O'REILLY®

Head First Git

A Learner's Guide to Understanding Git from the Inside Out

Raju Gandhi

A Brain-Friendly Guide

Head First Git

A Learner’s Guide to Understanding Git from the Inside-Out

Raju Gandhi

Head First Git

by Raju Gandhi

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Melissa Duffield and Sarah Grey

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

June 2022: First Edition

Revision History for the First Edition

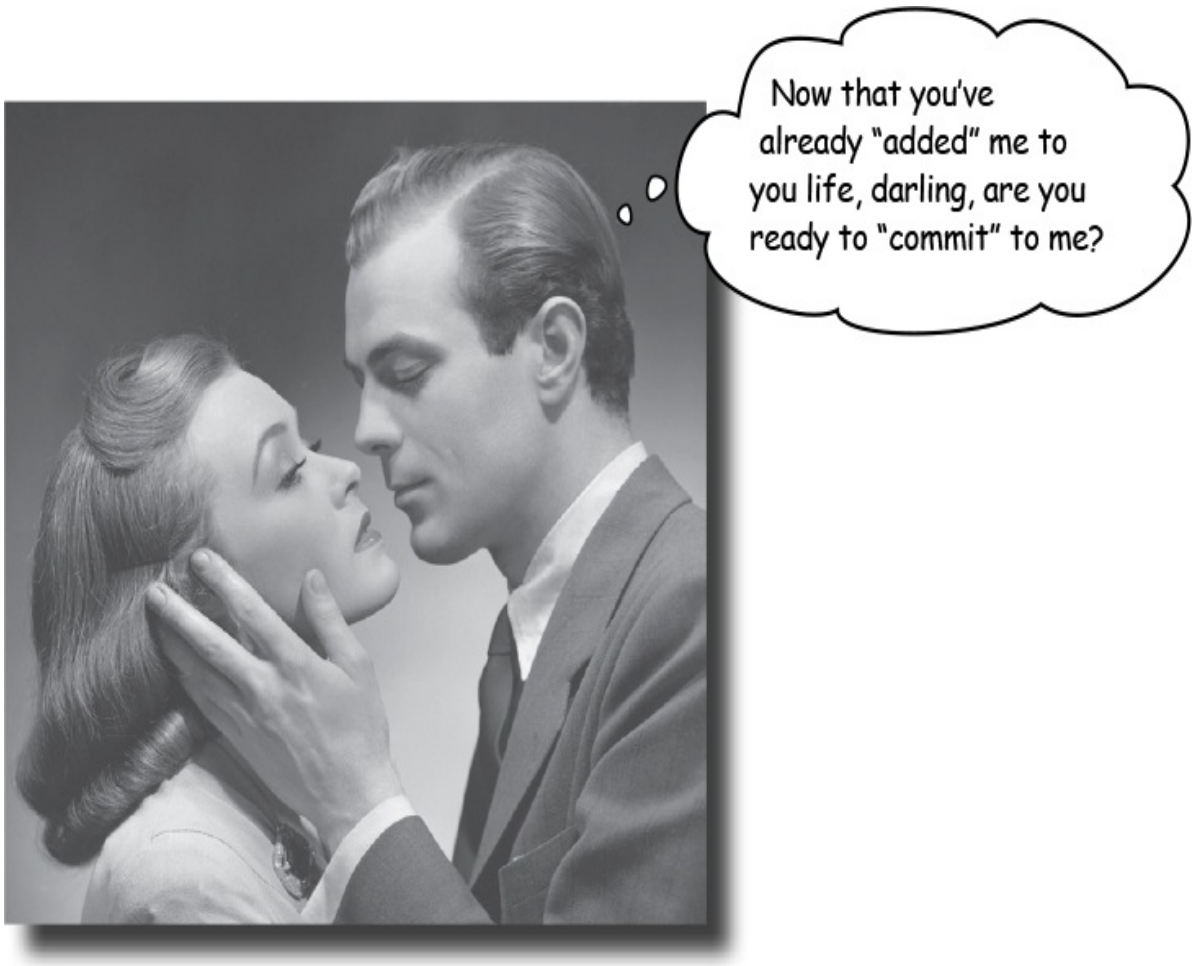
- 2021-03-26: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492092513 for release details.

978-1-492-09251-3

[FILL IN]

Chapter 1. Beginning Git: Get Going with Git



You need version control. Every software project begins with an idea, implemented in source code. These files are the magic that power our applications, so we must be sure to treat them with care. We want to be sure that we keep them safe, retain history of changes, and attribute credit (or blame!) to the rightful authors. We also want to allow for seamless collaboration between multiple team members.

And we want all this in a tool that stays out of our way, springing into action only at the moment of our choosing.

Does such a ***magical tool*** even exist? If you're reading this, you might have guessed the answer. Its name is Git! Developers and organizations around the word love Git. So what is it that makes Git so popular?

Why we need version control

You might have played video games that take more than one sitting to complete. As you progress through the game, you win and lose some battles, you might acquire some weaponry or an army. Every so often you might to try more than once to finish a particular challenge. Many games allow you to save your progress. So now, say you've just slayed the fire dragon and next on the agenda is saving the princess and collecting the massive treasure trove.

You decide, just to be safe, to save your progress, and then continue the adventure. This creates a "snapshot" of the game as it stands right now. The good news is that now, even if you meet an untimely demise when you run into the wretched acid-spitting lizards, you won't have to go back to square one. Instead, you simply reload the snapshot you took earlier, and try again. Fiery dragons be gone!

Version control allows you to do the same with your work—it gives you a way to save your progress. You can do a little bit of work, save your progress, and continue working. Now, even if you make a mistake or perhaps, you are not happy with the way you solved a specific problem. You can save your work, and then try a different tack. If you like the new approach you just save your progress again, or you just go back to the old way.

And there's more. Git allows you to confidently collaborate with your fellow developers over the same set of files, without stepping on each others toes. We will get into details about this in later chapters, but for now it should be enough to know this.

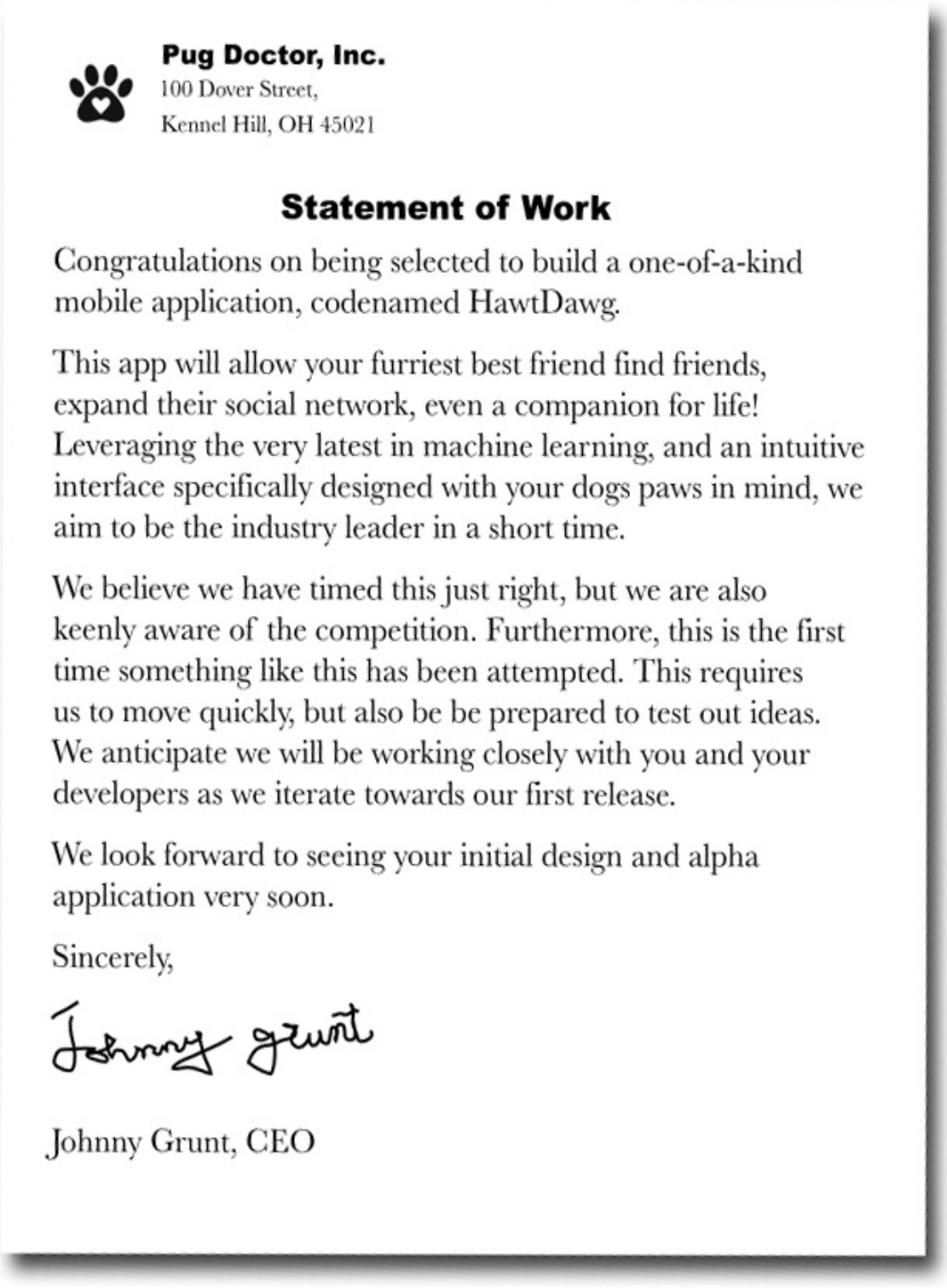
You can think of Git as your memory bank, safety net, and collaboration platform all built into one!

Understanding version control, and Git in particular—understanding what it is capable of, and the effect it has on how we develop software can help make us really, and we mean, *really,* productive.

Congratulations!

Your company has just been awarded the contract to build HawtDawg—the first-ever dating app for humans furriest best friend. However, it’s a dog-eat-dog world out there, and with the competition sniffing around, we don’t have much time to waste!


It's hard to swipe right. I wish there was an app that would accommodate my paws. Sigh ...



Pug Doctor, Inc.
100 Dover Street,
Kennel Hill, OH 45021

Statement of Work

Congratulations on being selected to build a one-of-a-kind mobile application, codenamed HawtDawg.

This app will allow your furriest best friend find friends, expand their social network, even a companion for life! Leveraging the very latest in machine learning, and an intuitive interface specifically designed with your dogs paws in mind, we aim to be the industry leader in a short time.

We believe we have timed this just right, but we are also keenly aware of the competition. Furthermore, this is the first time something like this has been attempted. This requires us to move quickly, but also be be prepared to test out ideas. We anticipate we will be working closely with you and your developers as we iterate towards our first release.

We look forward to seeing your initial design and alpha application very soon.

Sincerely,

Johnny Grunt

Johnny Grunt, CEO

Cubicle Conversation

Marge: Yes. We should consider using a version control system.

Sangita: I have heard of version control systems, though I have never had a chance to use one. But we don’t exactly have a lot of time here.

Marge: Getting started with Git is super easy. You just create a Git repository and you are off to the races.

Sangita: I create a what now?

Marge: A Git repository is a folder that is managed by Git. Let me take a step

back. You are going to need to house all the files for this project somewhere on your computer, right?

Sangita: I prefer to keep all relevant files pertaining to my project, including source, build and documentation, in one folder. That way, they are easy to find.

Marge: Great! Once you create that folder, you use Git to initialize a repository inside the folder. It's that simple.

Sangita: And what does that do?.

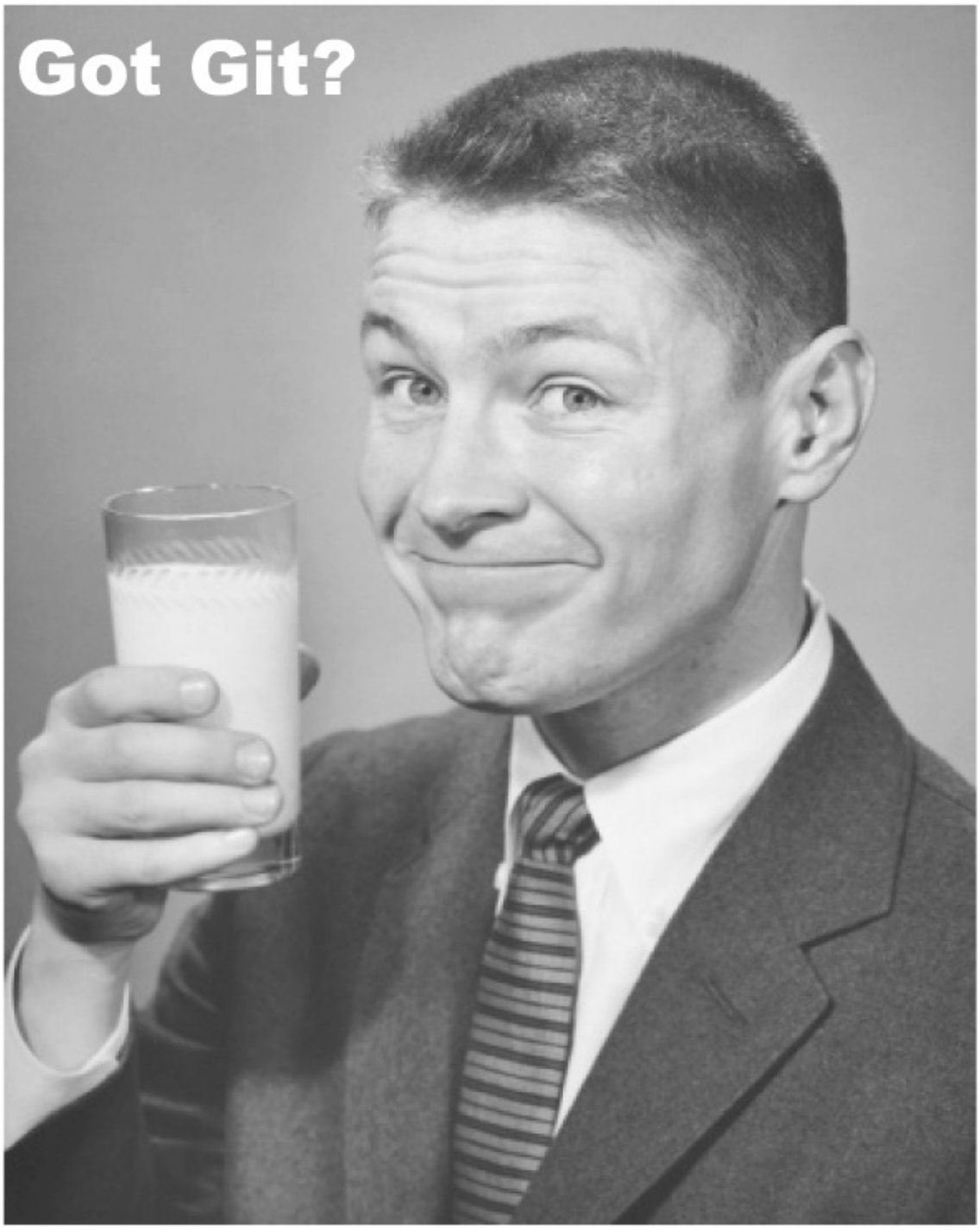
Marge: Well, whenever you start a new project that you want to manage with Git, you run a Git command that readies the folder so that you can start to use other Git commands inside that folder. Think of it as turning the key in your car to start the engine. It's the first step so you can now start to use your car.

Sangita: Ah! That makes sense.

Marge: Yep. It's just one command, and now your folder is "Git enabled". Just like kick-starting your engine—you can now put your project in gear.

Sangita: Got it.

Marge: Hit me up if you need something. I will be right here if you need me.



We're not going to get much further if you haven't installed Git yet. If you haven't taken the time to install Git, now is the time. Head back to the section titled "You're going to have to install Git" in the introduction to get started.

Even if you have Git installed, it will help to catch up with a new version of Git just to be sure that everything we discuss in this books works as expected.

Start your engines ...

Consider any project you have worked on; it typically involves one or more files —these may be source code files, documentation files, build scripts, what-have-you. If we want to manage these files with Git, then the first step is to create a Git repository.

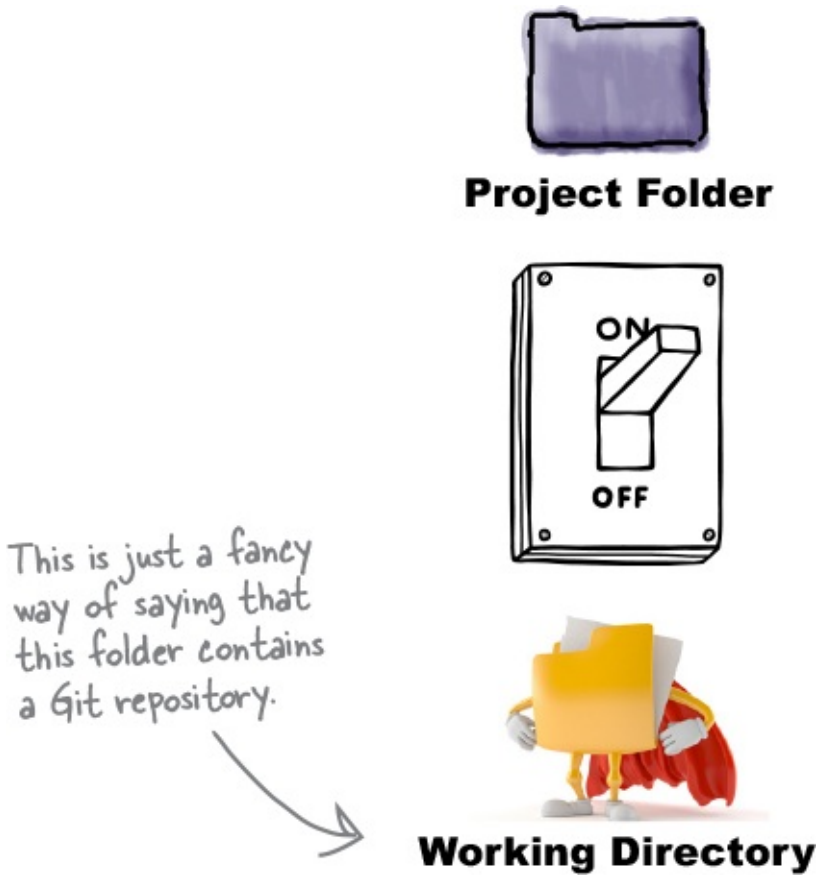
So what exactly is a Git repository? Recall that one reason to use a version control system is so we can save the snapshots of our work periodically. Of course, Git needs a place to store these snapshots. That place would be in the Git repository.

The next question is—where does this repository live? Typically we tend to keep

all the files for a project in one folder. If we are going to use Git as our version control system for that project, we first create a repository *within* that folder so that Git has a place to store our snapshots. Creating a Git repository involves running the **`git init`** command inside the top folder of your project.

We will go deeper into the details soon, but for now, all you need to know is, without creating a Git repository, you really can't do much with Git.

No matter how big your project is (in other words, no matter how many files or sub-directories your project has), the **top** (or root) folder of that project needs to have **`git init`** run to get things started with Git.



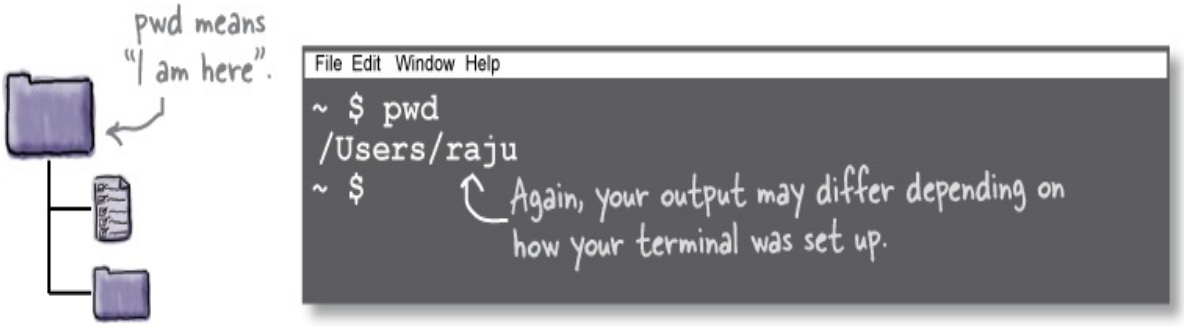
1. Create a project folder
2. Initialize Git

3. Initializing a Git repository inside a folder gives it superpowers. You will often here folks referring to this as the “working directory”.

A quick tour of the command line

One thing you are going to be using a lot while working the exercises in this book is the command line, so let’s spend a little time getting comfortable with it. Start by opening a terminal window like we did in the introduction, and navigate to a location on your hard-drive. As a reminder, on the Mac you’ll find the **`Terminal.app`** under **`Applications > Utilities`** folder. On Windows navigate using the Start button, and you should see Git BASH under the Git menu option. You will be greeted with a prompt, and that is your cue that the terminal is ready to accept commands.

Let’s start with something easy. Type `pwd` and hit return; `pwd` stands for “print working directory” and it displays the path of the directory the terminal is currently running in. In other words, if you were to create a new file or a new directory then they would show up in this directory.



SHARPEN YOUR PENCIL

Time to get busy! Fire up the terminal, and use the `pwd` command. Jot down the output you see here:

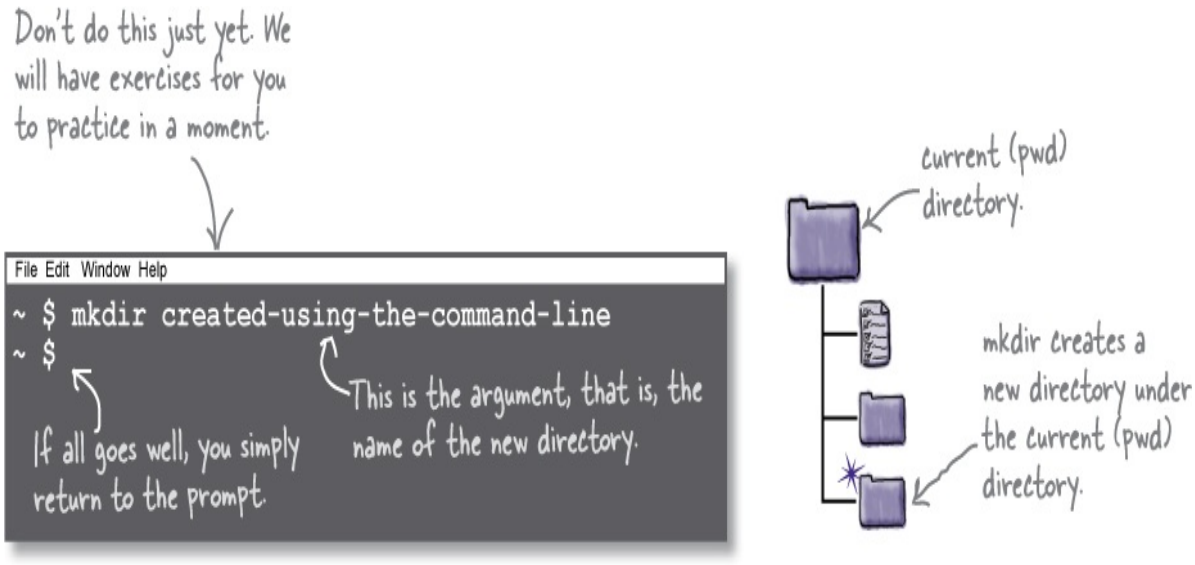
> **NOTE**
>
> You'll find the answers to Sharpen Your Pencil exercises at the end of the chapter

Great! If this is your first time using the terminal, or you are not very familiar with it, then it can be a little daunting. But know this—we will guide you every step of the way, not just for this exercise but all exercises in this book.

More on the command line (mkdir)

Knowing the location of the current directory in the terminal, using `pwd` is super useful because almost everything you do is relative to the current directory, which includes creating new folders. Speaking of new folders, the command for creating new folders is `mkdir` which stands for "make directory".

Unlike `pwd` which simply tells you the path of the current directory, `mkdir` takes an *argument*, which is the name of the directory you wish to create:



WATCH IT!

`mkdir` will error if you attempt to create a dirctory with a name that already exists.

If you attempt to create a new directory with the same name as one that already exists in the current directory, `mkdir` *will simply report* `File exists` *and not do anything. Also, don't let the "file" in "File exists" confuse you—in this case it simply means folder.*

SHARPEN YOUR PENCIL

Your turn. In the terminal window you have open go ahead and use `mkdir` to create a new directory called `my-first-commandline-directory`.

> **NOTE**
>
> Make sure you check your answer at the end of the chapter

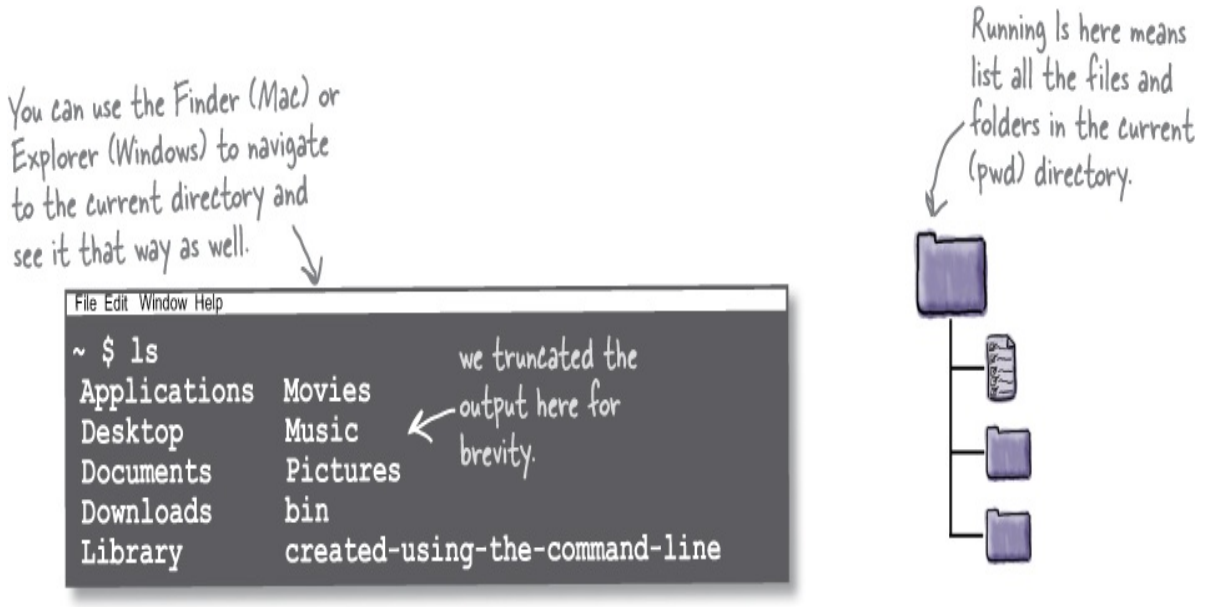
> **NOTE**
>
> Write the command and argument you used here.

Next, run the same command again, in the same directory. Write down the error you see here:

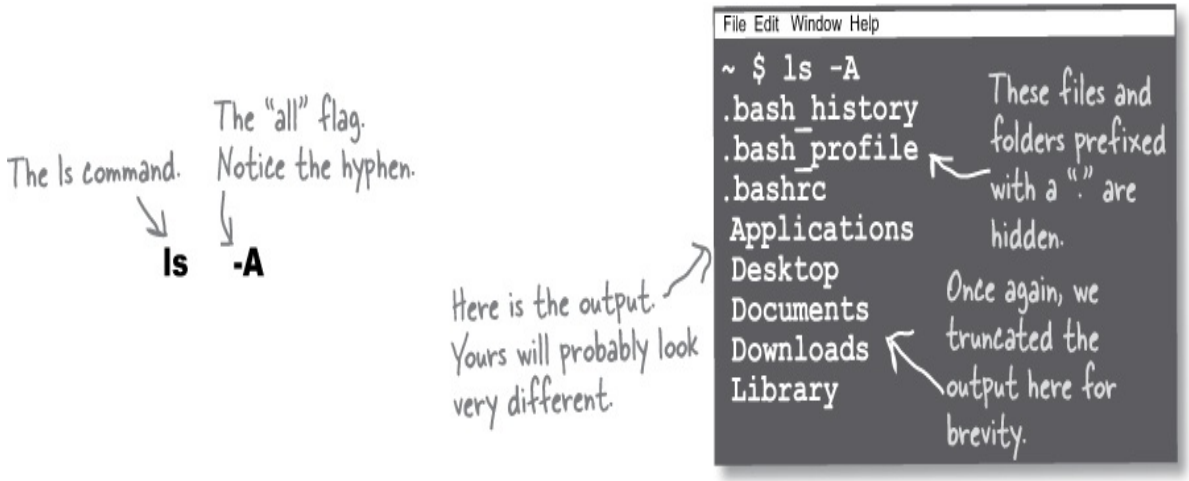
> **NOTE**
>
> Error goes here.

More on the command line (ls)

The output of `mkdir` isn't very encouraging to say the least. But as long as you did not get any errors, it did its job. To confirm if something did happen, you can list all the files in the current directory. The listing command is named `ls` (short of list).



The thing is, `ls` by default only lists regular files and folders. Every so often (and we are going to need this soon enough) you want to see hidden files and folders as well. To do that, you can supply ls with a *flag*. Flags, unlike arguments are prefixed with a hyphen (to differentiate them from arguments). To see "all" files and folders (including hidden ones) we can use the "A" flag, like so:



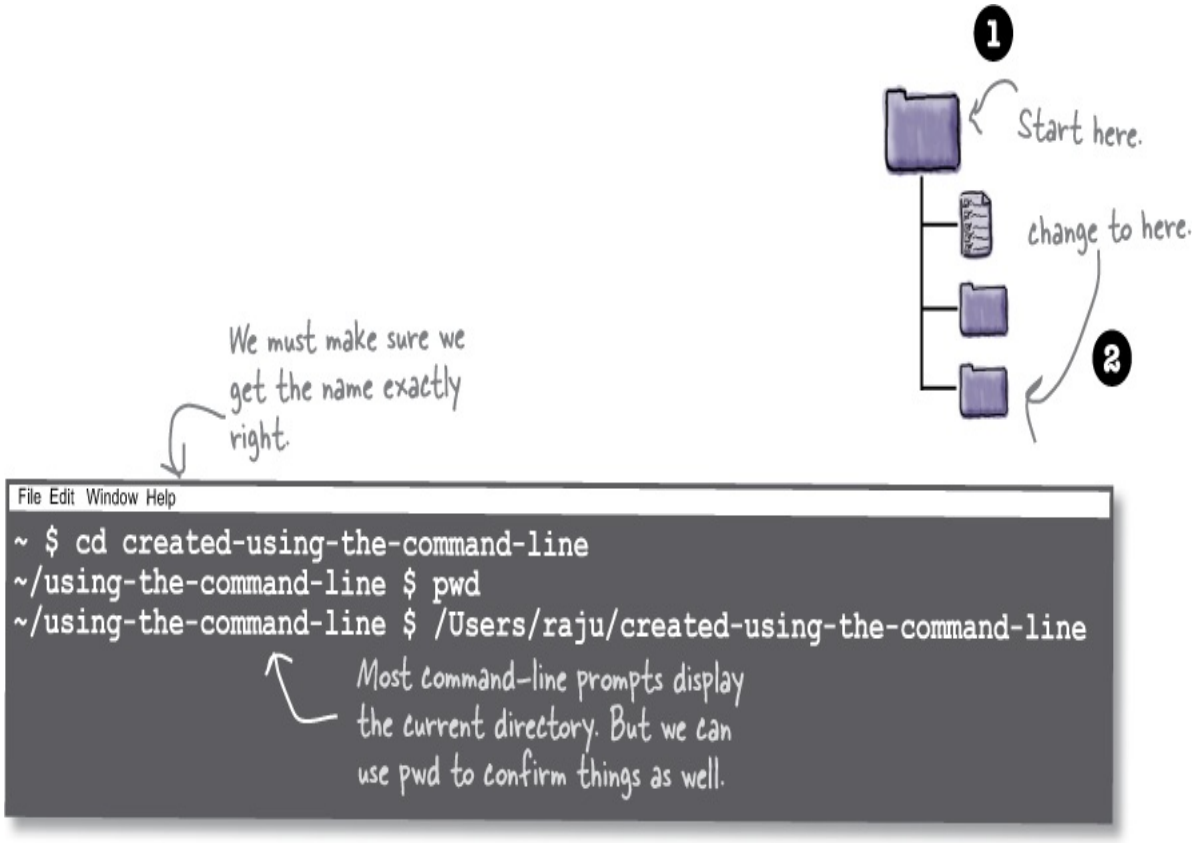
SHARPEN YOUR PENCIL

Use the terminal to list all the files in the current directory. See if you can find your recently created `my-first-commandline-directory`.

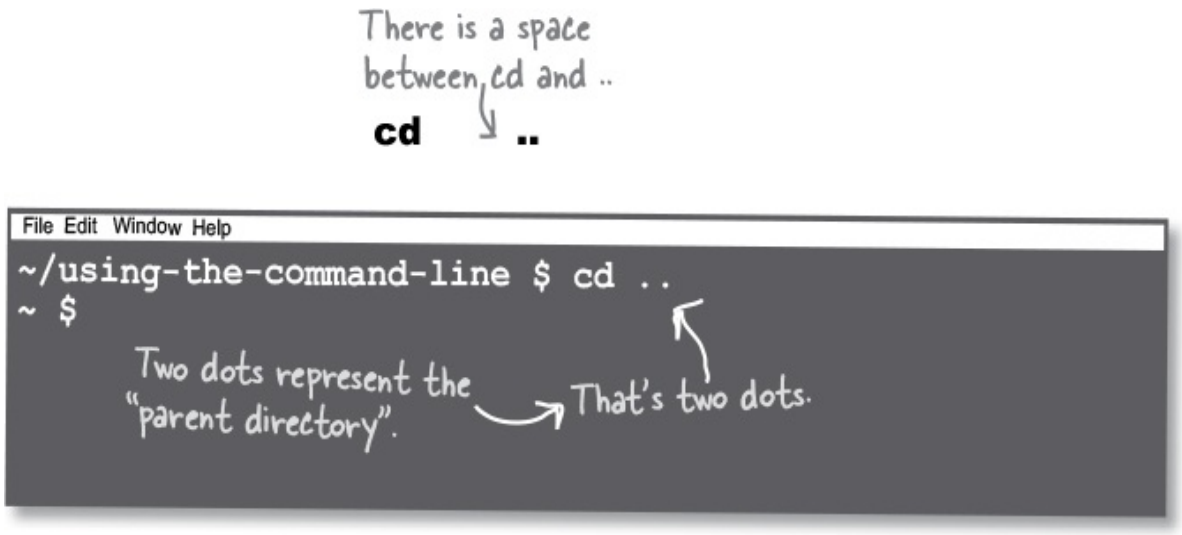
Then use the `-A` flag and see if there are any hidden folders in the current directory.

More on the command line (cd)

Next, navigation! We created a new directory, but how do we navigate to it? For that, we have the `cd` command, which stands for “change directory”. And of course, once we change directories, we can use `pwd` to make sure that we indeed did move locations.



`cd` navigates to a subdirectory under the current directory. To hop back up to the parent directory, we can also use `cd`, like so:



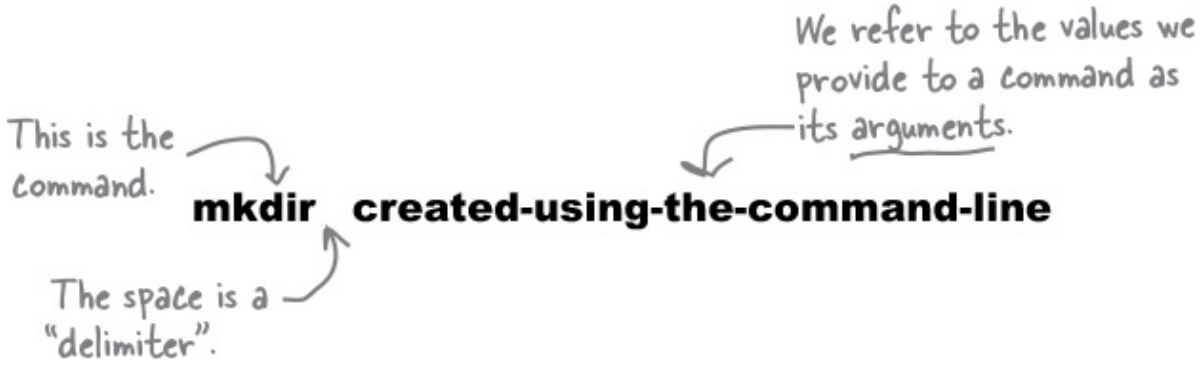
Always keep track of your working directory (using pwd)—most operations on the command line are relative to this directory.

EXERCISE

Go ahead, give it a spin. Play around with `cd` to hop into your newly created `my-first-commandline-directory` folder, then use `pwd` to make sure you did change directories, and then use `cd ..` to go back to the parent folder. Use this space as a scratchpad to practice out the commands as you use them.

No argument there

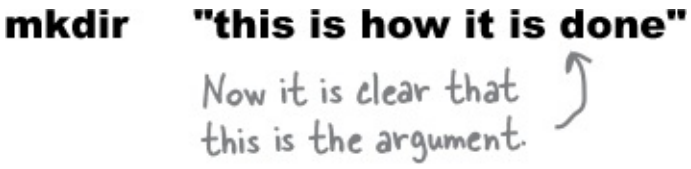
Command-line functions like `pwd` and `mkdir` are the "commands" we are invoking. Some commands like `mkdir` and `cd` expect you to tell them what you want to create or where to go. The way we supply those is by using arguments.



You might be wondering why we chose to use hyphens instead of spaces.Turns out, using spaces in arguments can get rather tricky. You see, the command line uses this to seperate the command from it's arguments. So, it can be super confusing to the command line if your arguments have spaces in them.

The command-line can be rather finicky, particularly when it comes to white-space. It turns out that for the command-line, white-space acts as a separator. In other words, this is how the command line separates the command from everything else. But if we put spaces in the arguments, it can't figure out where the argument starts and end.

So, anytime you have white-space in an argument, and you wish to treat it as one argument, you need to use quotes.



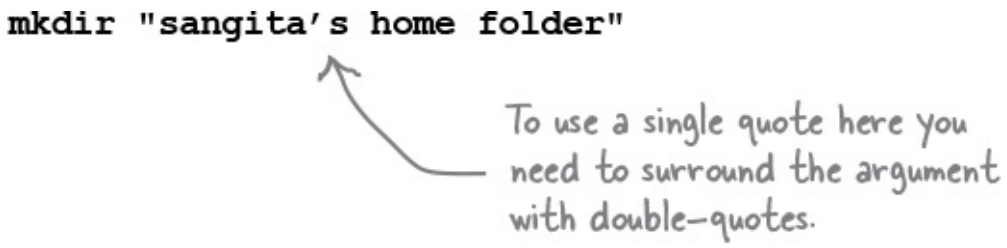
As a habit, try to avoid white-space in file names and paths.

> **NOTE**
>
> *For example, its better to have C:\my-projects\ than C:\my projects*

Great question. The command line does not really care if you use double quotes or single quotes. The thing to remember is that you need to be consistent. If you start the argument name with single quotes, then end it with a single quote. Likewise, for double quotes.

Typically, most folks using the command line tend to prefer double quotes and so do we; however, there is one situation where you will be forced to use double-quotes and that is if your argument has a single quote in it.

Notice that in this case we are using a single quote in the word `sangita's`:



The opposite is true if you need to use a double quote in your argument, in which case you'll need to surround your argument with single quotes.

However, we alluded to this, it's best if we avoid whitespace in our arguments, particularly in the names of directories and files. **Anytime you need a space, simply use a hyphen or an underscore**. This helps you avoid using quotes (of any kind) when supplying arguments.

WHO DOES WHAT?

With the command line there's a lot of commands and options flying around. In this game of who does what, match each command to its description.

cd	Displays the path of the current directory.
pwd	Creates a new directory.
ls	Navigates to the parent directory.
mkdir	Changes directories.
ls -A	Lists regular files in the current directory.
cd ..	Lists **all** files in the current directory.

Cleaning up

Now that you are done with this section, we suggest you clean up the folders you created like `my-first-commandline-directory` and any others. For this, just use the Explorer or the Finder window and delete them. While the command line offers you ways to do this, deleting files using the command line usually bypasses the trashcan. In other words, it's hard to recover if you

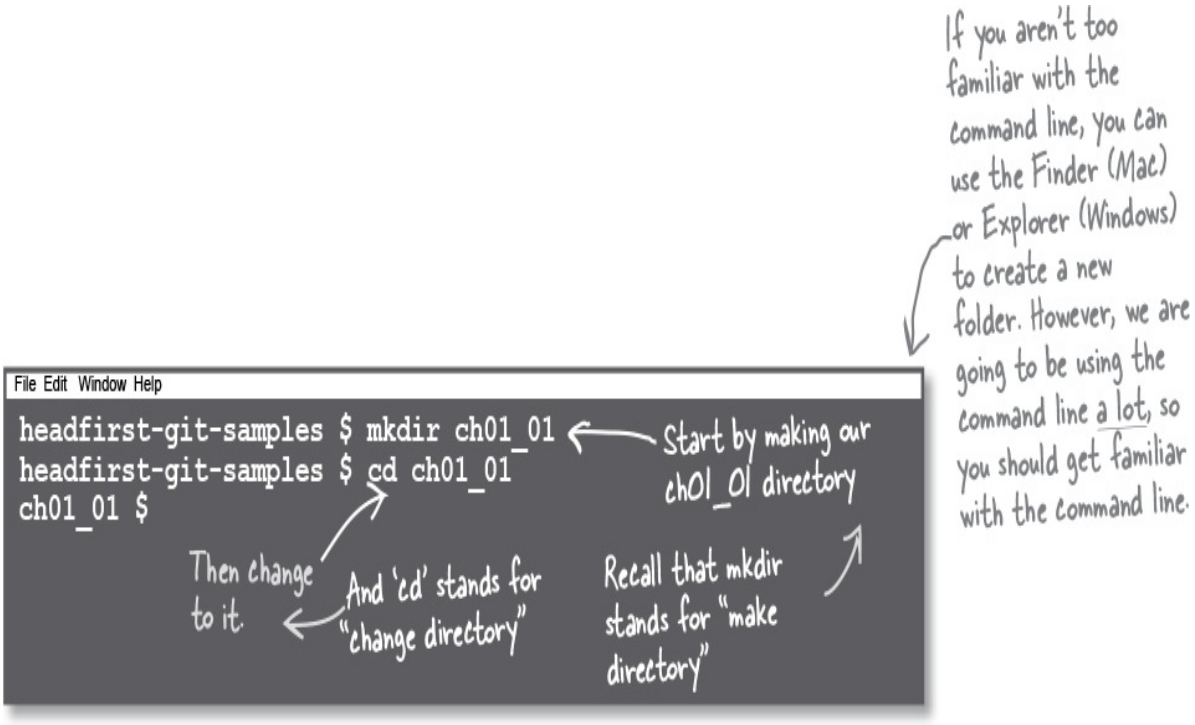
accidentally delete the wrong folder.

In the future, when you get more familiar with the command line, perhaps you might use the appropriate command to delete files, but for now, let's play it safe.

Creating your first repository

Let's get a little acquainted with Git before we dive into the deep end of the pool. You already have Git installed, so this will give us a chance to make sure everything is set up and get a sense of what it takes to create a Git repository. To do that, you will need a terminal window. That's it!

Start by opening a terminal window like we did in the previous exercise. Just to keep things easier to manage, we suggest you create a **`headfirst-git-samples`** folder to house **all** the examples in this book. Within that, go ahead and create a new folder for our first exercise for Chapter 1, called **`ch01_01`**.



Now that we are in a brand new directory, let's create our first Git repository. To do this, we simply run `git init` inside our newly created folder.

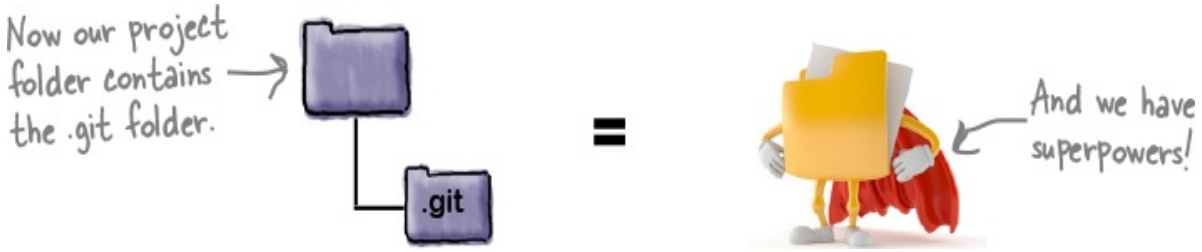
That was pretty painless, wasn't it? And there you have it—your first Git repository.

Inside the init command

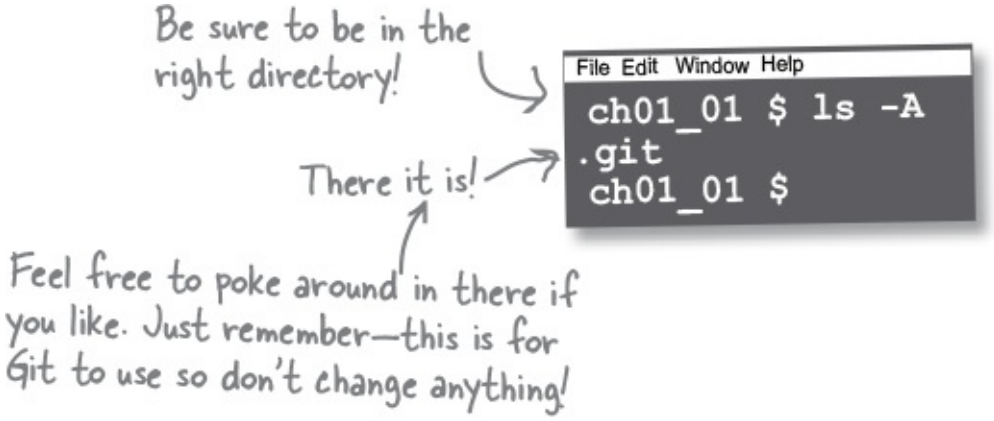
So what exactly did we just accomplish? The `git init` command might not look like much, but it sure packs a punch. Let's peel back the covers to see what it really did.

To begin with, we started with a new, and empty directory.

Using the terminal we navigated to the folder location and invoked the magic words, `git init`, where `init` is short for **initialize**. Git realizes we are asking it to create a repository at this location, and it responds by creating a hidden folder called `.git`, and stuffs it with some configuration files, and a subfolder where it will store our snapshots when we ask it to.



One way to confirm this happened is by listing all the files using our terminal, like so.



This hidden folder represents the Git repository. It's job is to store everything related to your project, including all commits, the project history, configuration files, what-have-you. It also stores any specific Git configuration and settings that might have enabled for this particular project.

THERE ARE NO DUMB QUESTIONS

Q: I prefer to use my file-system explorer when navigating my computer? Can I use that to see the `.git` folder?

A: Of course! By default most operating systems do not reveal hidden files and folders in the explorer. Be sure to look at your preferences and ensure that you can see hidden files and folders.

Q: What happens if someone were to accidentally delete this directory?

A: First of all, let's not do that. Second, this directory is the "vault" in which Git stores all its information—including your entire project history and a bunch of other files that Git needs for house-keeping, and some configuration files that we can use to customize our experience with Git.

This means that if you were to delete this folder you will lose all project history. However, all the other files in your project folder will remain unaffected.

Q: What happens if I accidentally run `git init` more than once in the same folder?

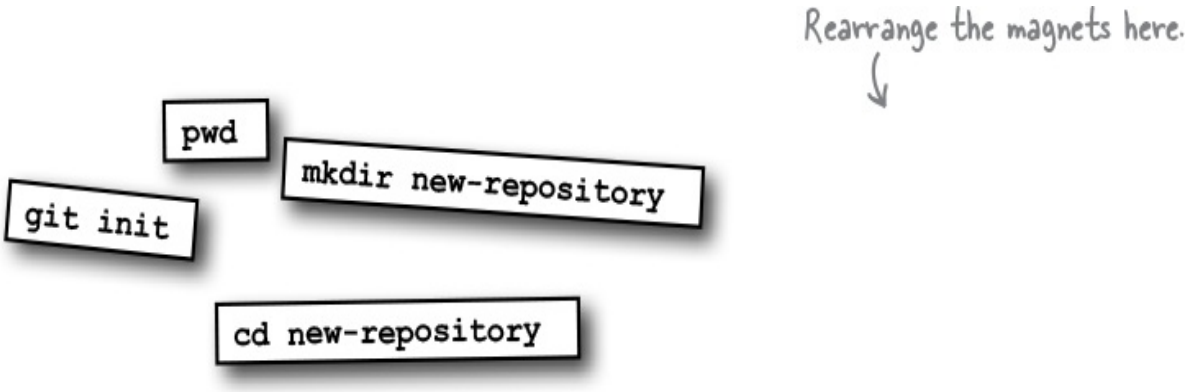
A: Good question. This is completely safe. Git will simply tell you that it is reinitializing the Git repository, but you will not lose any data nor will you hurt anything. In fact, you should try it in `ch01_01`. We are early in our journey, and the best way to learn is to experiment. Whatcha got to lose?

Q: Other version control systems that I have used have a server component. Don't we need that here?

A: Getting started with Git is *really* easy. `git init` creates a Git repository and you can get to work. Eventually you will need a mechanism to share your work with your teammates, and we promise we will get to that soon enough. But for now, you are all set.

Magnetic thoughts

We have all the steps listed to create a new folder, change to it, and initialize to create a new Git repository. Being diligent developers, we often check to make sure we are in the correct directory. To help our colleagues we had the code nicely laid out on our fridge using fridge magnets, but they fell on the floor. Your job is to put them back together. Note that some magnets may get used more than once.

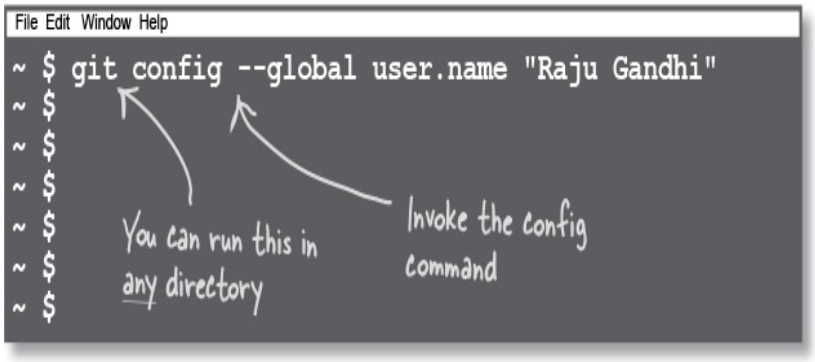


Introduce yourself to Git

There is one more step before we get to work with Git and Git repositories. Git expects you to tell it a few things about yourself, so let’s knock that out. You only have to do this once, and this will apply to any and all projects that you work with on your machine.

We will start with our trusty old friend, the terminal and follow along. **Be sure to use your name and email instead of ours!** (We know you love us, but we wouldn’t want to take credit for your work!). Start by opening a new terminal window. Don’t worry about changing directories—for this part of our setup it does not matter where you run this.

1 We will start with telling Git our full name



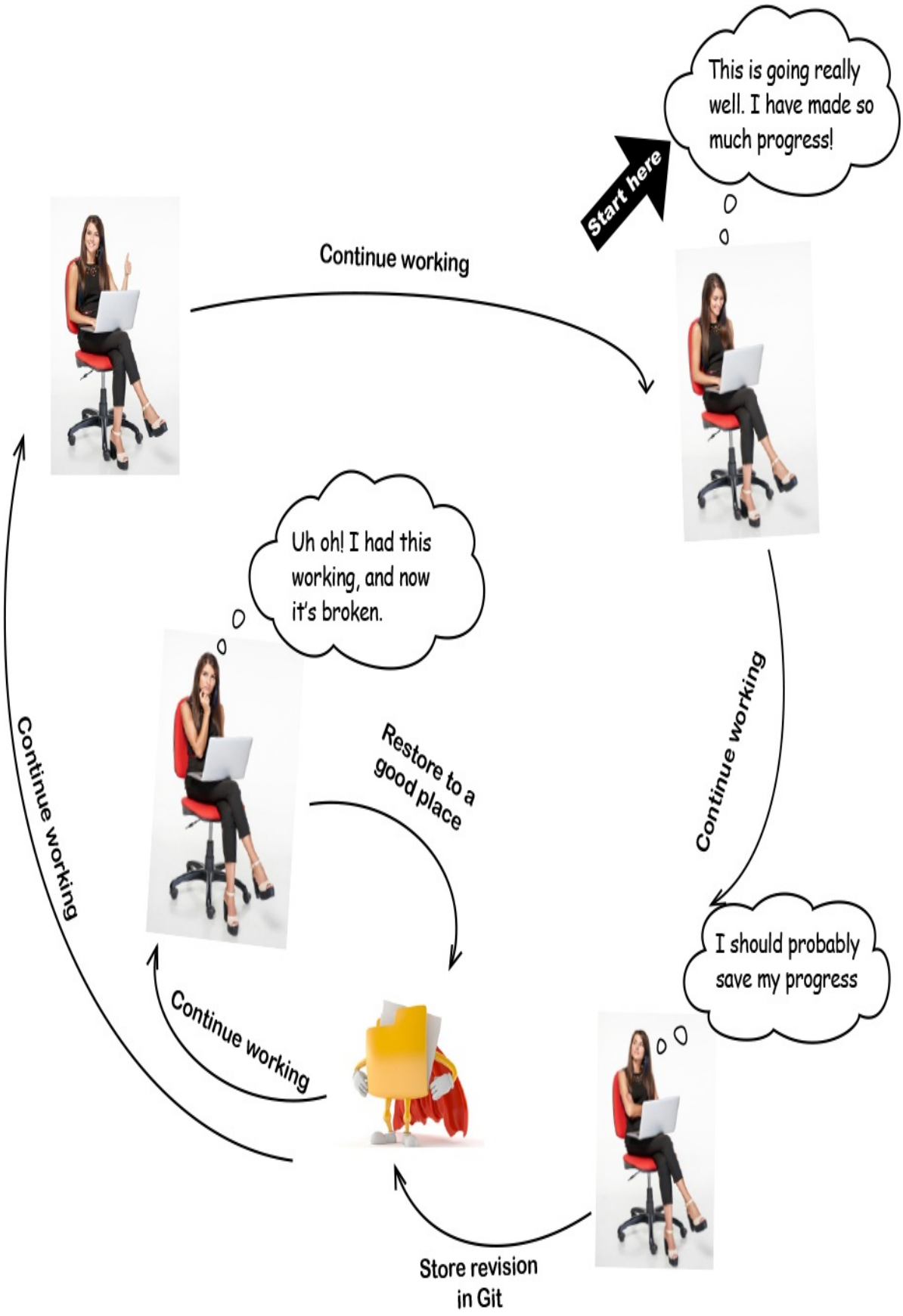
2 Next, we tell Git our email address. You can use your personal here for now, and always change it later.

Note: You can always change these later by running the same command again with diffferent values. So if you choose to use your work email address once you are done with this book, feel free to do just that. You might wanna bookmark this page just in case.

How you will use Git

Let's get a sense of how a typical interaction with Git looks like. Remember how we spoke about video games allowing you to save your progress? Well, asking Git to "save your progress" involves "committing" your work to Git. Essentially this means that Git stores a revision of your work. Once you do that, you can

continue working away merrily till you feel it’s time to store another revision, and the cycle continues. Let’s see how this works.


This is going really well. I have made so much progress!
Start here
Continue working
Uh oh! I had this working, and now it's broken.
Restore to a good place
Continue working
I should probably save my progress
Continue working
Continue working
Store revision in Git

Putting Git to Work

We are sure you raring to get started (we know we are!). So far, we have initialized a Git repository, told Git our name and email, and kinda-sorta have a sense of how we usually work with Git. So how about we actually put Git to work. We will start small and just put Git through its paces—we will see what it takes to “take a snapshot” in Git by creating a “commit”.

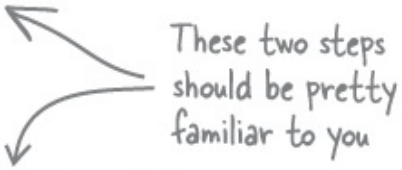
For the sake of this exercise let’s pretend to start working on a new project. If you are anything like us, we usually start with a checklist so we can keep track of everything we have to do. As we progress with the project, we keep checking things off (gotta keep that dopamine flowing!), and as we learn more about the project, we keep adding to it. Naturally, this file is version controlled with the rest of the files in the project, for which we will use Git.

Let’s break down what we are going to do, step by step.

Step One:

Create a new project folder.

Step Two:



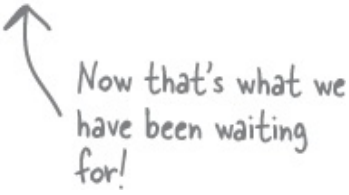
Initialize a Git repository within that folder.

Step Three:

Create our checklist with a few items to get us started.

Step Four:

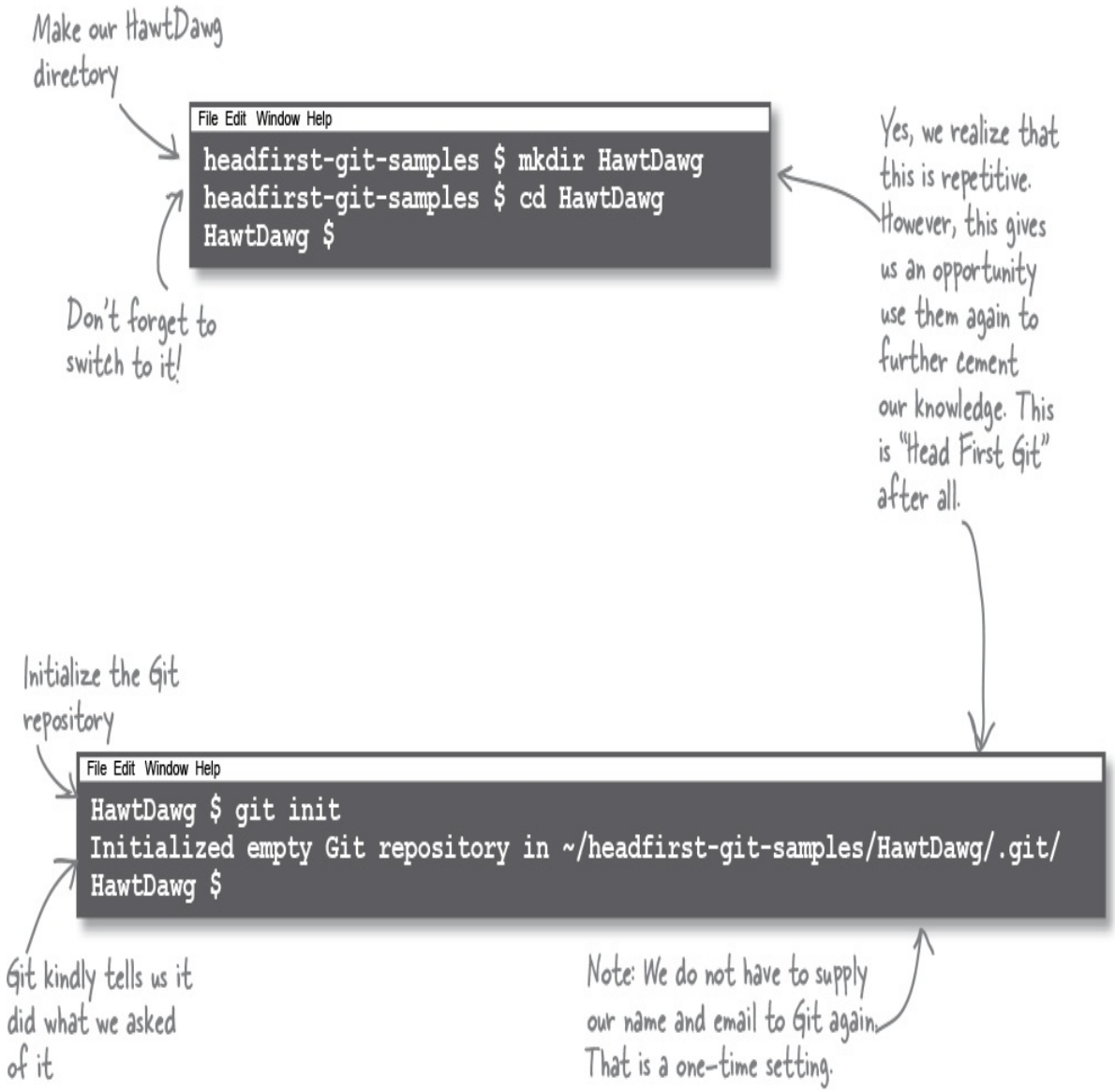
Store a snapshot of our checklist in Git by committing the file.



Meanwhile, back at the HawtDog Dating Service ...

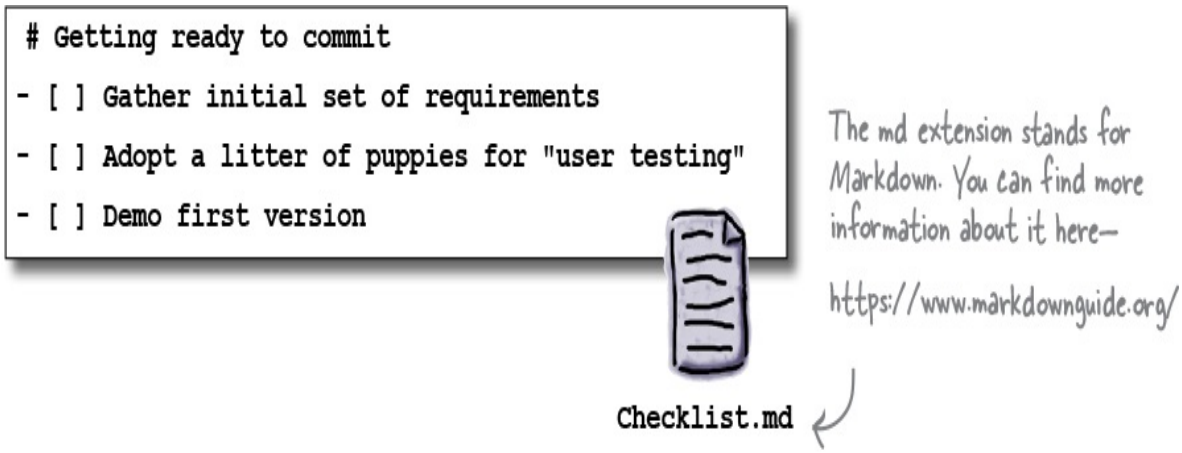
Your first step involves creating a new folder under the umbrella **`headfirst-git-samples`** folder. Be sure you are in the right directory using **`pwd`**. You may have to use **`cd ..`** (remember the two dots there) to go up one level if your terminal is still in the `ch01_01` directory.

Next, we simply initialize a new repository inside **`HawtDawg`** using the altogether familiar **`git init`**.



Next, create a new document in your favorite text editor, and type in the following lines of text. If you followed the instructions in the introduction to install Visual Studio Code, then just like the terminal, you will find **`Visual Studio Code.app`** under the **`Applications`** folder. On Windows, just click on the Start menu and you should see Visual Studio Code listed under all the applications installed on your machine.

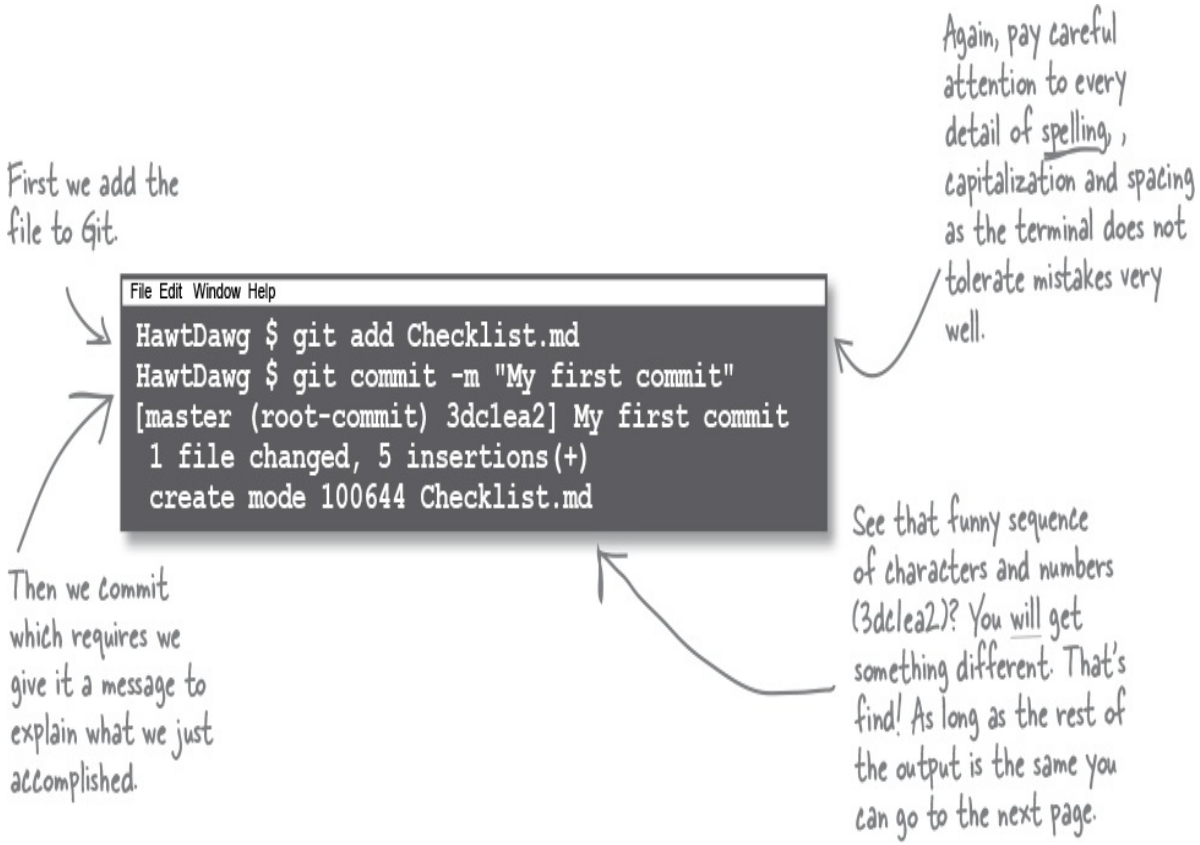
> **NOTE**
>
> To create a new file, simply click on File menu item at the top and pick "New File".



Save the file as **`Checklist.md`** in the **`HawtDawg`** directory

> **NOTE**
>
> To save the file, select File from the top menu, select Save, and then navigate to where you created the HauwDawg directory.

Now we are ready to commit our work. This involves two Git commands, namely **`git add`** and **`git commit`**.



Speaking of ...

Congratulations on your first commit!

You have completed a whirlwind tour of Git. You installed Git, initialized a Git

repository, and committed a file to Git's memory. This gives us a great starting point and we should be ready to dive deeper into Git.

WATCH IT!

Did you get some other output than the one we showed you in the previous exercise?

The comand line can be rather unforgiving when it comes to typos, white-space and casing. If you did not get the same output as ours, then here are few things to try:

- *If you see an error like* `fatal: not a git repository` *be sure that you are in the* `ch01_02` *directory*
- *If you got an error like* `command not found` *then be sure to check to make sure you got the case and the spelling right. Usually the command-line tells you which command it did not recognize.*
- *If you see an error along the lines of* `fatal: pathspec checklist.md did not match any files` *when you tried a* `git add`*, know that the filename you supply needs to match the filename exactly, which in our case would be Checklist.md.*
- *If you get* `error: pathspec '-' did not match any file(s) known to git` *when trying to* `git commit`*, make sure that there is no space between the* `-` *and* `m`*.*
- *If the command-line reports an error like* `error: pathspec 'first' did not match any file(s) known to git` *make sure to wrap the commit message "My first commit" in double-quotes.*
- *If you get an error like* `nothing added to commit but untracked files present`*, then try running* `git add Checklist.md` *again, this time making sure you get the filename correct, including the casing.*

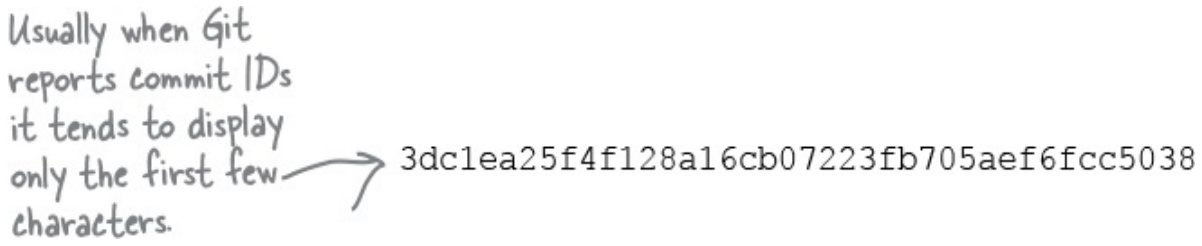
What exactly does it mean to commit?

We saw that committing to Git is a two-step process. You first `add` the files and then `commit`.

The first thing to know is that only the files that you add are committed. Let's say you had two files—`Checklist.md` and `README.md`, but you only added `Checklist.md`. When you create a commit, Git will **only** store the changes made to `Checklist.md`.

Now, when we commit, Git uses a specialized algorithm to safely tuck away

everything that we added in it's memory. When we say we "committed" our changes to Git, what that translates into is that Git creates a **commit object** that it stores inside the `.git` folder. This commit object is 'stamped' by a unique identifier. You might recall that we got `3dc1ea2` when we made our commit in our last exercise (you certainly saw something different)—this is actually a much longer string containing numbers and letters that looks something like this:

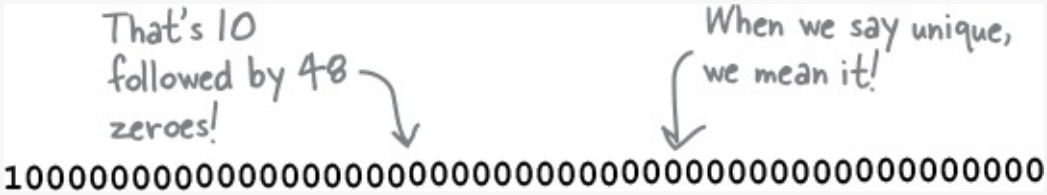


This identifier is computed on a bunch of metadata, including your fullname, the time as it were when you made the commit, the commit message you provided, and information derived from the changes you committed.

The last thing you need to know is ...

SERIOUS CODING

Amazingly enough, the chances that two commits will EVER have the same ID (and yes, that is across all the Git repositories in the world, those that exist and even those that haven't even been created yet) is less than 1 in 1048! Yes, that is 10 followed by 48 zeroes!



...The commit object does **not** actually store your changes—well not directly anyway. Instead, Git stores your changes in a different location in the Git repository, and simply records (in the commit) where your changes have been

stored.

A pointer to the location inside the .git folder where Git has stored your changes, called a `tree`.

This is another set of alphanumeric characters, the details of which we will skip for now.

The "author" info—that is, your name and email address.

In an earlier exercise we provided Git with our fullname, and our email. This is also recorded in the Git so that you can claim full credit for the marvelous work you put in.

> **NOTE**
>
> This is why it's important to introduce yourself to Git.

The time the commit was made, represented in seconds elapsed since Jan 01 1970.

Git also records the time when you made the commit, along with the timezone your machine is located in.

> **NOTE**
>
> Commit objects are stored by Git in binary format, making it very hard for humans to read, but super safe and efficient for Git.

The commit message you supplied when you invoked `git commit -m`.

> **NOTE**
>
> There is a little bit more than what we listed here, but we can leave that aside for now.

Look before you leap

Alright, we just made our first commit. Making a commit involves two separate commands—**`git add`** followed by **`git commit`**. You are probably wondering why it takes two commands to make a commit in Git—why does Git make us jump through all these hoops so we can store a revision of our work in Git?

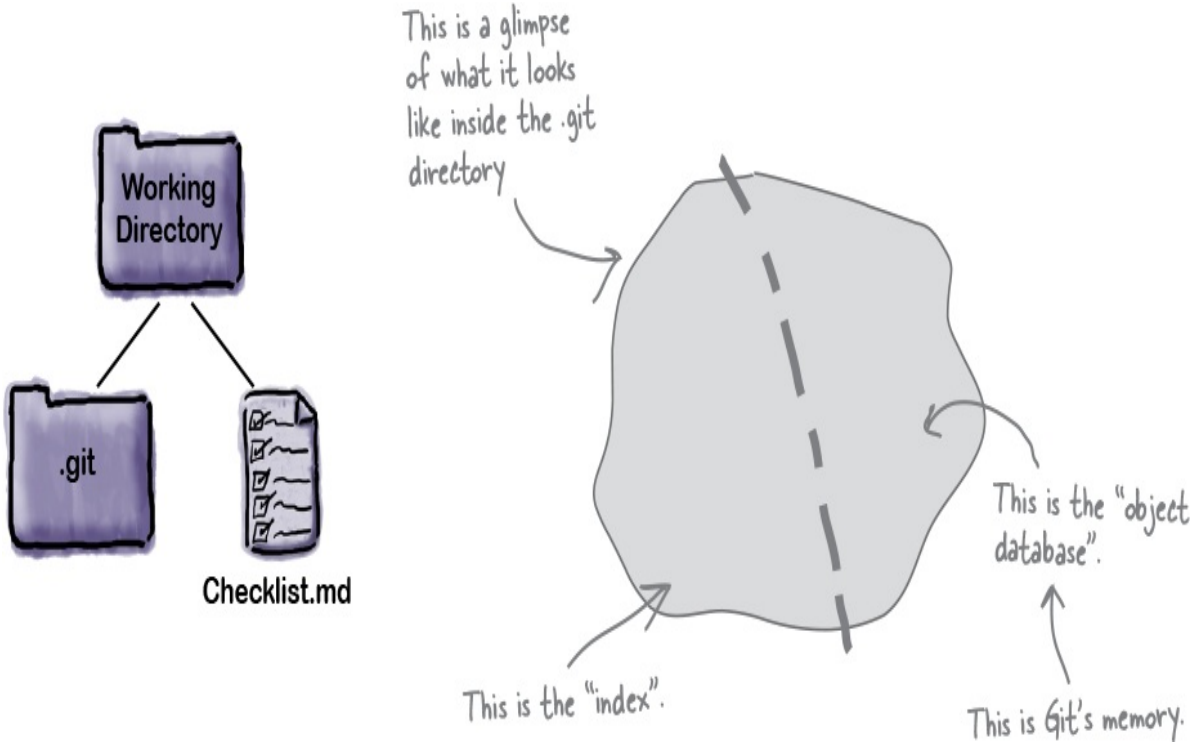
The answer lies in the design of the Git repository. Remember that the Git repository is housed in the `.git` folder that gets created when you run `git init`.

The Git repository itself is divided into two parts—the first part is called the "index", and the second part is what we will refer to as the "object database".

When we run `git add <filename>` Git makes a copy of the file, and puts it in the index. We can think of the index as the "staging area", wherein we can put things till we are sure we want to commit to them.

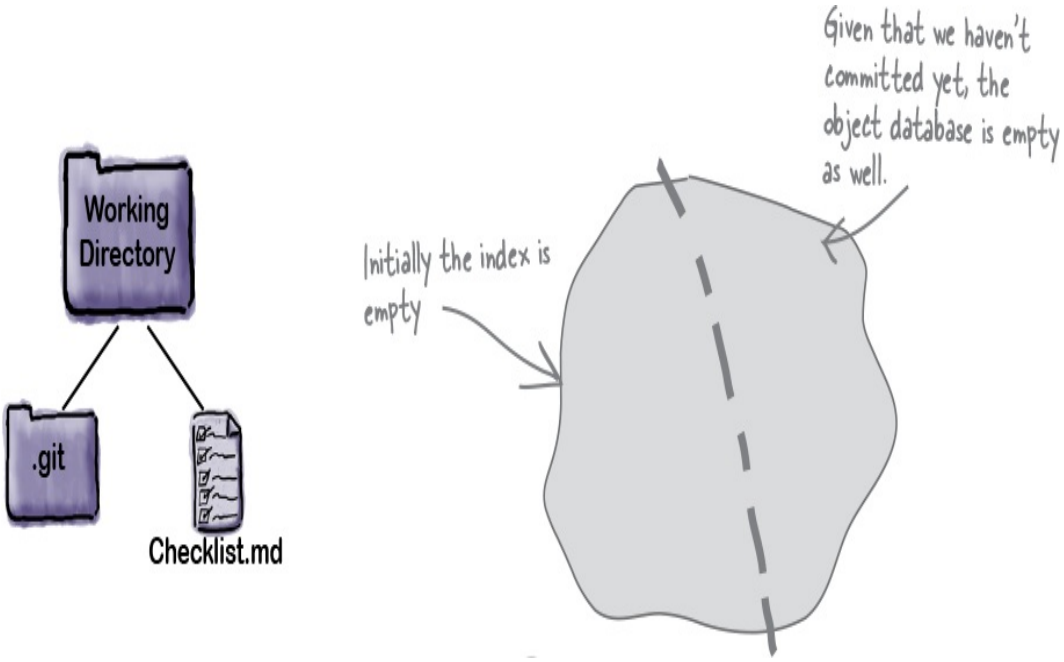
Now when we run `git commit` takes the **contents of the staging area** and stores those in the object database, a.k.a Git's memory bank. To put it another

way, the index is a place to temporarily house changes. Typically, you make some changes, add them to the index, and then decide if you are ready to commit —if yes, then you make a commit. Else, you can continue making changes, add more changes to the staging area, and then when you feel you are in a good place, commit.



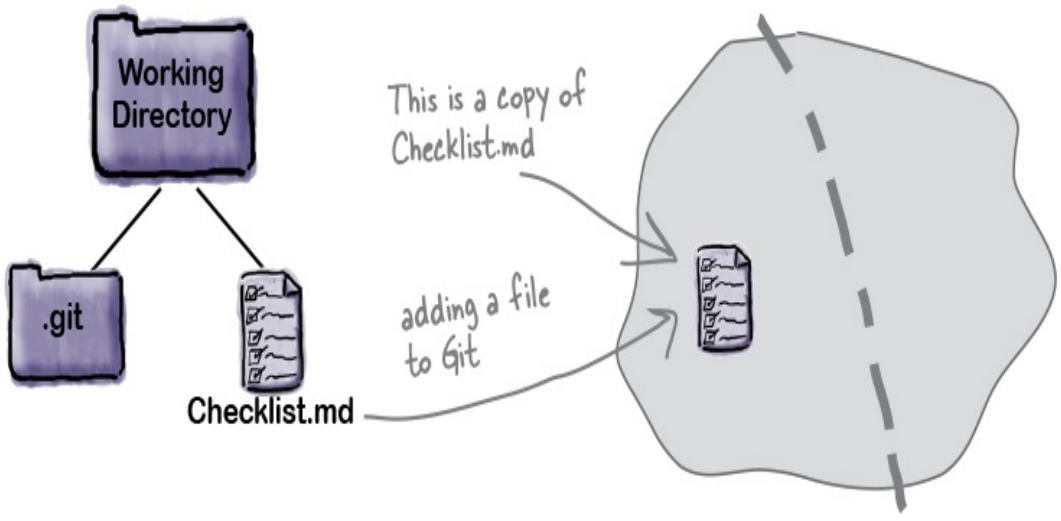
The three stages of Git

1. Let’s start at the top. We have a working directory with just one file.

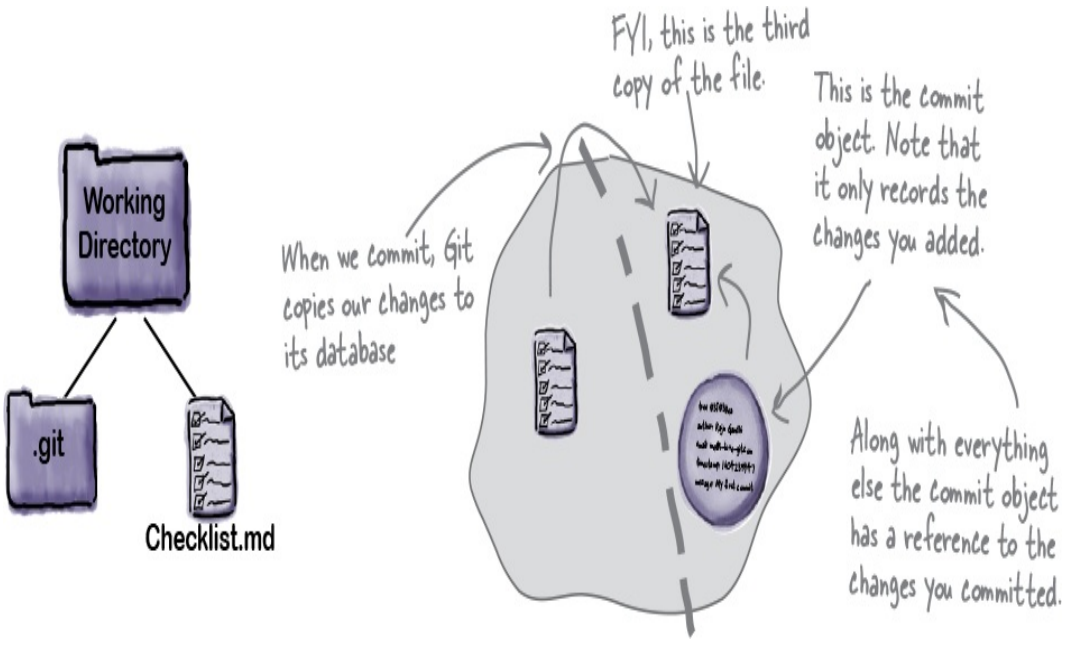


2. When we `git add Checklist.md` Git stores a **copy** of that file in the index.

> **NOTE**
>
> Hold on to this thought—we will come back to it in the following pages.



3. Finally, when we commit, Git creates a commit object that records the state of the index in its memory.



Great question!

We mentioned earlier that the index can be thought of a staging area. It gives you a way to collect everything you need for the next commit.

Consider a scenario where you are working on a new feature or fixing a bug. As you navigate the project you notice a typo in a documentation file and being the

good teammate that you are, you fix it. However, this fix is completely unrelated to your original task. So how do you separate the the documentation fix from your original task?

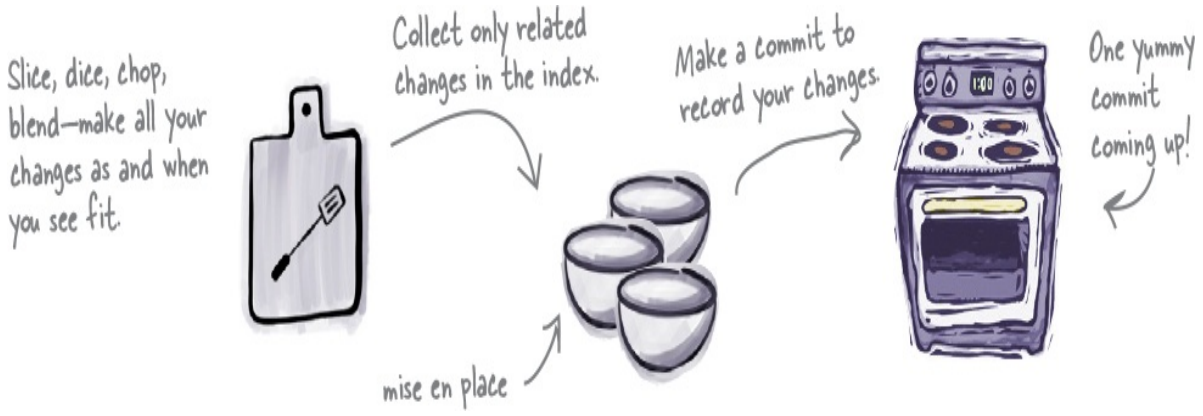
Simple.

You finish the task you were working on, and add all the files that were affected by that change to the index. And then you commit, giving it an appropriate message. Remember, Git only commits the files that were added to the index.

Next you `git add` the file in which you fixed the typo, and make another commit, this time providing a message that describes your fix.

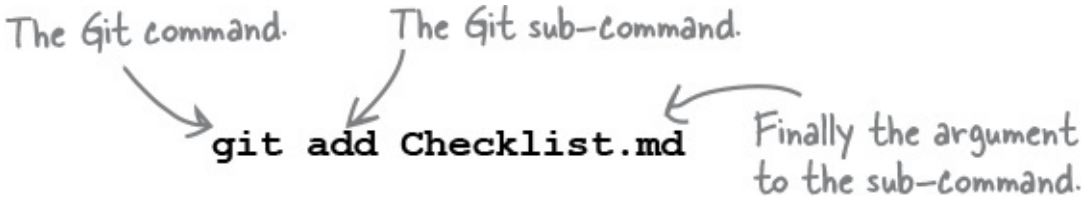
An analogy that might help would be one of cooking. You are having friends over, and are feverishly preparing a bunch of delicious dishes. You may start by chopping up everything you know you will need. However, once you start putting things on the stove, you may choose to collect everything you need for that *particular* dish so they are right there when you need them. You leave everything else by the cutting board. Chefs refer to this as “mise en place”.

The index is your mise en place.



Git in the command-line

We covered some of the idiosyncrasies of the command line previously. This time around let’s make sure we understand how we use Git at the command line. As you have seen, Git uses the `git` command, usually followed by a “sub-command”, like `add` or `commit`, and finally followed by arguments to the *sub-command*.



Command-line cheatsheet

Git commands and sub-commands are always lowercase.

If you want to treat something that has white-space as a "singlular" thing, you need to quote it.

If you need to use quotes, prefer using double-quotes (though single-quotes are allowed).

Since we are using the command line, the same rules that we discussed previously apply. Anytime you have white-space in an argument, and you wish to treat it as *one* argument, you need to use quotes. Consider a very different scenario where we named our file "This is our Checklist.md". In this case, we will have to use quotes when invoking `git add`, like so:

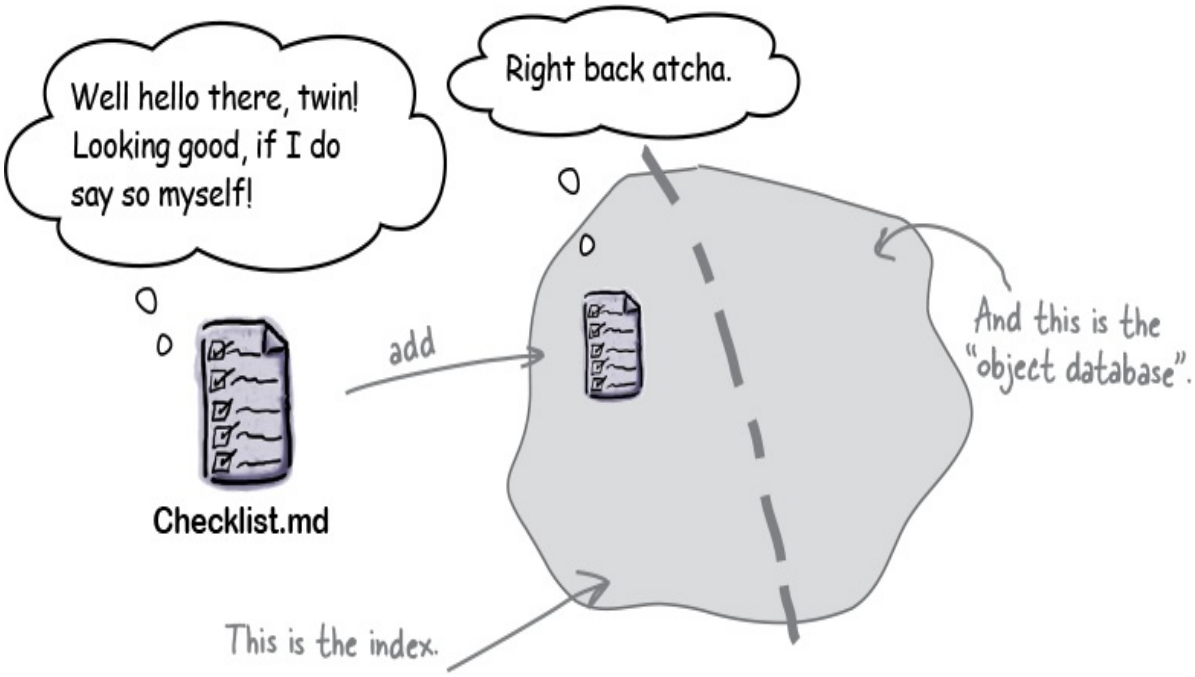
Finally, `git commit` takes both a flag, `-m`, and a message. `-m` is a flag, and here, we should **not** put a space between the hyphen and `m`.

> **NOTE**
>
> Like many flags, -m is short for --message. You can use either, but we are lazy so we prefer the shorter version.

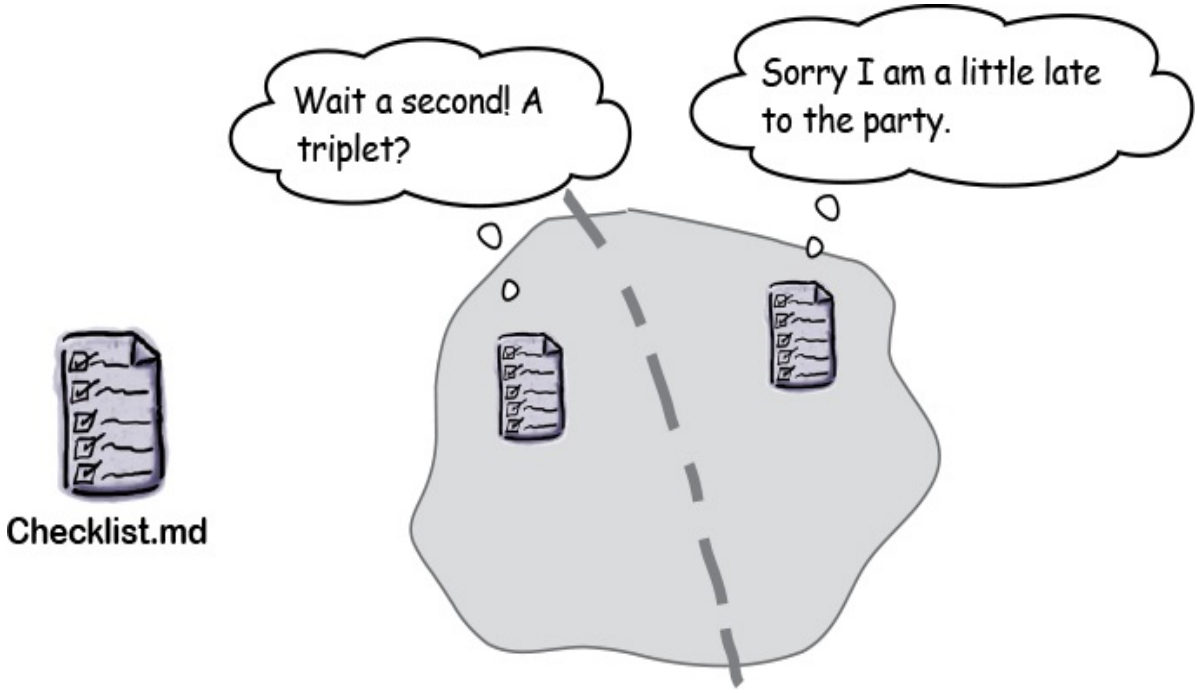
A peek behind the curtain

We are going to let you in on Git's little secret. When you add (one or more files) to Git's index, Git doesn't touch any of the files in your working directory. Instead, it copies the contents of those files to the index. This is an important point because it is crucial to how Git is tracks the content of our files.

> **NOTE**
>
> We alluded to this in the previous pages.



So what happens when we commit? Well, as we know, Git takes the contents of the index, tucks those safely into it's memory bank, and represents that version with a commit object. This means that now, Git has a third copy of your files contents in its object database!

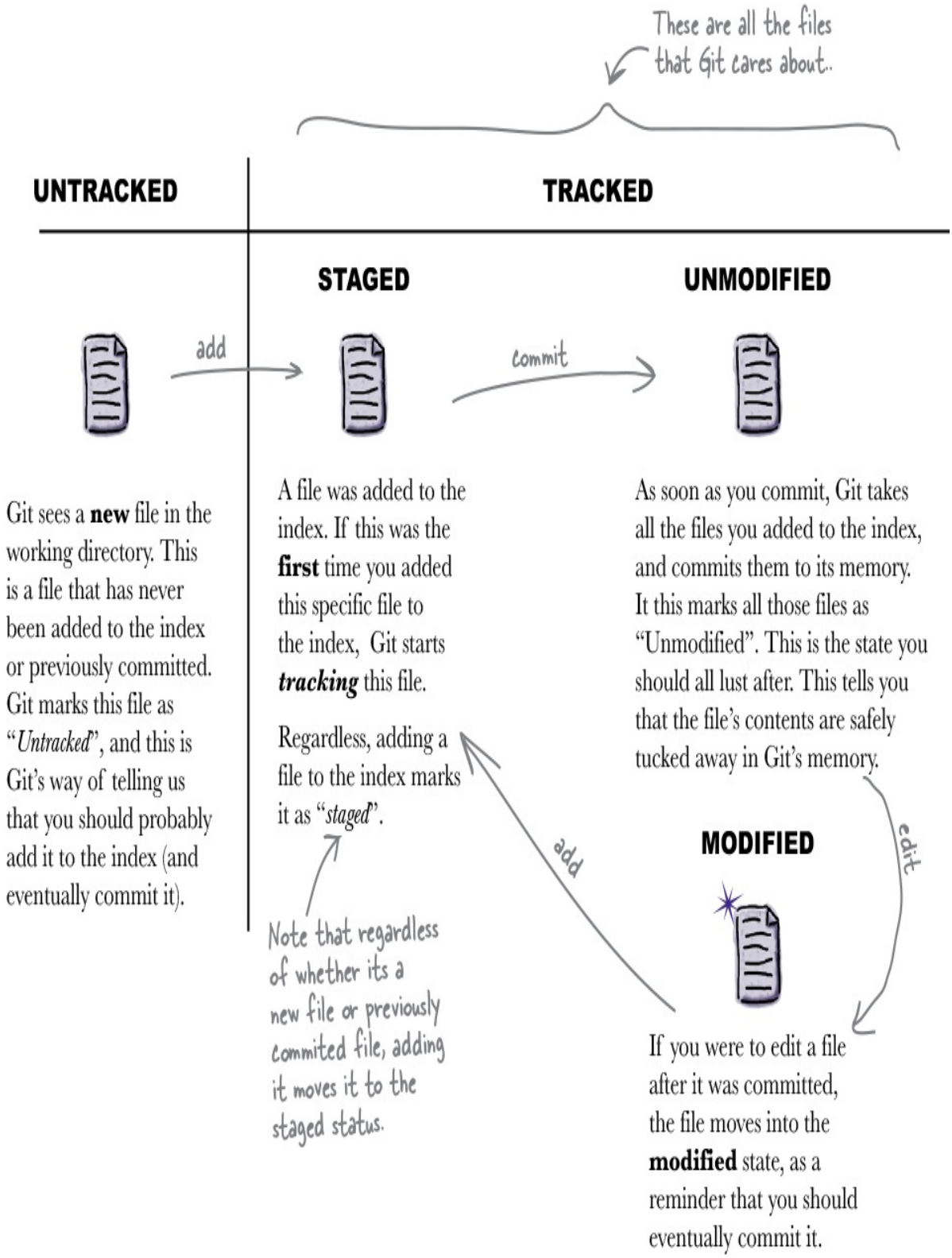


There can be <u>upto</u> three copies of any file in your working directory.

The multiple states of files in a Git repository

Here is what a typical interaction with Git looks like—you make some edits to one or more files, then add them to the index, and when you are ready, you commit them. Now, as you are going through this workflow, Git is attempting to track the state of your files so it know which files are part of your working directory, which files have been added to the index, and which files have already been committed to its object store.

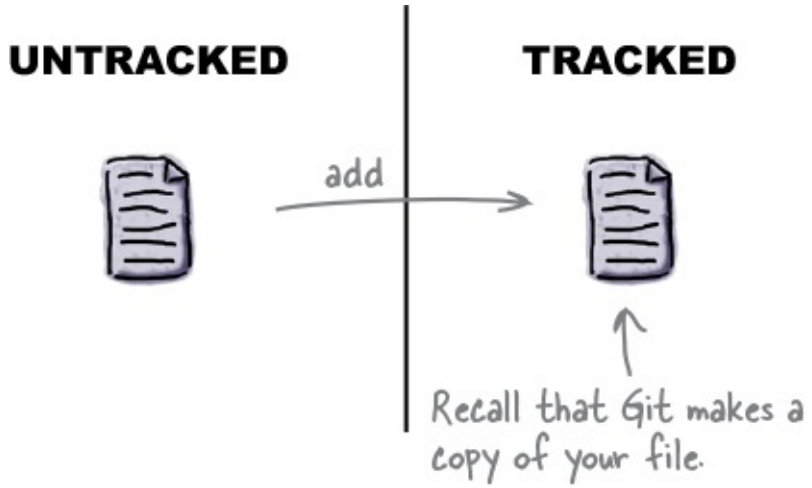
Throughout, keep in mind that Git is moving *copies* of your file from the working directory, to the index, to it’s object datastore.



But there's more. A file may move through all the various stages, but could be in more than one state simultaneously!

A typical day in the life of a <u>new</u> file

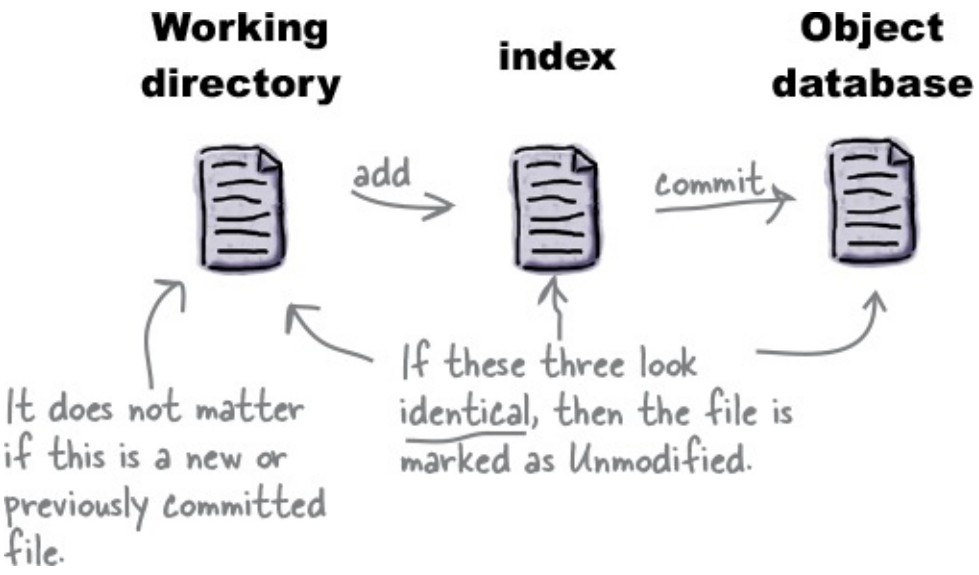
When we add a new file to a Git repository, Git sees the file, but also chooses **not** to do anything till we explicitly tell it to. A file that Git has never seen before (that is, a file that has never been added to the index) is marked as "*Untracked*". Adding the file to the index is our way of telling Git "Hey! We really like you to keep an eye on this file for us". Any file that Git is watching for us is referred to as a "*tracked*" file.



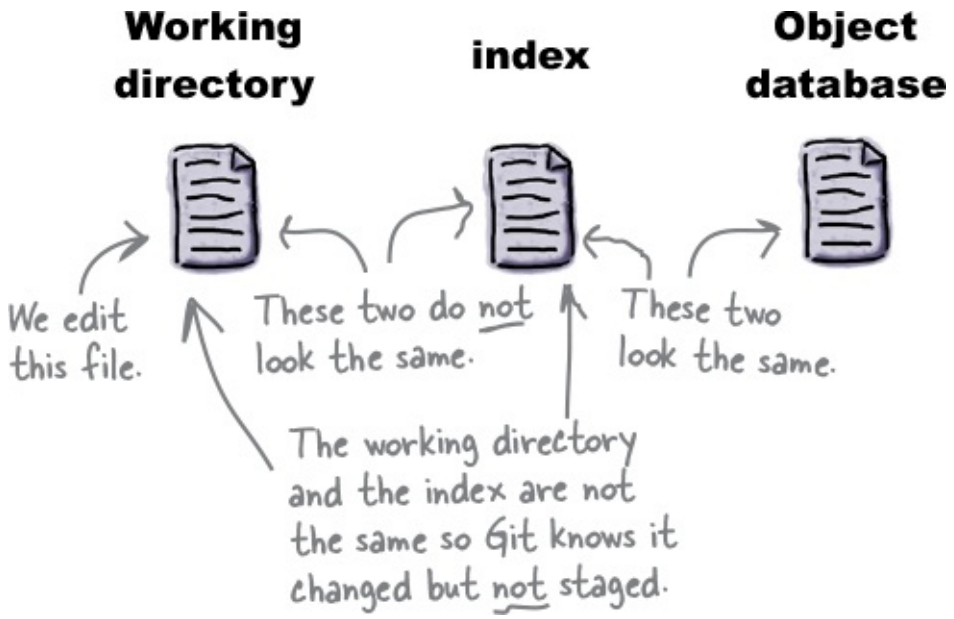
The object database is the "source of truth".

This time, consider adding a file to the index, and then immediately making a commit. Git stores the contents of the index in it's object database, and then marks the file as "*Unmodified*".

Why unmodified, you ask? Well, Git compares the copy it has in its object database with the one in the index, and sees they are the same. It also compares the copy in the index with the one in the working directory, and sees that they are the same. So the file has not been modified (or is Unmodified) since the last commit.



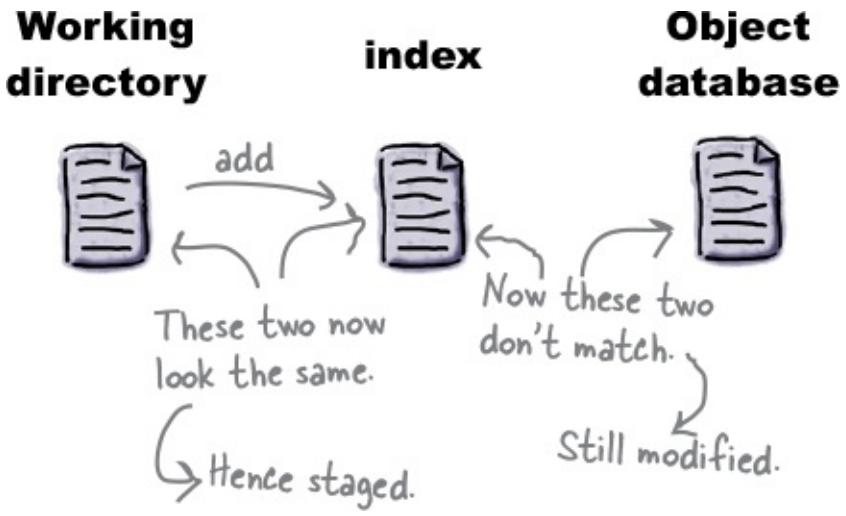
Of course it follows that if we were to make a change to a file that we had previously commited, Git sees a difference between the file in the working directory and the index, but **no difference** between the index and the object database. So Git marks the file as “Modified” but it also marks it as “not staged” because we haven’t added it to the index yet.



Next, if we were to add the *modified* file again to the index. Git sees that the index and the working directory are the same, so the file is marked “staged”, or in other words, it is **both** modified and staged.

And we complete the circle—if we commit, the contents of the index will be

committed, and the file will be marked as “Unmodified”.



BE GIT

Recall that any file in your working directory is either untracked or tracked. Also, a tracked file can be either staged, unmodified, or modified.

In this exercise assume you just created a new repository. Can you identify the state of the files for each of the following steps?

You create a new file in the repository called Hello.txt.

Untracked	Tracked	Staged	Unmodified	Modified

You add Hello.txt to the index.

Untracked	Tracked	Staged	Unmodified	Modified

You commit all the changes that you staged.

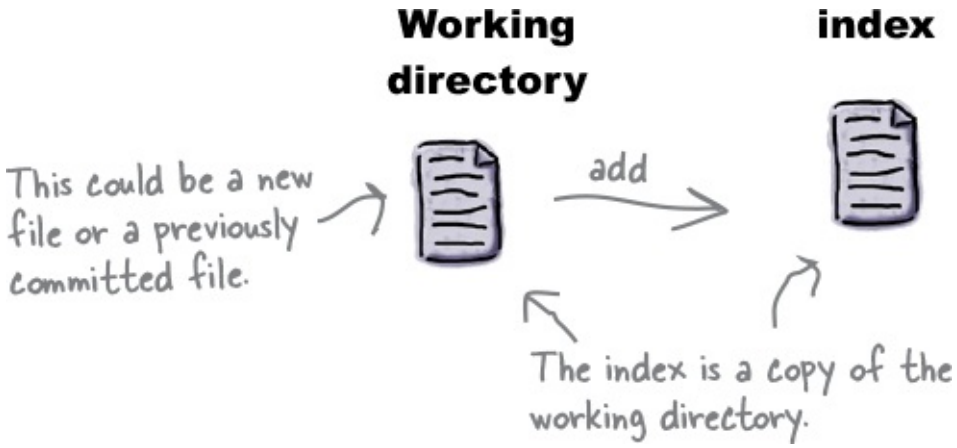
Untracked	Tracked	Staged	Unmodified	Modified

You edit Hello.txt with some new content.

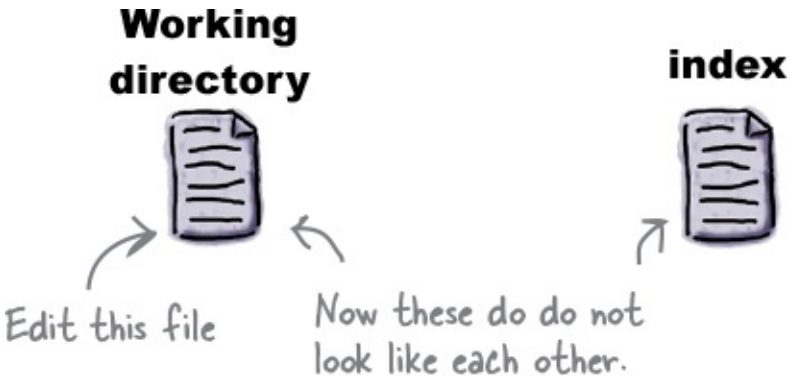
Untracked	Tracked	Staged	Unmodified	Modified

The index is a “scratchpad”

Let’s revisit the role of the index. We know that as we edit files in our working directory, we can `add` them to the index, which marks the file as “staged”.



Of course we can continue editing the file even after adding it to the index. Now, we have two versions of the file—one in the working directory and one in the index.



Now if you add the file again, Git ***overwrites*** the index with the latest changes reflected in that file. In other words, the index is a temporary scratchpad—one you can use to stuff edits into till you are sure you want to commit.

> **NOTE**
>
> This is a super important point. Take a moment for it to sink in before moving on.

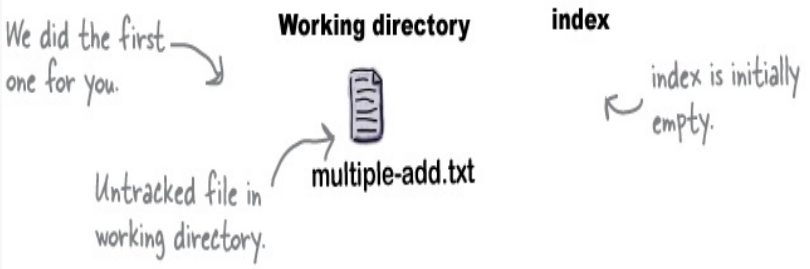
To give you a sense of how we tend to work, we usually add files the files we wish to commit to the index when we feel we are ready. We then make sure that everything looks good, and if so, make a commit. On the other hand if we spot

something (like a typo, or if we missed a minor detail), we make our edits, add those files again to the index, and then commit the files. Wash, rinse, repeat.

SHARPEN YOUR PENCIL

Time to experiment. Navigate to `headfirst-git-samples` directory, and create a new directory called `play-with-index`, and then `cd` into this directory Go ahead and initialize a new repository using `git init`. Using your text editor create a new file in the `play-with-index` called `multiple-add.txt`. **After each step, draw what the working directory and the index look like**:

1. The initial contents of multiple-add.txt should be "`This is my first edit`". Be sure to save the file!



2. Switch back to the terminal, and use `git add multiple-add.txt` to the index.

3. Back in the editor, change the text in the file to "This is my second edit". Again, be sure to save.

Use this space for your drawings.

4. Back at the terminal, add the file to the index again.

Computer, status report!

As you continue to work with Git, it's often useful to check the status of the files in your working directory. One of the most useful, and, most used commands in your Git arsenal is the **`git status`** command. This command is particularly

useful as your project grows in size, with multiple files.

> **NOTE**
>
> Remember that the working directory is the directory containing the hidden .git folder.

So let's explore how to use the status command: you'll create Yet Another Git Repository™ except this time you will create multiple files in your repository. This will give you a change to see what `git status` reports, and get an intuitive sense of how Git works.

As you have done before, you will create a brand new folder inside the umbrella `headfirst-git-samples` folder called **`ch01_03`**, and initialize a Git repository inside that folder.

Despite not having done anything, you can still check the status of our repository. The command, like others that we have used, is a Git subcommand, called **`status`**. Let's use that.

Your first ever usage of `git status` may seem like a little bit of a letdown, but it does give you a chance to get used to reading it's output. Git nicely tells you that you have made no commits yet, and gives us a useful hint on what you should do next.

You should get used to reading the output that Git commands produce.

Next, you will create the first of **two** files. Open a new document in your text editor, and type in the following lines of text.

```
# README

This repository will allow us to play with the git status command.
```

README.md

Be sure to save the file as **`README.md`** in the **`ch01_03`** directory.

Do the same thing to create **another** file called **`Checklist.md`** with the following text.

```
# Checklist
- [ ] Create two files, README.md and Checklist.md
- [ ] Add README.md and make a commit
- [ ] Update Checklist.md, then add it and make a commit
```

Checklist.md

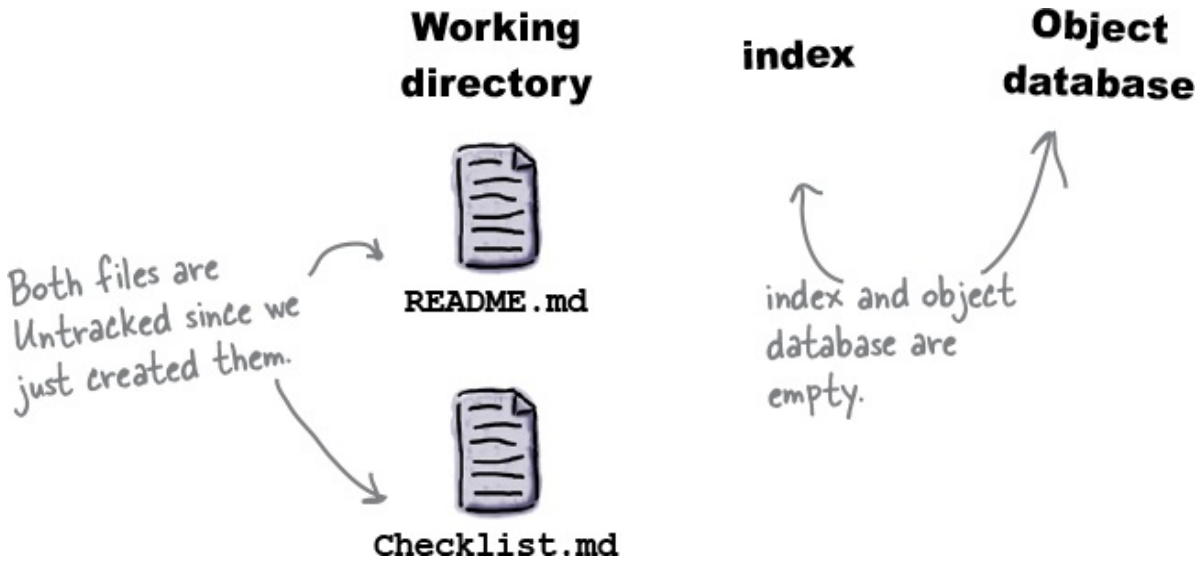
Whoa, easy tiger!

We have done quite a bit very quickly. Let's recap what you have done so far. You created a new folder, and initialized a brand new Git repository inside that folder. You then created two new files.

Now we will walk Git through its paces, and at every step, ask Git what it thinks what the status of the files are. Ready?

You have set up everything to get started. Let's see what `git status` has to report.

What we have:

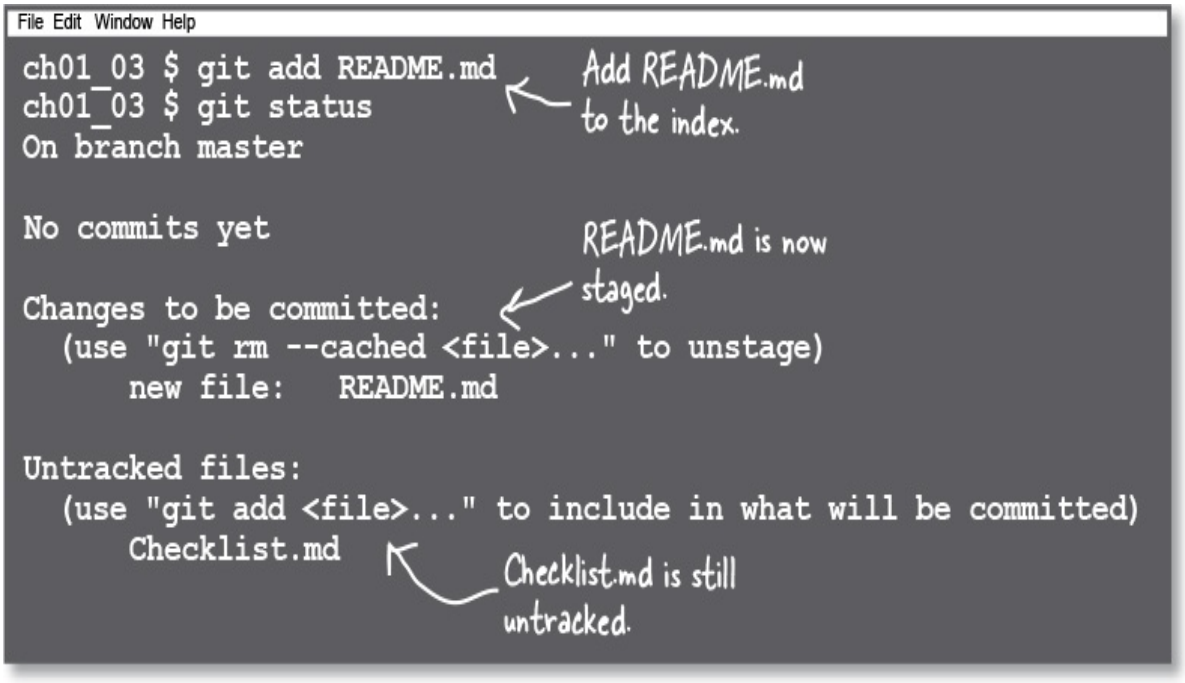


Recall that when you ask Git for the status of the repository, it compares what it knows about our files with what it sees in the working directory. In this case, Git sees two **new** files that it has never seen before. So it marks them as "*Untracked*"—in other words, Git has not been introduced to these files, so it is not watching these files just yet. The index is empty since we haven't added either of the files to the index, and the object database has no commits—well,

since we haven’t commited yet. Let’s change that!

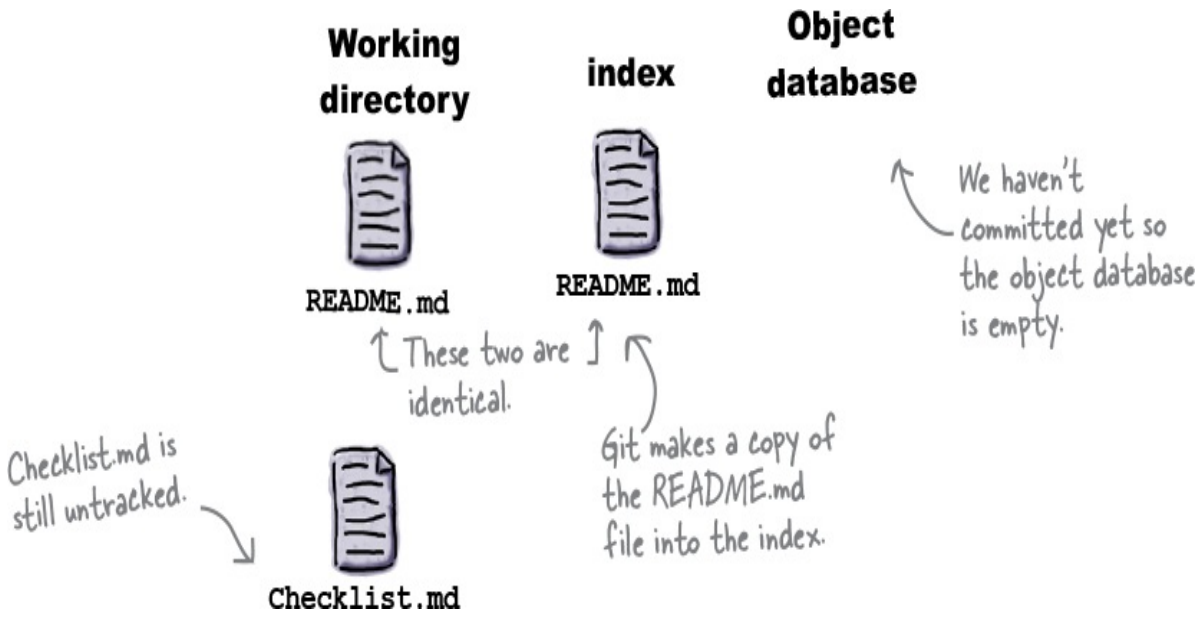
We’ll start by introducing Git to **one** of our files. **Go ahead and add `README.md` to Git**, and then check the status again.

> **NOTE**
>
> Let Checklist.md be for now. We will come to it in a few.



What just happened? Adding the README.md file to Git’s index means now Git knows about this file. Two things changed—the README.md file is now being tracked by Git, and it is in the index, which means it’s also staged.

What we have:



Git tells us that if were to make a commit at this point, only the README.md will be committed. Which makes sense because only the changes that are staged get to participate in the next commit.

So let's commit!

BRAIN POWER

Before you proceed, can you visualize what would change if we were to make a commit right now? Remember, there are two files, and only one is in the index.

Git commits require that we pass in a message. Let's keep it simple and use "`my first commit`". Back to the terminal you!

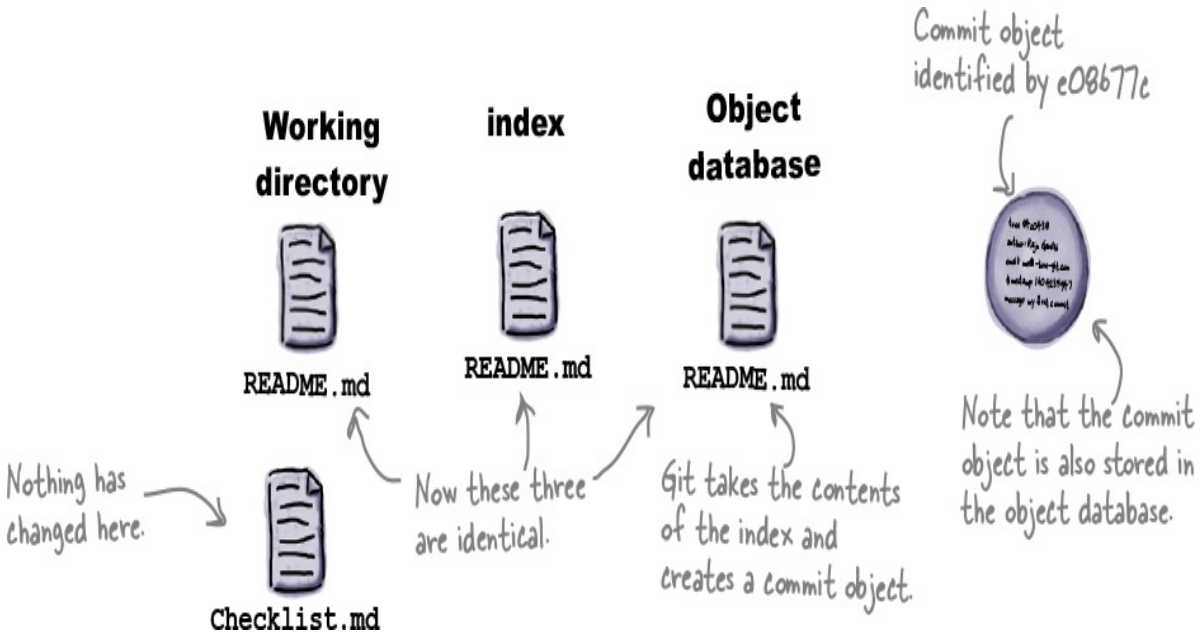
```
File Edit Window Help
ch01_03 $ git commit -m "my first commit"
[master (root-commit) e08677c] my first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md


ch01_03 $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
      Checklist.md

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

Make our first commit in this repository, supplying it a commit message.

Git reports a successful commit.

In our case the commit ID is e08677c. Yours will be different.

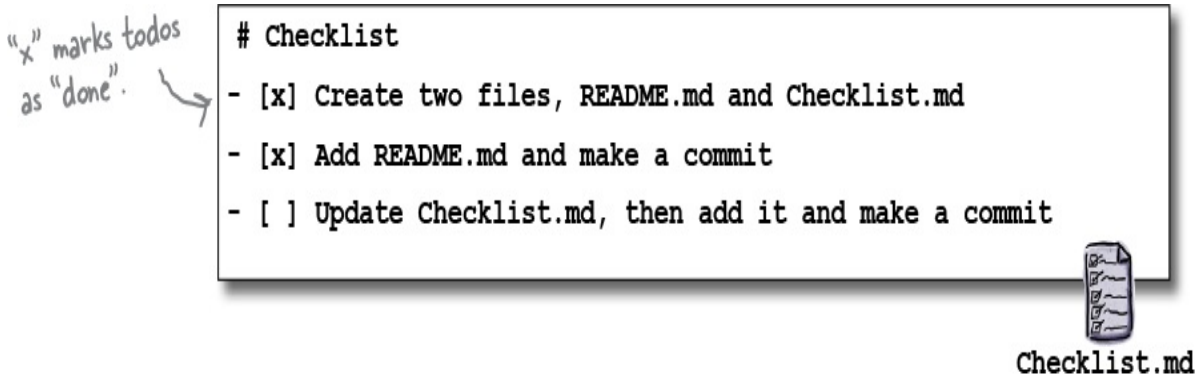
What we have:



Test Drive

The `ch01_03` repository still has one untracked file, namely `Checklist.md`.

Edit it to look like this.



Perform each of the steps below, each time noting the output of `git status`.

1. Add Checklist.md to the index.
2. Make a commit with the commit message “my second commit”.

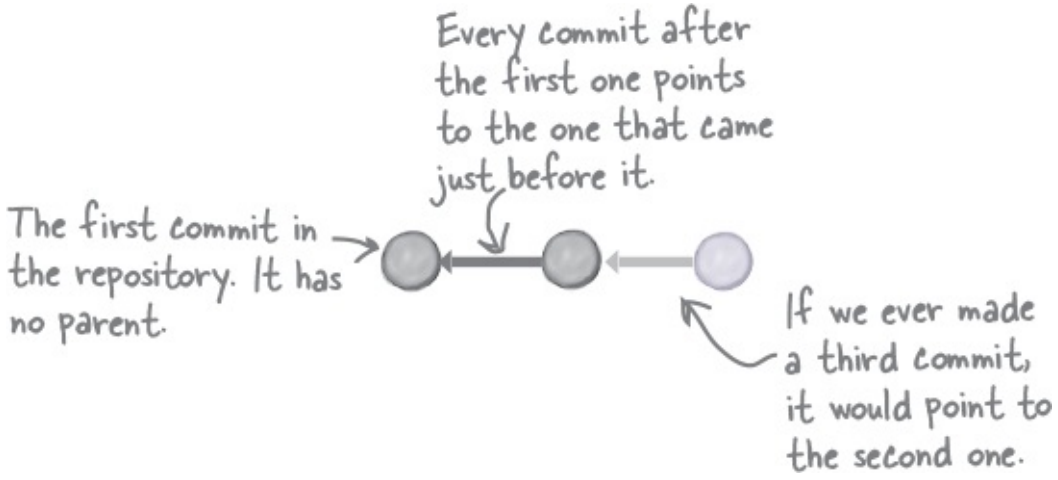
You’ve made history!

In our last exercise we made two seperate commits as we took both the `README.md` and `Checklist.md` files from being untracked, to being staged, and then finally committed to Git’s object database. At the end of it all, our repository now has two commits.

We know that Git commits are essentially metadata that record what was committed, along with information about the author as well as the commit message. There is one final detail about commits that you ought to know about. For every commit that you make (other than the very first one in a repository) the commit also records the commit ID of the commit that came just before it.

That is to say, the commits form a chain, much like the branch of a tree, or a string of christmas lights. Given a commit, Git can trace it's lineage by simply following the "parent" pointer. This is referred to as the commit history, and is an integral piece to how Git works.

> **NOTE**
>
> In case you are wondering if this is a segue of what is to come, well, good job young jedi!



Just know that child commits refer back to their parents, but parents do not refer to their children. In other words, the pointers are unidirectional.

SERIOUS CODING

The Git commit history is often referred to as a *directed acyclic graph*, or DAG for short, wherein the commits form the "nodes" and the pointers to the parent form the "edges". They are *directed* because children point to parent, and *acyclic* because parents do **not** point back to their children.

BULLET POINTS

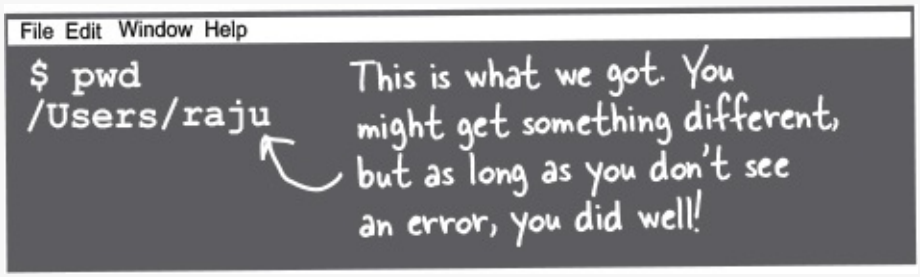
- A version control system like Git allows you to store snapshots of your work.
- Git is much more than a tool that allows you to record snapshots. Git allows us to confidently collaborate with other team members.
- Using Git effectively requires you to be comfortable with the command line.
- The command line offers a slew of other capabilities including creating and navigating directories and listing files.
- Git is available as an executable, which you install, and it makes Git available to use in the command line with the name `git`.
- Once you install Git, you need to tell Git your full name, and your email address. Git will use this whenever you use Git to take a snapshot of your work.
- If you wish you to Git to manage the files for any project we have to initialize a Git repository at the root level of the project.
- To initialize Git you use the init command, like so: `git init`
- The result of initializing a new Git repository is that Git will create

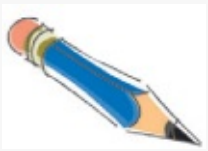
a hidden folder called `.git` in the directory where you ran the `git init` command. This hidden folder is used by Git to store your snapshots, as well as some configuration for Git itself.

- Any directory that is managed by Git is referred to as the working directory.
- Git, by design, has an index, which acts as a "staging area". To add files to the index, you use the `git add <filename>` command.
- Committing in Git translates to taking a snapshop of the changes that were stored in the index. The command to create a commit is `git commit`, which requires that you supply it with a commit message to describe the changes you are commiting, using the `-m` (or `--message`) flag: `git commit -m "some message"`
- Every file in the working directory is assigned one or more states.
- A brand new file added to the working directory is marked as "Untracked" which suggests that Git does not know about this file.
- Adding a *new* file to Git's index does two things—it marks the file as being "tracked", and creates a copy of that file into the index.
- When you make a commit, Git creates a copy of the files in the index and stores them in the object datastore. It also creates a commit object that records metadata about the commit, including a pointer to the files that were just stored, the author name and email, the time the commit was made, as well as the commit message.
- Every commit in Git is identified by a unique identifier, refererred to as the commit ID.
- At any time you can ask Git for the status of the files in the working directory and the Git repository, using the `git status` command.
- Every commit **except** the initial commit in Git stores the commit ID of the commit that appeared just before it, thus creating a string of commits, like leaves on a branch.
- This string of commits is referred to as the commit history.

SHARPEN YOUR PENCIL SOLUTION

Time to get busy! Fire up the terminal, and use the `pwd` command. Jot down the output you see here:





SHARPEN YOUR PENCIL SOLUTION

Your turn. In the terminal window you have open go ahead and use `mkdir` to create a new directory called `my-first-commandline-directory`.

We invoke the mkdir command, supplying it the name of the new directory as an argument.

```
mkdir my-first-commandline-directory
```

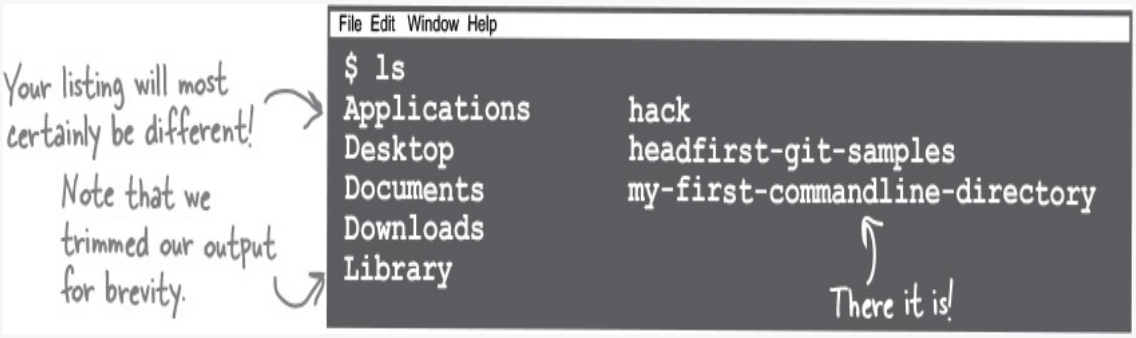
Next, run the same command again, in the same directory. Write down the error you see here:

```
mkdir: my-first-commandline-directory: File exists
```

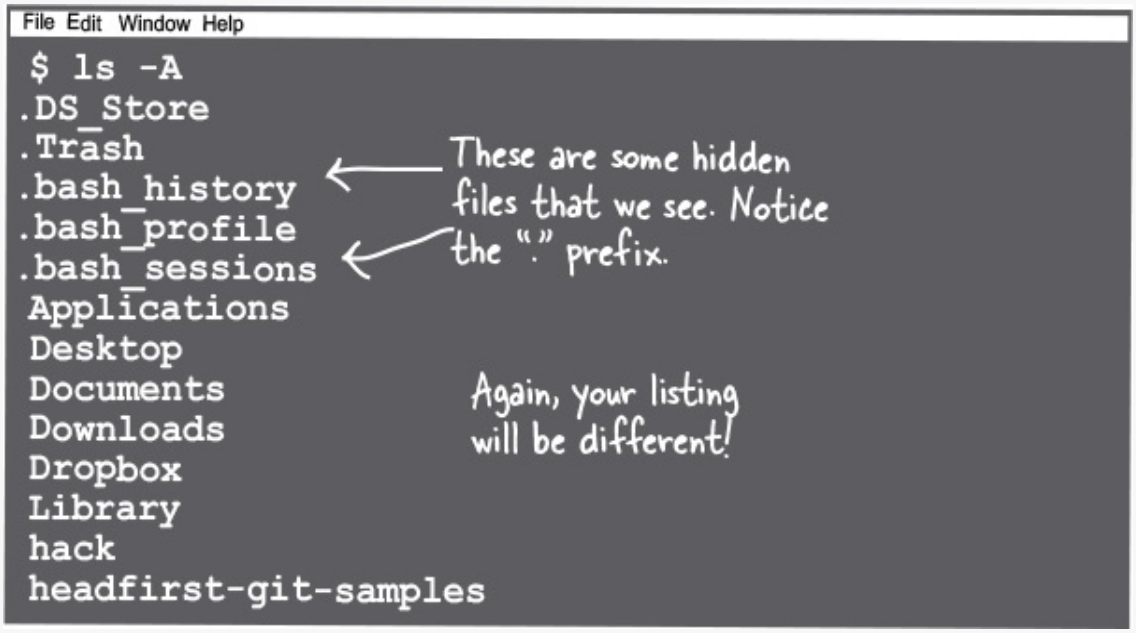
mkdir errors out if the directory already exists

SHARPEN YOUR PENCIL SOLUTION

Use the terminal to list all the files in the current directory. See if you can find your recently created `my-first-commandline-directory`.

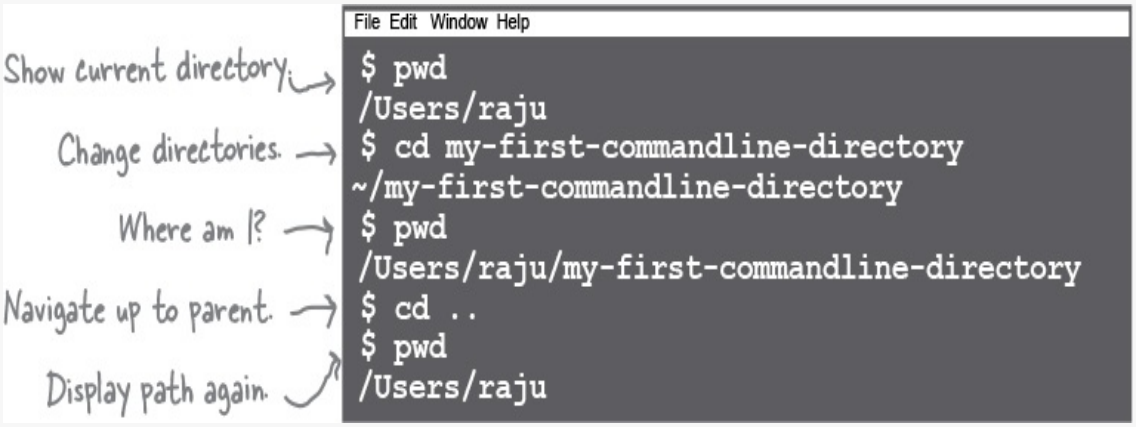


Then use the `-A` flag and see if there are any hidden folders in the current directory.



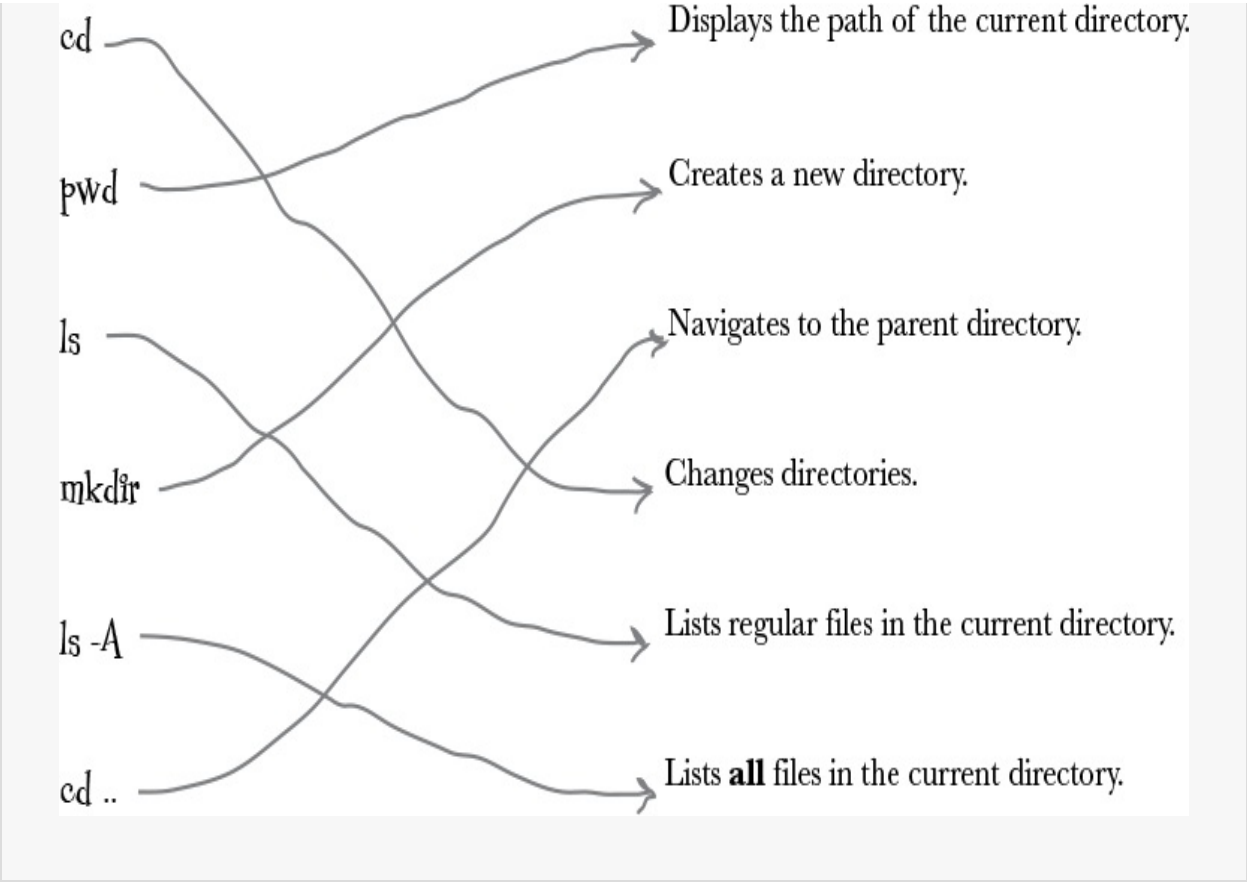
EXERCISE SOLUTION

Go ahead, give it a spin. Play around with `cd` to hop into your newly created `my-first-commandline-directory` folder, then use `pwd` to make sure you did change directories, and then use `cd ..` to go back to the parent folder. Use this space as a scratchpad to practice out the commands as you use them.



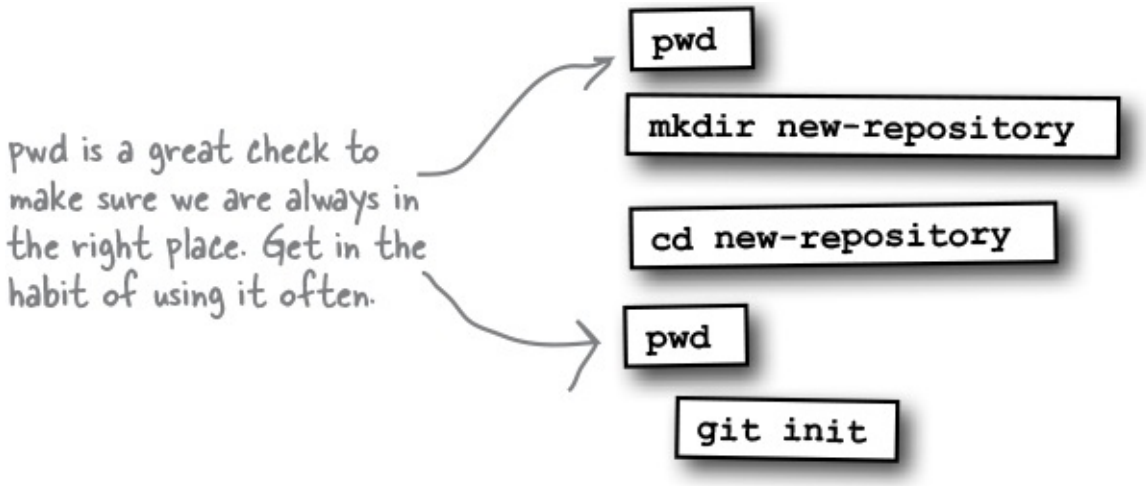
WHO DOES WHAT? SOLUION

With the command line there's a lot of commands and options flying around. In this game of who does what, match each command to its description.



Magnetic thoughts

We have all the steps listed to create a new folder, change to it, and initialize to create a new Git repository. Being diligent developers, we often check to make sure we are in the correct directory. To help our colleagues we had the code nicely laid out on our fridge using fridge magnets, but they fell on the floor. Your job is to put them back together. Note that some magnets may get used more than once.



BE GIT

Recall that any file in your working directory is either untracked or tracked. Also, a tracked file can be either staged, unmodified, or modified.

In this exercise assume you just created a new repository. Can you identify the state of the files for each of the following steps?

You create a new file in the repository called Hello.txt.

Untracked	Tracked	Staged	Unmodified	Modified
X				

You add Hello.txt to the index.

Untracked	Tracked	Staged	Unmodified	Modified
	X	X		

You commit all the changes that you staged.

Untracked	Tracked	Staged	Unmodified	Modified
	X		X	

You edit Hello.txt with some new content.

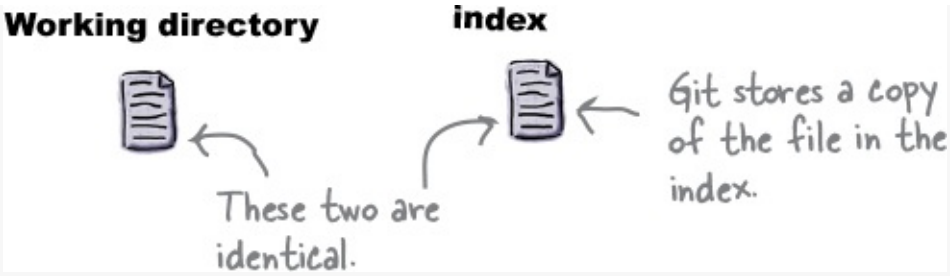
Untracked	Tracked	Staged	Unmodified	Modified
	X			X

SHARPEN YOUR PENCIL SOLUTION

Time to experiment. Navigate to `headfirst-git-samples` directory, and create a new directory called `play-with-index`, and then `cd` into this directory Go ahead and initialize a new repository using `git init`. Using your text editor create a new file in the `play-with-index` called `multiple-add.txt`. **After each step, draw what the working directory and the index look like**:

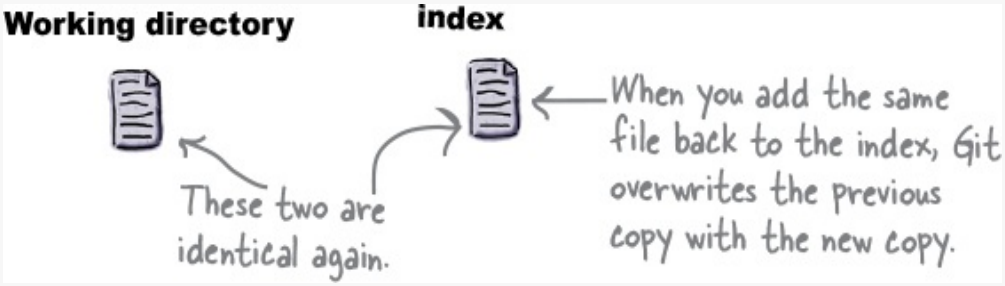
1. The initial contents of multiple-add.txt should be "`This is my first edit`". Be sure to save the file!

2. Switch back to the terminal, and use `git add multiple-add.txt` to the index.



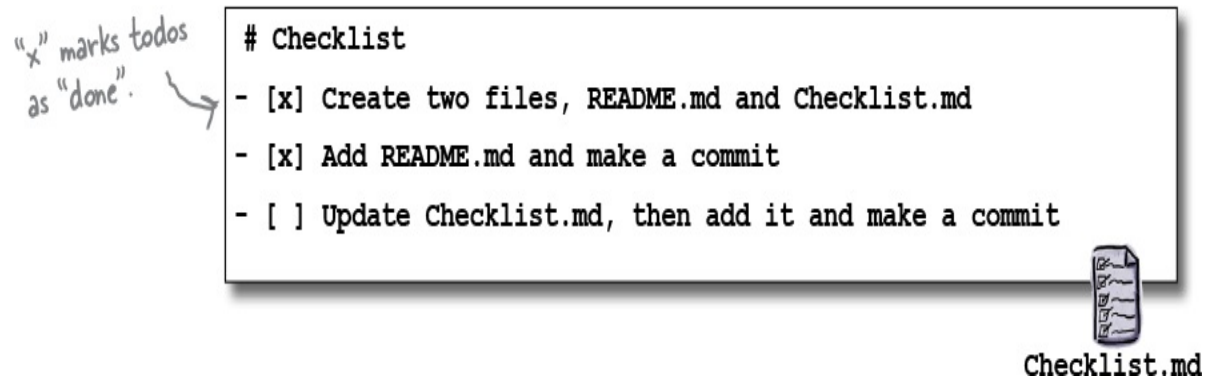
3. Back in the editor, change the text in the file to “This is my second edit”. Again, be sure to save.

4. Back at the terminal, add the file to the index again.



Test Drive Solution

The `ch01_03` repository still has one untracked file, namely `Checklist.md`. Edit it to look like this.



Perform each of the steps below, each time noting the output of `git status`.

1. Add Checklist.md to the index.

```
File Edit Window Help
ch01_03 $ git add Checklist.md
ch01_03 $ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
      new file:   Checklist.md
```

2. Make a commit with the commit message “my second commit”.

```
File Edit Window Help
ch01_03 $ git commit -m "my second commit"
[master 77bcd74] my second commit
 1 file changed, 6 insertions(+)
 create mode 100644 Checklist.md
ch01_03 $ git status
On branch master
nothing to commit, working tree clean
```

1. 1. Beginning Git: Get Going with Git
    a. Why we need version control
    b. Cubicle Conversation
    c. Start your engines ...
    d. A quick tour of the command line
    e. More on the command line (mkdir)
    f. More on the command line (ls)
    g. More on the command line (cd)
    h. No argument there
    i. Cleaning up
    j. Creating your first repository
    k. Inside the init command
    l. Magnetic thoughts
    m. Introduce yourself to Git
    n. How you will use Git
    o. Putting Git to Work
    p. Meanwhile, back at the HawtDog Dating Service ...
    q. Speaking of ...
        i. Congratulations on your first commit!
    r. What exactly does it mean to commit?
    s. Look before you leap
    t. The three stages of Git
        i. Great question!