1 file changed, 1 insertion(+), 3 deletions(-)   (4)
```

(1) local branch update
(2) merge type
(3) lines changed in file
(4) diff summary

This shows the latter part of the first `git pull` output we saw. As there were no more changes to be fetched from the `origin` remote but the `master` branch had not been updated this `git pull` effectively the same as running `git merge origin/master`.

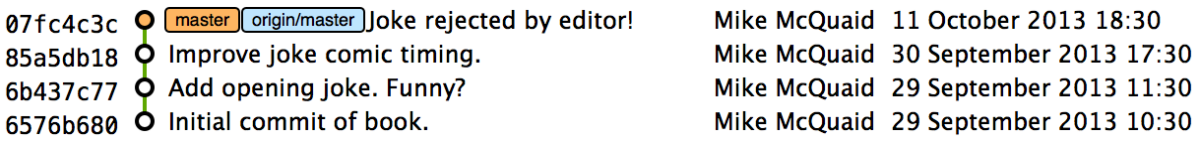

Figure 3.13 Local repository after `git fetch` then `git pull`

Figure 3.13 shows that the `master` branch has now been updated to match the `origin/master` latest commit once more.

We've talked about local branches and remote branches but haven't actually created any ourselves yet. Let's learn about how branches work and how to create them.

3.3 Branches

In Chapter 1 you learnt about *branches* and their usefulness in version control systems. They allow committing on multiple different tracks through history in parallel so you can make changes in one branch while currently ignoring all changes made in another branch. Let's learn how to use branches with Git.

Remember from Chapter 1 that branches are widely used in multiple-programmer projects but can still be useful for single-programmer projects. In this section we'll use the example of a branch for a new chapter for our book in our `GitInPracticeRedux` repository.

3.3.1 Create a new local branch from the current branch

The `git branch` command is used to create new branches in Git. A branch in Git (unlike other version control systems like Subversion) is simply a pointer to a single commit. This pointer is updated as you make more commits on that branch.

> **NOTE** **Can branches be named anything?**
> Branches cannot have two consecutive dots (`..`) anywhere in their name so `chapter..two` would be an invalid branch name and `git branch` will refuse to create it. This particular case is due to the special meaning of `..` for the `git diff` command which we saw in Chapter 2.

To create a new local branch named `chapter-two` from the current (`master`) branch:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git branch chapter-two`. There will be no output.

You can verify the branch was created by running `git branch` which should have the following output:

Listing 3.8 List branches output

```
# git branch

  chapter-two
* master
```

(1) new branch
(2) current branch

You can verify that the "new branch (1)" was created with the name you expect. The "current branch <2>" is indicated by the `*` prefix which indicates you are still on the master branch as before. `git branch` creates a new branch but does not change to it.

You have created a new local branch named `chapter-two` which currently points to the same commit as `master`.

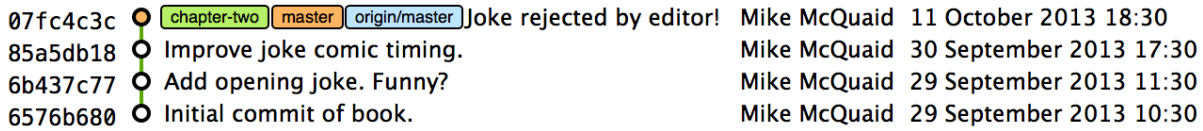

Figure 3.14 Local repository after `git branch chapter-two`

You can see from Figure 3.14 that there is a new, green branch label for the `chapter-two` branch. The current colors indicate:

- orange: the currently checked-out local branch
- green: a non-checked-out local branch
- blue: a remote branch

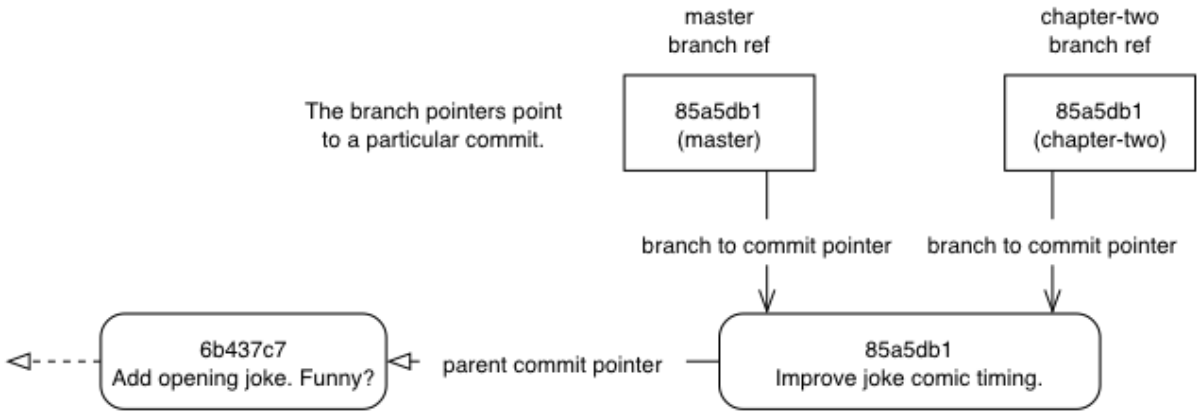

Figure 3.15 Branch pointers

Figure 3.15 shows how these two branch pointers point to the same commit.

You've seen `git branch` creates a local branch it does not change to it. To do that requires using `git checkout`.

3.3.2 Checkout a local branch

The `git checkout` command is used to change branches by checking out the contents of branches from the local repository into Git's working directory. The state of all the current files in the working directory will be replaced with the new state based on the revision that the new branch is currently pointing to.

To change to a local branch named `chapter-two` from the current (`master`) branch:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two`. The output should be `Switched to branch 'chapter-two'`.

You've checked out the local branch named `chapter-two` and moved from the `master` branch.

NOTE **Will `git checkout` overwrite any uncommitted changes?**
Make sure you've committed any changes on the current branch before checking out a new branch. If you do not do this `git checkout` will refuse to check out the new branch if there are changes in that branch to a file with uncommitted changes. If you wish to overwrite these uncommitted changes anyway you can force this with `git checkout --force`.

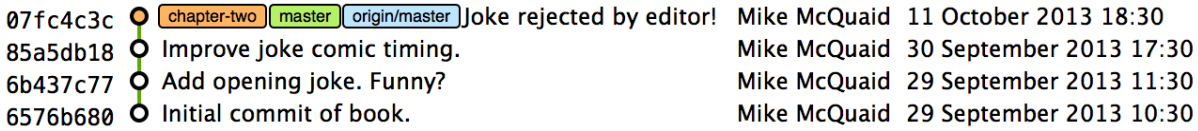

Figure 3.16 Local repository after `git checkout chapter-two`

The only difference between Figure 3.16 and Figure 3.14 is that the `chapter-two` branch is now orange and the `master` is green. Remember this means the `chapter-two` branch is currently checked out and `master` is not.

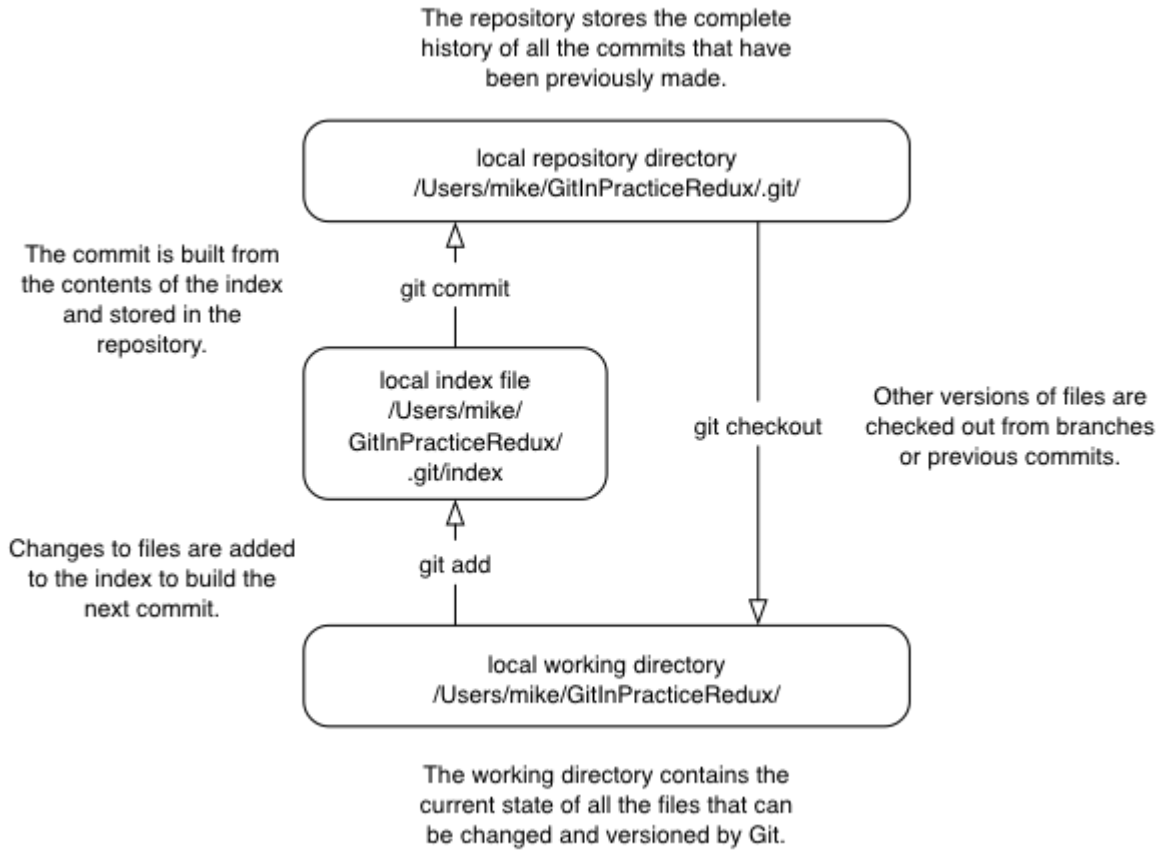

Figure 3.17 Git add/commit/checkout workflow

NOTE **Why do Subversion and Git use `checkout` to mean different things?**

As mentioned earlier some other version control systems (e.g. Subversion) use `checkout` to refer to the initial download from a remote repository but `git checkout` is used here to change branches. This may be slightly confusing until we look at Git's full remote workflow. Figure 3.17 shows Git's local workflow again. Under closer examination `git checkout` and `svn checkout` behave similarly; both check out the contents of a version control repository into the working directory but Subversion's repository is remote and Git's repository is local. In this case `git checkout` is requesting the checkout of a particular branch so the current state of that branch is checked out into the working directory.

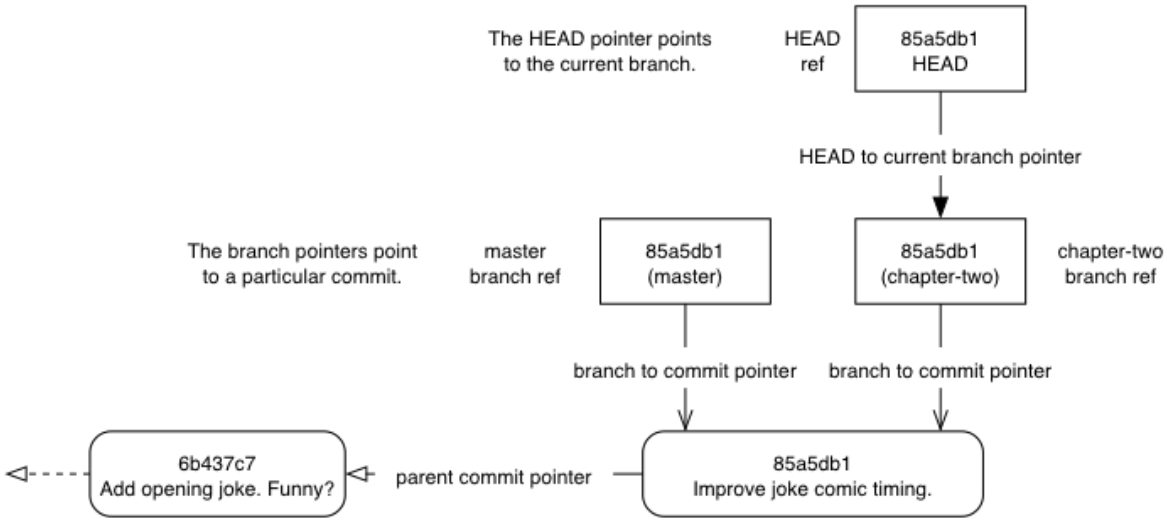

Figure 3.18 HEAD pointer with multiple branches

Afterwards the HEAD pointer (seen in Figure 3.18) is updated to point to the current, `chapter-two` branch pointer which in turn points to the top commit of that branch. The HEAD pointer moved from the `master` to the `chapter-two` branch when you ran `git checkout chapter-two`; setting `chapter-two` to be the current branch.

3.3.3 Pushing a local branch remotely

Now that you've created a new branch and checked it out it would be useful to push any new commits made to the remote repository. To do this requires using `git push` again.

To push the changes from the local `chapter-two` branch to create the remote

branch `chapter-two` on GitHub:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two` to ensure you are on the `chapter-two` branch.
3. Run `git push --set-upstream origin chapter-two`. The output should resemble:

Listing 3.9 Push and set upstream branch output

```
git push --set-upstream origin chapter-two

Total 0 (delta 0), reused 0 (delta 0)  (1)
To https://github.com/GitInPractice/GitInPracticeRedux.git  (2)
 * [new branch]      chapter-two -> chapter-two  (3)
Branch chapter-two set up to track remote branch
chapter-two from origin.  (4)
```

(1) object preparation/transmission
(2) remote URL
(3) local/remote branch
(4) set tracking branch

The output is much the same as the previous `git push` run except with the "local/remote branch (3)" and "set tracking branch (4)" have `chapter-two` as their branch name everywhere.

It may be interesting to note that the "object preparation/transmission (1)" (although still ignorable) shows that no new objects were sent. The reason for this is that the `chapter-two` branch still points to the same commit as the `master` branch; it's effectively a different name (or, more accurately, ref) pointing to the same commit. As a result there have been no more commit objects created and therefore no more were send up.

You have pushed your local `chapter-two` branch and created a new remote branch named `chapter-two` on the remote repository. Remember that now the local `chapter-two` branch is tracking the remote `chapter-two` branch so any future `git pull` or `git push` on the `chapter-two` branch will use the `origin` remote's `chapter-two` branch.

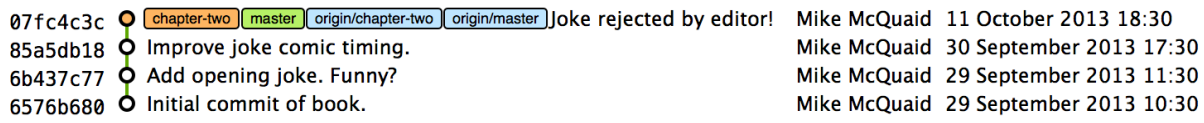

Figure 3.19 Local repository after `git push --set-upstream origin chapter-two`

As you'll hopefully have anticipated Figure 3.19 shows the addition of another remote branch named `origin/chapter-two`.

3.4 Merging

You've learnt how to create branches, push and pull them from remote repositories. This is useful for working on parallel tasks but at some point you'll want to merge work from one branch into another branch.

In this section we will actually commit to the `chapter-two` branch and then merge this work into the `master` branch and delete it afterwards.

NOTE **Why delete the branches?**

Sometimes branches in version control systems are kept around for a long time and sometimes they are very temporary. A long-running branch may be one that represents the version deployed to a particular server. A short-running branch may be a single bug fix or feature which has been completed. In Git once a branch has been merged the history of the branch is still visible in the history and the branch can be safely deleted as a merged branch is, at that point, just a ref to an existing commit in the history of the branch it was merged into.

3.4.1 Merging an existing branch into the current branch

Once you've reached a state on a branch where work is ready to be merged into another branch you will use the `git merge` to do so.

To make a commit on the local branch named `chapter-two` and merge this into into the `master` branch:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two` to ensure you are on the `chapter-two` branch.
3. Run `git commit --message 'Start Chapter 2.'`.
4. Run `git checkout master`.
5. Run `git merge chapter-two`. The output should resemble:

Listing 3.10 Merge branch output

```
# git merge chapter-two

Updating 07fc4c3..ac14a50  (1)
Fast-forward  (2)
 GitInPractice.asciidoc | 2 ++  (3)
 1 file changed, 2 insertions(+)  (4)
```

(1) local branch update
(2) merge type
(3) lines changed in file
(4) diff summary

The output may seem familiar from the `git pull` output. Remember this is because `git pull` actually does a `git fetch && git merge`.

The "local branch update (1)" shows the changes that have been merged into the local `master` branch. Note that the SHA-1 has been updated from the previous `master` SHA-1 (`07fc4c3`) to the current `chapter-two` SHA-1 (`ac14a50`).

The "merge type (2)" was a *fast-forward merge*. This means that no merge commit (a commit with multiple parents) was needed so none was made. The `chapter-two` commits were made on top of the `master` branch but no more commits had been added to the `master` branch before the merge was made. In Git's typical language: the merged commit (tip of the `chapter-two` branch) is a descendent of the current commit (tip of the `master` branch). If there had been another commit on the `master` branch before merging then this merge would have created a merge commit. If there had been conflicts between the changes made in both branches that could not automatically be resolved then a merge conflict would be created and need to be resolved.

The "lines changed in file <3>" and "diff summary <4>" are showing a summary of the changes that have been merged into your `master` branch from the `chapter-two` branch.

You have merged the `chapter-two` branch into the `master` branch. This brings the commit that was made in the `chapter-two` branch into the `master` branch.

Figure 3.20 Local repository after `git merge chapter-two`

You can see from Figure 3.20 that now the `chapter-two` and `master` branches point to the same commit once more.

3.4.2 Deleting a remote branch

We now want to delete the `chapter-two` branch from the remote repository now it is merged into the `master` branch and we do not want to make any more changes to it.

To push the current `master` branch and delete the branch named `chapter-two` on the remote `origin`:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout master` to ensure you are on the `master` branch.
3. Run `git push`.
4. Run `git push origin :chapter-two`. The output should resemble:

Listing 3.11 Delete remote branch output

```
# git push origin :chapter-two

To https://github.com/GitInPractice/GitInPracticeRedux.git
 - [deleted]         chapter-two
```

(1) remote URL
(2) deleted branch

The "deleted branch (2)" named `chapter-two` has been deleted from the remote repository at "remote URL (1)".

NOTE **What does the `:chapter-two` mean?**

The syntax here is somewhat unintuitive and hard to remember. What the `:chapter-two` is doing is better understood by examining a more verbose equivalent of a previous push command. Instead of `git push origin chapter-two` to create the branch initially you could have used `git push origin chapter-two:chapter-two`. What this differing syntax is saying is to push the local branch `chapter-two` (the first of the two) to the remote branch `chapter-two` (the second of the two). The first, local `chapter-two` branch reference can be omitted as it defaults to the current branch which you were already on. In the case of `git push origin :chapter-two` you are telling Git to push no branch or SHA-1 to the remote branch `chapter-two` which, as a branch is a pointer to a commit, is saying to remove the pointer and thus the branch.

You have deleted the `chapter-two` branch from the remote repository.

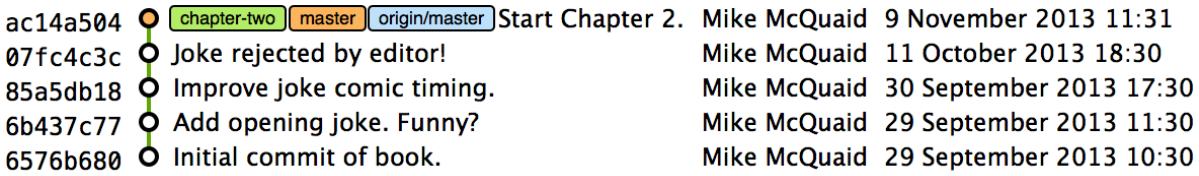

Figure 3.21 Local repository after `git push origin :chapter-two`

In Figure 3.21 you can see that the `origin/master` has been updated to the same commit as `master` and that `origin/chapter-two` has now been removed.

3.4.3 Deleting the current local branch after merging

The `chapter-two` branch has all its commits merged into the `master` branch and the remote branch deleted so the local branch can now be deleted too as we don't wish to make any more commits to it but instead will continue any work on Chapter 2 in the master branch.

To delete the local branch named `chapter-two`:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout master` to ensure you are on the `master` branch.
3. Run `git branch --delete chapter-two`. The output should be `Deleted branch chapter-two (was ac14a50).`

NOTE	**Why delete the remote branch before the local branch?** We deleted the remote branch first because we had pushed all the `chapter-two` changes in the `git push` on the `master` branch so it was no longer needed. Deleting it first means that the local branch can be safely deleted without Git being worried that the `chapter-two` local branch has changes that need to be pushed to the `origin/chapter-two` remote branch.

You've deleted the `chapter-two` branch from the local repository.

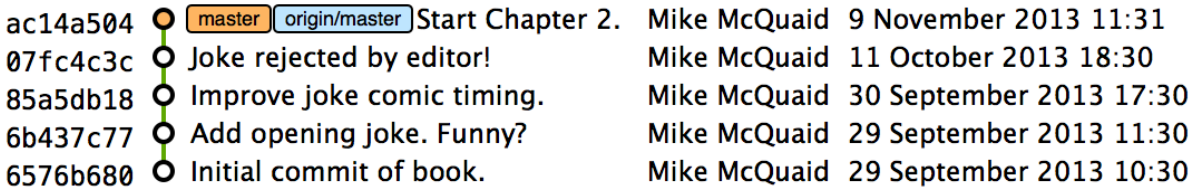

Figure 3.22 Local repository after `git branch --delete chapter-two`

Figure 3.22 shows the final state with all evidence of the `chapter-two` branch now removed (other than the commit message).

3.5 Summary

In this chapter you hopefully learned:

- How to signup for an account on GitHub and when to use free/private plans
- How to create a new GitHub repository and push your local repository to it
- How to clone an existing remote repository
- How to push and pull changes to/from a remote repository
- That fetching allows obtaining changes without modifying local branches
- That pulling is the equivalent to fetching then merging
- How to checkout local and remote branches
- How to merge branches and then delete from the local and remote repository

Now let's learn how to perform some more advanced interactions with files inside the GitHub working directory.

4 Filesystem Interactions

In this chapter you will learn about interacting with files and directories in your Git working directory by learning the following topics:

- How to rename, move or remove versioned files or directories
- How to tell Git to ignore certain files or changes
- How to delete all untracked or ignored files or directories
- How to reset all files to their previously committed state
- How to temporarily stash and reapply changes to files

When working with a project in Git you will sometimes wish to move, delete, change and/or ignore certain files in your working directory. You could mentally keep track of the state of which files and changes are important but this is not a sustainable approach. Instead, Git provides commands for performing filesystem operations for you.

Understanding the filesystem commands around Git will allow you to quickly perform these operations rather than being slowed down by Git's interactions.

Let's start with the most basic file operations: renaming or moving a file.

4.1 Rename or move a file

4.1.1 Background

Git keeps track of the changes to files in the working directory of a repository by their name. When you move or rename a file, Git does not see that a file was moved but that there is a file with a new filename and the file with the old filename was deleted (even if the contents remains the same). As a result of this renaming or moving a file in Git is essentially the same operation; both are telling Git to look for an existing file in a new location.

This may happen if you are working with tools (e.g. IDEs) which move files for you and aren't aware of Git (so don't give Git the correct move instruction).

Sometimes you will still need to manually rename or move files in your Git repository and you wish to preserve the history of the files after the rename or move operation. As you learnt in Chapter 1, readable history is one of the key benefits of a version control system so it's important to avoid losing it whenever possible. If a file has had 100 small changes made to it with good commit messages it would be a shame to undo all that work just by renaming or moving a file.

4.1.2 Problem

You wish to rename a previously committed file in your Git working directory named `GitInPractice.asciidoc` to `01-IntroducingGitInPractice.asciidoc` and commit the newly renamed file.

4.1.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git mv GitInPractice.asciidoc 01-IntroducingGitInPractice.asciidoc`. There will be no output.
3. Run `git commit --message 'Rename book file to first part file.'` The output should resemble:

Listing 4.1 Renamed commit output

```
# git commit --message 'Rename Chapter 1 file.'

[master c6eed66] Rename book file to first part file.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename GitInPractice.asciidoc =>
  01-IntroducingGitInPractice.asciidoc (100%)
```

1 commit message
2 no insertions/deletions
3 old new filename

You have renamed `Chapter1.asciidoc` to `01-FirstChapter.asciidoc` and committed it.

4.1.4 Discussion

Moving and renaming files in version control systems rather than deleting and recreating them is done to preserve their history. For example, when a file has been moved into a new directory you will still be interested in the previous versions of the file before it was moved. In Git's case it will try and autodetect renames or moves on `git add` or `git commit`; if a file is deleted and new file is created which has a majority of lines in common then Git will automatically detect the file was moved and `git mv` is not necessary. Despite this handy feature it's good practice to use `git mv` so you don't need to wait for a `git add` or `git commit` for Git to be aware of the move and to have consistent behavior across different versions of Git (which may have differing move autodetection behaviour).

After running `git mv` the move or rename will be added to Git's index staging area which, if you remember from Chapter 2, means that the change has been staged for inclusion in the next commit.

It's also possible to rename files, directories or symlinks and move files, directories or symlinks into other directories inside the Git repository using the `git mv` command and the same syntax as above. If you wish to move files into or out of a repository you must use a different, non-Git command (such as a Unix `mv` command).

NOTE **What if the new filename already exists?**

If the filename that you move to already exists you will need to use the `git mv -f` (or `--force`) option to state to Git that you wish to overwrite whatever file is at the destination. If the destination file has not already been added or committed to Git that it will not be possible to retrieve the contents if you erroneously asked Git to overwrite it.

4.2 Remove a file

4.2.1 Background

Removing files from version control systems requires, like moving/renaming, not just performing the filesystem operation as usual but notifying Git and committing the file. Almost any version-controlled project will see you wanting to remove some files at some point so it's essential to know how to do so. Removing version-controlled files is also more safe than removing non-version-controlled files as, even after removal, the files will still exist in the history.

Sometimes tools that don't interact with Git may remove files for you and require you to manually indicate to Git that you wish these files to be removed.

For testing purposes let's create and commit a temporary file to be removed: 1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/` 2. Run `echo Git Sandwich > GitInPracticeReviews.tmp`. This will create a new file named `GitInPracticeReviews.tmp` with the contents "Git Sandwich". 3. Run `git add GitInPracticeReviews.tmp`. 4. Run `git commit --message 'Add review temporary file.'`

4.2.2 Problem

You wish to remove a previously committed file named `GitInPracticeReviews.tmp` in your Git working directory and commit the removed file.

4.2.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git rm GitInPracticeReviews.tmp`.
3. Run `git commit --message 'Remove unfavourable review file.'` The output should resemble:

Listing 4.2 Removed commit output

```
# git rm GitInPracticeReviews.tmp

rm 'GitInPracticeReviews.tmp'

# git commit --message 'Remove Chapter 2 temporary file.'

[master 06b5eb5] Remove unfavourable review file.
 1 file changed, 1 deletion(-)
 delete mode 100644 GitInPracticeReviews.tmp
```

1. commit message
2. 1 line deleted
3. deleted filename

You have removed `GitInPracticeReviews.tmp` and committed it.

4.2.4 Discussion

Git will only interact with the Git repository when you explicitly give it commands which is why when you remove a file Git does not automatically run `git rm` command. The `git rm` command is not just indicating to Git that you wish for a file to be removed but also (like `git mv`) that this removal should be part of the next commit.

If you wish to see a simulated run of `git rm` without actually removing the requested file then you can use `git rm -n` (or `--dry-run`). This will print the output of the command as if it were running normally and indicate success or failure but without actually removing the file.

To remove a directory and all the files and subdirectories within it you will need to use `git rm -r` (where the `-r` stands for *recursive*). When run this will delete the directory and all files under it. This is combined well with `--dry-run` if you want to see what would be removed before removing it.

NOTE **What if a file has uncommitted changes?**
If a file has uncommited changes but you still wish to remove it you will need to use the `git rm -f` (or `--force`) option to indicate to Git you wish to remove it before committing the changes.

4.3 Resetting files to the last commit

4.3.1 Background

There are times when you have made some changes to files in the working directory but you do not wish to commit these changes.

Perhaps you added debugging statements to files and have now committed a fix so want to reset all of the files that have not been committed to their last committed state (on the current branch).

4.3.2 Problem

You wish to reset the state of all the files in your working directory to their last committed state.

4.3.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc` to append "EXTRA" to the end of `01-IntroducingGitInPractice.asciidoc`.
3. Run `git reset --hard`. The output should resemble:

Listing 4.3 Hard reset output

```
# git reset --hard

HEAD is now at 06b5eb5 Remove unfavourable review file.
```

(1) **Reset commit**

You have reset the Git working directory to the last committed state.

4.3.4 Discussion

The `--hard` argument signals to `git reset` that you wish it to reset both the index staging area and the working directory to the state of the previous commit on this branch. If run without an argument it defaults to `git reset --mixed` which will reset the index staging area but not the contents of the working directory. In short, `git reset --mixed` only undoes `git add`s but `git reset --hard` undoes `git add`s and all file modifications.

`git reset` will be used to perform more operations (including those that alter history) in Chapter 7.

WARNING **Dangers of using `git reset --hard`**

Care should be used with `git reset --hard`; it will immediately and without prompting remove all your uncommitted changes to any file in your working directory. This is probably the command which has caused me more regret than any other; I've typed it accidentally and removed work I hadn't intended to. Safer options may be to only reset files you have open in an editor (so you can undo) or use Git's stash functionality instead.

4.4 Delete untracked files

4.4.1 Background

When working in a Git repository some tools may output undesirable files into your working directory.

Some text editors may use temporary files, operating systems may write thumbnail cache files or programs may write crash dumps. Alternatively, sometimes there may be files that are desirable but you do not wish to check them into your version control system and wish to remove them and build clean versions (although this is generally better handled by *ignoring* these files as in Section 4.5).

When you wish to remove these files you could remove them manually but it's easier to ask Git to do so as it already knows which files in the working directory are versioned and which are *untracked*.

For testing purposes let's create a temporary file to be removed: 1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/` 2. Run `echo Needs more cowbell > GitInPracticeIdeas.tmp`. This will create a new file named `GitInPracticeIdeas.tmp` with the contents "Needs more cowbell".

4.4.2 Problem

You wish to remove an untracked file named `GitInPracticeIdeas.tmp` from a Git working directory.

4.4.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git clean --force`. The output should resemble:

Listing 4.4 Force cleaned files output

```
# git clean --force

Removing GitInPracticeIdeas.tmp                    (1) removed file
```

You have removed `GitInPracticeIdeas.tmp` from the Git working directory.

4.4.4 Discussion

`git clean` requires the `--force` argument because this command is potentially dangerous; with a single command you can remove many, many files very quickly. Remember in Chapter 1 we learnt that accidentally losing any file or change committed to a version control system is very hard (and in Git, nearly impossible). This is the opposite situation; `git clean` will happily remove thousands of files very quickly which cannot be easily recovered (unless backed up through another mechanism).

To make `git clean` a bit safer you can preview what will be removed before doing so by using `git clean -n` (or `--dry-run`). This behaves like the `git rm --dry-run` in that it prints the output of the removals that would be performed but does not actually do so.

To remove untracked directories as well as untracked files you can use the `-d` (which stands for "directory") parameter.

4.5 Ignore files

4.5.1 Background

As discussed in the previous section, sometimes working directories will contain files which are *untracked* by Git and you do not wish to add them to the repository.

Sometimes these files are one-off occurrences; you accidentally copy a file to the wrong directory and wish to delete it. Usually, however, they are the product of some software (e.g. the software stored in the version control system or some part of your operating system) putting files into the working directory of your version control system.

You could just `git clean` these files each time but that would rapidly become tedious. Instead we could tell Git to ignore them so it never complains about these files being untracked and you do not accidentally add them to commits.

4.5.2 Problem

You wish to ignore all files with the extension `.tmp` in a Git repository.

4.5.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo \*.tmp > .gitignore`. This will create a new file named `.gitignore` with the contents "*.tmp".
3. Run `git add .gitignore` to add `.gitignore` to the index staging area for the next commit. There will be no output.
4. Run `git commit --message='Ignore .tmp files.'` The output should resemble:

Listing 4.5 Ignore file commit output

```
# git commit --message='Ignore .tmp files.'

[master 0b4087c] Ignore .tmp files.
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

1. **commit message**
2. **1 line added**
3. **created filename**

You have added a `.gitignore` file with instructions to ignore all `.tmp` files in the Git working directory.

4.5.4 Discussion

A good and widely-held principle for version control systems is to avoid committing *output files* to a version control repository. Output files are those that are the created from input files that are stored within the version control repository.

For example, I may have a `hello.c` file which is compiled into `hello.o` object file. The `hello.c` *input file* should be committed to the version control system but the `hello.o` *output file* should not.

Committing `.gitignore` to the Git repository makes it easy to build up lists of expected output files so that they can be shared between all the users of a repository and not accidentally committed.

Let's try and add an ignored file.

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `touch GitInPractiseGoodIdeas.tmp`. This will create a new, empty file named `GitInPractiseGoodIdeas.tmp`.
3. Run `git add GitInPractiseGoodIdeas.tmp`. The output should resemble:

Listing 4.6 Trying to add an ignored file

```
# git add GitInPractiseGoodIdeas.tmp

The following paths are ignored by one of your .gitignore files:
GitInPractiseGoodIdeas.tmp  (1)
Use -f if you really want to add them.
fatal: no files added  (2)
```

(1) ignored file
(2) error message

The "ignored file (1)" `GitInPractiseGoodIdeas.tmp` was not added as its addition would contradict your `.gitignore` rules. As no files were added the "error message (2)" was printed.

This interaction between `.gitignore` and `git add` is particularly useful when adding subdirectories of files and directories which may contain files that should to be ignored. `git add` will not add these files but will still successfully add all other that should not be ignored.

4.6 Delete ignored files

4.6.1 Background

When files have been successfully ignored by the addition of a `.gitignore` file you will sometimes with to delete them all.

For example, you may have a project in a Git repository which compiles input files (e.g. `.c` files) into output files (e.g. `.o` files) and wish to remove all of these output files from the working directory to perform a new build from scratch.

Let's create some temporary files that can be removed.

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `touch GitInPractiseFunnyJokes.tmp GitInPractiseWittyBanter.tmp`.

4.6.2 Problem

You wish to delete all ignored files from a Git working directory.

4.6.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git clean --force -X`. The output should resemble:

Listing 4.7 Force clean of ignored files output

```
# git clean --force -X

Removing GitInPractiseFunnyJokes.tmp
Removing GitInPractiseWittyBanter.tmp
```

(1) removed file

You have removed all ignored files from the Git working directory.

4.6.4 Discussion

The `-X` argument specifies that `git clean` should remove *only* the ignored files from the working directory. If you wish to remove the ignored files *and* all the untracked files (as `git clean --force` would do) you can instead use `git clean -x` (note the `-x` is lowercase rather than uppercase).

The specified arguments can be combined with the others discussed in Section 4.4.4. For example, `git clean -xdf` would remove all untracked or ignored files (`-x`) and directories (`-d`) from a working directory. This will remove all files and directories for a Git repository that were not previously committed. Please take care when running this; there will be no prompt and all the files will be quickly deleted.

Often `git clean -xdf` will be run after `git reset --hard`; this means that you will have reset all files to their last-committed state and removed all uncommitted files. This gets you a clean working directory; no added files or changes to any of those files.

4.7 Temporarily stash some changes

4.7.1 Background

There are times when you may find yourself working on a new commit and want to temporarily undo your current changes but redo them at a later point.

Perhaps there was an urgent issue that means you need to quickly write some code and commit a fix. In this case you could make a temporary branch and merge it in later but this would add a commit to the history that may not be necessary.

Instead you could *stash* your uncommitted changes to store them temporarily away and then be able to e.g. change branches, pull changes etc. without needing to worry about these changes getting in the way.

4.7.2 Problem

You wish to stash all your uncommitted changes for later retrieval.

4.7.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc`.
3. Run `git stash save`. The output should resemble:

Listing 4.8 Stashing uncommitted changes output

```
# git stash save

Saved working directory and index state WIP on master:
36640a5 Ignore .tmp files.
HEAD is now at 36640a5 Ignore .tmp files.
```

❶ **Current commit**

You have stashed your uncommitted changes.

4.7.4 Discussion

`git stash save` actually creates a temporary commit with a pre-populated commit message and then returns your current branch to the state before the temporary commit was made. It's possibly to access this commit directly but you should only do so through `git stash` to avoid confusion.

You can see all the stashes that have been made by running `git stash list`. The output will resemble:

Listing 4.9 List of stashes

```
stash@{0}: WIP on master: 36640a5 Ignore .tmp files.
```

❶ **Stashed commit.**

This shows the single stash that you made. You can access it using the `ref stash@{0}` so e.g. `git diff stash@{0}` will show you the difference between the working directory and the contents of that stash.

If you save another stash then it will become `stash@{0}` and the previous

stash will become `stash@{1}`. This is because the stashes are stored on a *stack* structure. A stack structure is best thought of as being like a stack of plates. You add new plates on the top of the existing plates and if you remove a single plate you will take it from the top. Similarly when you run `git stash` the new stash will be added will be added to the top (i.e. become `stash@{0}`) and the previous stash will no longer be at the top (i.e. become `stash@{1}`).

NOTE	**Do you need to use `git add` before `git stash`**
	No, `git add` is not needed. `git stash` will stash your changes whether or not they have been added to the index staging area by `git add` or not.

Does `git stash` work without the `save` argument?

If `git stash` is run with no "save" argument it performs the same operation; the "save" argument is not needed. I've used it in the examples as it's more explicit and easier to remember.

4.8 Reapply stashed changes

4.8.1 Background

When you have stashed your temporary changes and performed whatever the operations that required a clean working directory (e.g. perhaps fixed and committed the urgent issue) you will want to reapply the changes (as otherwise you could have just run `git reset --hard`). When you've checked out the correct branch again (which may differ from the original branch) you can request for the changes to be taken from the stash and applied onto the working directory.

4.8.2 Problem

You wish to pop the last changes from the last `git stash save` into the current working directory.

4.8.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git stash pop`. The output should resemble:

Listing 4.10 Reapply stashed changes output

```
# git stash pop

# On branch master                                                    (1)
# Changes not staged for commit:                                      (2)
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#    modified:   01-IntroducingGitInPractice.asciidoc
#
no changes added to commit (use "git add" and/or "git commit -a")     (3)
Dropped refs/stash@{0} (f7e39e2590067510be1a540b073e74704395e881)     (4)
```

(1) current branch output
(2) begin status output
(3) end status output
(4) stashed commit

You have reapplied the changes from the last `git stash save`.

4.8.4 Discussion

When running `git stash pop` the top stash on the stack (i.e. `stash@{0}`) will be applied to the working directory and removed from the stack. If there is a second stash in the stack (`stash@{1}`) then it will now be at the top (i.e. become `stash@{1}`). This means if you run `git stash pop` multiple times it will keep working down the stack until no more stashes are found and it outputs `No stash found.`.

If you wish to apply an item from the stack multiple times (e.g. perhaps on multiple branches) then you can instead use `git stash apply`. This applies the stash to the working tree as `git stash pop` does but keeps the top stack stash on the stack so it can be run again to reapply.

4.9 Clear stashed changes

4.9.1 Background

You may have stashed changes with the intent of popping them later but then realize that you no longer wish to do so. You know that the changes in the stack are now unnecessary so wish to get rid of them all. You could do this by popping each change off the stack and then deleting it but it would be good if there was a command that allowed you to do this in a single step. Thankfully, `git stash clear` allows you to do just this.

4.9.2 Problem

You wish to clear all previously stashed changes.

4.9.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git stash clear`. There will be no output.

You have cleared all the previously stashed changes.

4.9.4 Discussion

WARNING	**No prompt for `git stash clear`** Clearing the stash will be done without a prompt and will remove every previous item from the stash so be careful when doing so. Cleared stashes cannot be recovered.

4.10 Assume files are unchanged

4.10.1 Background

Sometimes you may wish to make changes to files but have Git ignore the specific changes you have made so that operations such as `git stash` and `git diff` ignore these changes. In these cases you could just ignore them yourself or stash them elsewhere but it would be ideal to be able to tell Git to ignore these particular changes.

I've found myself in a situation in the past where I'm wanting to test a Rails configuration file change for a week or two while continuing to do my normal

work. I don't want to commit it because I don't want it to apply to servers or my coworkers but I do want to continue testing it while I make other commits rather than changing to a particular branch each time.

4.10.2 Problem

You wish for Git to assume there have been no changes made to `01-IntroducingGitInPractice.asciidoc`.

4.10.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git update-index --assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

Git will ignore any changes made to `01-IntroducingGitInPractice.asciidoc`.

4.10.4 Discussion

When you run `git update-index --assume-unchanged` Git sets a special flag on the file to indicate that it should not be checked for any changes that have been made. This can be useful to temporarily ignore changes made to a particular file when looking at `git status` or `git diff` but also to tell Git to avoid checking a file that is particular huge and/or slow to read. This is not normally a problem on normal filesystems on which Git can quickly query if a file is modified by checking the "file modified" timestamp (rather than having to read the entire file and compare it).

The `git update-index` command has other complex options but we will only cover those around the "assume" logic. The rest of the behavior is better accessed through the `git add` command; a higher-level and more user-friendly way of modifying the state of the index.

4.11 List assumed unchanged files

4.11.1 Background

When you have told Git to assume there are no changes made to particular files it can be hard to remember which files were updated. In this case you may end up modifying a file and wondering why Git does not seem to want to show you these changes. Additionally, you could forget that you had made these changes at all and be very confused as to why the state in your text editor does not seem to match the state that Git is seeing.

4.11.2 Problem

You wish for Git to list all the files that it has been told to assume haven't changed.

4.11.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git ls-files -v`. The output should resemble:

Listing 4.11 Assumed unchanged files listing output

```
# git ls-files -v

H .gitignore
H 01-IntroducingGitInPractice.asciidoc
```

(1) committed file
(2) assumed unchanged file

4.11.4 Discussion

Like `git update-index`, `git ls-files -v` is a low level command that you will typically not run often. `git ls-files` without any arguments lists the files in the current directory that Git knows about but the `-v` argument means that it is followed by tags which indicate file state. I will not detail these here as it is not important to understand the tags beyond that in Listing 4.9 the "committed files (1)" are indicated by an uppercase `H` tag at the beginning of the line and the "assumed unchanged file (2)" has a lowercase `h` tag.

Rather than reading through the output for this command you could instead run `git ls-files -v | grep '^[hsmrck?]' | cut -c 3-`. This makes use of Unix pipes where the output of each command is passed into the next and modified.

`grep '^[hsmrck?]'` filters the output filenames to only show those that

begin with any of the lowercase `hsmrck?` characters.

`cut -c 3-` filters the first two characters of each of the output lines so e.g. `h` followed by a space in the above example.

With these combined the output should resemble:

Listing 4.12 Assumed unchanged files output

```
# git ls-files -v | grep '^[hsmrck?]' | cut -c 3-

01-IntroducingGitInPractice.asciidoc
```

(1) assumed unchanged file

NOTE **How do pipes, `grep` and `cut` work?**

Do not worry if you don't understand quite how Unix pipes, `grep` or `cut` work; this book is about Git rather than shell scripting after all! Feel free to just use the above command as-is as a quick way of listing files that are assumed to be unchanged.

4.12 Stop assuming files are unchanged

4.12.1 Background

Usually telling Git to assume there have been no changes made to a particular file is a temporary option; if you have to keep files changed long-term they should probably be committed. At some point you will wish to tell Git to monitor any changes that are made to these files once more.

With the example I gave previously in Section 4.10 eventually the Rails configuration file change I had been testing was deemed to be successful enough that I wanted to commit it so my coworkers and the servers could use it. If I merely used `git add` to make a new commit then the change would not show up so I had to stop Git ignoring this particular change before I could make a new commit.

4.12.2 Problem

You wish for Git to stop assuming there have been no changes made to `01-IntroducingGitInPractice.asciidoc`.

4.12.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git update-index --no-assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

Git will notice any current or future changes made to `01-IntroducingGitInPractice.asciidoc`.

4.12.4 Discussion

Once you tell Git to stop ignoring changes made to a particular file then all commands such as `git add` and `git diff` will start behaving normally on this file again.

4.13 Summary

In this chapter you hopefully learned:

- How to use `git mv` to move or rename files
- How to use `git rm` to remove files or directories
- How to use `git clean` to remove untracked or ignored files or directories
- How and why to create a `.gitignore` file
- How to (carefully) use `git reset --hard` to reset the working directory to the previously committed state
- How to use `git stash` to temporarily store and retrieve changes
- How to use `git update-index` to tell Git to assume files are unchanged

Now let's learn how to visualize history in a Git repository in different formats.