## Understanding Git Branch Basics

### Branching In-flight

Hi, this is Craig Golightly, and welcome to working with Git Branches, Understanding Git Branch Basics. as you work in different codebases, you may notice that some branches are called master. This refers to the English definition for the adjective master, which means main or principal. Like the main water line going into a building, it supplies everything to all of the branches. In October 2020, GitHub changed the default branch name for new repositories from master to main. They also enable organizations to set whatever default branch name they choose. In this course we will use the term main for the first or principal branch in a repository, since that is the current convention. Just be aware that as you work in existing repositories, the principal branch will probably be named master. If one person could write and understand all of the code in a project, then there would never be any conflicts. The reality is that most systems are large and complex and require several teams of developers to create and maintain all of the code needed to support those systems. We're going to look at how branching in Git can help make your life easier as you work on projects, both big and small. The first case we're going to look at is what I call the oops path. Maybe you started making a quick fix on main but realized it wasn't so quick, and you need to do something else on main before you can finish your fix. How can you save your work and come back to it later? You can create a branch to commit all of the existing changes. You can then continue working in that branch now or later, and if you need to do something else on main, when you switch back, it will be clean. Let's go through an example on the command line to see this in action. We'll use the command line, since it's available in any environment, and it helps you understand what is happening step by step. We'll initialize a directory as a Git repository, add some files to represent the current state of main, make some changes to those files, then move those changes into a branch. Let's get set up for this demo. I've included a readme with all of the examples and the steps to follow if you want to do this on your own machine. I'll also try to go slow enough that you can follow along here in the demo. I'm here in an empty directory. Let's initialize this directory as a Git repo by running the git init command. We'll add a file for our bug. We'll add some sample text, save the file, then add and commit it to Git. If we run git status, we'll see that we have no pending changes. Let's go and fix the bug. There are several things to fix. We need to get rid of this line because it's old; we need to fix this typo, which is the real problem; and we actually need to make two new files called function1 and function2. We'll move this code into a different file, then make a file for function2. (Typing) Many times you may go into what seems like a simple bug, but it ends up being more involved. We can check the status now and see that we've modified a file, and we've added two other files. At this point, we're still on the main branch. Let's go ahead and move these changes into their own branch. We can create a branch with these changes by running git checkout ‑b and the name of the branch. I'll call it quickfix. Let's stage all of these changes for commit by using git add, we'll check the status, and finally commit all of our changes. At this point, if we switch back to the main branch by running git checkout main, we can see by listing the files that we only have the original bug.txt. The new files that we've created have been checked in on the quickfix branch. This is a way that branches can help you organize your work. If you got into the fixes and they were more involved than you thought and you needed to come back to main to work on something else, you can do that. When you're done, you would switch back to the quickfix branch by running git checkout quickfix. Now that we're on the quickfix branch, we can list our files and see that we have all of the changes that were made. Let's review some of the key commands from the demo. If you're on a branch and have some changes, could be files you've modified or files you've added, you can create a branch called quickfix and switch to that branch with the git checkout ‑b option, followed by the name of the branch. Next, you can stage the changes for commit with git add, then actually commit them to the quickfix branch with git commit. Now, when you switch back to main using git checkout main, you won't see any of the new changes you made. Those are safely checked into the quickfix branch for you to get back to whenever you're ready.

### Moving from Branch to Branch

You may think of something like this when you hear jumping around branches or moving from branch to branch. In Git, branching is very lightweight, so using branches is really quite easy. Let's review the basic syntax, then get into some more examples. Suppose you're not sure about how to tackle a certain problem. If you just start trying something out and find that it doesn't work, how do you get back to a good starting point? Delete it and start over? Ctrl+Z? What if you later realized that you actually had most of the solution, it just needed a little more tweaking to be complete? If you deleted or undid everything, you can't get it back. This is another case where branching can help. You can create a branch when you start the new work. You can also create a branch when you're starting something that you're not quite sure of. If the code isn't working out how you want it, you can also use a branch to preserve what you think is the wrong path, but might actually be partially correct. If you end up needing some code from a previous attempt, it will be there in the branch. This can help you keep things organized and separated, yet easily go back if you need to until you arrive at a solution. Let's go to the command line for an example to see this in action. First, we'll create a branch to solve a problem. This first attempt won't work, so we'll commit those changes to the branch. We'll switch back to main to get a clean start and try again using a second branch. While working, we'll realize that we can use something from the first attempt, so we'll commit the partial solution, switch back to the branch for the first attempt, and copy what we need, then go back to the second branch to complete the solution. For a lot of these demos, we'll go ahead and just initialize a new directory. You can do this on your own machine as long as you have Git installed. Here, I have an empty directory, and I'll go ahead and initialize it as a Git repo using git init. I'm now on the main branch. I'll make a file called experiment.txt to set up this demo. We'll add some text to this file. Again, each demo is outlined in a README that you can download in the exercise files for this course. We'll save and commit this as our starting point, using the git add and git commit commands. Recall that the ‑m allows us to add a message to the commit right from the command line. Now we're ready to start on this difficult problem. We'll go ahead and create a branch right from the beginning because we know it might take a while. I'll call it first‑try, so I'll run git checkout ‑b to create a new branch, then the name of the branch, first‑try. Now that we're in the first‑try branch, we can start making some changes. I'll add in some text to represent some different steps for this problem. And after working on this for a while, maybe it didn't quite work. Rather than throw this away, we'll go ahead and commit it to this branch, then create another branch to try a different approach, so git add and git commit. (Typing) Now we'll switch back to main to get a fresh start. We're not quite sure if this is going to work out, so we'll go ahead and create a different branch to try a second time. We'll use the git checkout ‑b, and let's call this branch second‑try. Now that we're on the second‑try branch, we'll go back to our file, and it's in the original state. We'll add some text to represent our second attempt at the problem. And suppose, at this point, you realize that some code from your first attempt can be combined with this second attempt to get a complete answer. To accomplish that, we'll go ahead and commit these changes to the second‑try branch, then switch back to the first‑try branch so we can look over the code from our first try. Again, git checkout first‑try will switch us to the first‑try branch. Back in our editor, we can see the code from the first attempt. Suppose you needed this step 2. We'll copy that, switch back to the second‑try branch, then add our code to the solution. At this point, suppose you've solved the problem. We'll go ahead and commit it and leave a message to indicate that our solution is complete. Branches can really help you keep things organized as you tackle difficult problems and help you preserve the history of what worked and what didn't work. This can save you a lot of time and frustration while keeping things organized if all a sudden you need to go back to main and fix something on the current code base. We used a couple of new commands in that demo, so let's take a look at those. Git branch will list all of your branches and show you which one you are currently working on. When you want to switch to a different branch, run git checkout, followed by the name of the branch you want to work on. Git also has a command called switch to change what branch you're working on. The syntax is similar to checkout in that you'll type git switch, then the name of the branch you want to switch to. To create a new branch and switch to that branch, checkout and switch are similar, except checkout uses a ‑b flag for branch, and switch uses a ‑c flag for create. Since the git checkout command can be used to do many different things, git switch was introduced to provide a more focused, simple command for switching branches. They both work, so you can use either one. When you've made changes to a branch, but you have not yet committed those changes to the branch, that state is sometimes referred to as a dirty branch. In this case, the word dirty is being used as the opposite of clean. It won't let you switch from a dirty branch that has uncommitted changes until you clean up your changes. This helps prevent you from unintentionally overriding or deleting code from your repository. For now, we'll clean up changes by committing them to a branch. Let's see what happens when you make a change on a branch and attempt to move to another branch before you commit the change. We'll then resolve the error message that happens in that scenario so that we can switch branches. Let's pick up where we left off on the last demo to try this out. We're in the second‑try branch, and we have a solution that we've developed. Let's go ahead and add a new change to this file, then try to check out the first‑try branch. Notice the error message that tells us that we've made changes to a file that's been checked into second‑try. And if we switch over to the first‑try branch, we're going to lose those changes. We'll solve this by committing our changes to the second‑try by doing a git add and git commit. (Working) Now that we've cleaned up our branch from any changes, we can now check out the first‑try branch and successfully switch over. When you change branches, Git automatically updates the filesystem for files managed by Git. This allows any open editors to reflect the differences in files between one branch and another. It may seem like magic if you don't know what's going on, so be sure to notice which branch you currently have checked out. Most integrated development environments, or IDEs, will give some indication in your environment about what branch you are actually working on. For example, let's look at the experiment.txt file open in an editor. Running git status shows that we're in the first‑try branch. If we check out main using git checkout main, notice how the contents of experiment.txt are changed to reflect the state of that file in main. If we switch back to the first‑try branch, notice how the file changes back to reflect the state of the file in the first‑try branch. Branches can also help when you get stuck waiting for something. Suppose you are working on a ticket, but are blocked by an incomplete requirement. You probably don't have the luxury of taking a break until someone gets back to you since you have other tickets you need to finish that are in different areas of the project. You can create a second branch from main to work on another ticket while you're waiting. When you get the information you need to complete the first ticket, you can simply switch back to that branch and continue your work. Let's take a look at this example on the command line. Let's go through an example of starting work on a ticket in one branch, getting blocked, then creating a new branch off of main to start on a different ticket while you wait. Once you receive the information you need to unblock the first ticket, you'll switch back to that branch to continue work where you left off. We'll start this example in an empty directory. We'll initialize it as a Git repo and add a file to represent the main branch of your code. We'll add and commit the file (Typing) and check our git status. Now that we have a clean working tree, let's create a ticket1 branch to start working on ticket1. Remember, git checkout ‑b to create a branch, then the name of the branch. We'll do some work on ticket1 using ticket1.txt and add this text to represent the work. As you're going along, you're blocked on a missing requirement. At this point, you could sit around and complain or do nothing, but we can use Git branches to help us preserve this and move on and work on something else from main. If we check our status now, we can see that we have untracked files, our new ticket1.txt. Let's go ahead and add and commit that to the ticket1 branch. (Typing) If we list the directory, we can see ticket1.txt and main.txt. Git status confirms that our working tree is clean. Let's switch back to main. If we check the contents of main, notice that ticket1.txt isn't there. It's committed in the ticket1 branch. At this point, we can create a ticket2 branch and start work on a different ticket. Let's make a file called ticket2 and do some work on ticket2. Suppose in the middle of working on ticket2, we receive the missing requirement for ticket1. We can save our spot on ticket2 using the ticket2 branch. We'll add and commit the file. And now we can switch back to the ticket1 branch. As I was saying switch branches, I typed git branch ticket1. Normally, this would create a branch called ticket1. Let's go ahead and use the git switch command. Now we're on the ticket1 branch. Notice how we don't have ticket2.txt. We only have ticket1.txt. We'll go back to our editor, make the change, and commit the completed requirements. (Typing) Now you can go back to the ticket2 branch and continue where you left off. Now we can jump right back into where we left off and continue work on ticket2. Notice how easy Git makes it for you to switch from task to task by using branches. All of your work is encapsulated in each branch, and you won't get things mixed up between different tasks that you're working on. The filesystem is automatically updated. So while it may appear like things are disappearing, that's just Git doing its job. When you switch back to the branch where you created something, it's there for you. The end result is that you can keep your editor open, and you don't have to keep worrying about opening odd folder structures that you may try to create if you attempt to manage this on your own without using branching.

### Renaming and Deleting Branches

You may have heard that there are two hard problems in computer science, cache invalidation, naming things, and off‑by‑1 errors. Suppose you don't like the name of a branch. No problem, you can rename it. To rename a branch, use git branch with the ‑m flag, followed by the current name of the branch you want to rename, then the new name you would like to give to that branch. For example, git branch ‑m quickfix longfix. This will rename the quickfix branch to longfix. If you want to rename the branch you are currently working on, then Git has a shortcut for this. Simply provide the new name after the ‑m. For example, to rename the current working branch to hotfix1, run git branch ‑m hotfix1. We'll start out in an existing repo and list the current branches with git branch. We're currently on the main branch, and there's another branch called quickfix. While on main, we'll rename quickfix to longfix by running git branch ‑m for rename, the current branch name, quickfix, what we want to change it to, longfix. If we run git branch again, we can see that the branch has been changed from quickfix to longfix. Let's go ahead and switch to the longfix branch by writing git switch longfix. Recall that we can also say git checkout longfix. I'm just showing that you can use either one. Let's use the shortcut to rename the current branch, longfix, to hotfix1, git branch ‑m, then the name of the new branch. We can run a git status and see that the current branch is hotfix1. We can also run git branch to list the current branches. The longfix branch is no longer listed because we renamed it to hotfix1. If you're not careful, you could find yourself surrounded by branches on your workstation. Well, maybe not, but you might not like scrolling through pages of branch names every time you run git branch. Deleting branches in Git is simple and painless and does not require you to climb a tree or use a chainsaw. You'll use the git branch command, but now you'll pass in a ‑d for delete, followed by the name of the branch to delete. For example, git branch ‑d longfix. This will delete the branch named longfix. If you try to delete a branch that has commits that have not been merged to another branch, you'll get an error message. This is because you'll lose work that only exists on that branch. If you're still sure you want to delete the branch anyway, you can pass in the ‑D flag to force the branch to delete. Git also does not allow you to delete your current working branch. If you need to delete a branch that you are working on, you must first switch to another branch. I'm here on the hotfix1 branch. If I try to delete this with git branch ‑d hotfix1, notice the error that I get. I can't delete the branch that I'm on. If I switch to main and try deleting it again, I've now got a different error message. Hotfix1 contains things that have not fully been merged. In this case, it was branched from main, some changes were made, but they haven't been merged back. In this case, I know that I don't want this branch anymore, so I'll use the ‑D flag to force the delete. Running git branch again confirms that that branch has been deleted. Finally, let's just run a normal delete. I'm on main, and I have a couple of sample branches that I just need to get rid of. In this case, I can simply run the git branch ‑d and the name of the branch. By running git branch, I can confirm that those branches have been deleted. Once you've initialized a directory as a Git repo, files will behave differently on your local file system when you switch branches, based on what state they're in. Files that are untracked have not been added with git add or committed with git commit. These files will stay in the directory no matter what branch you're in. Staged files have been added via the git add command and are ready to commit. If you switch branches, any staged files will remain in the staged state on the new branch because they don't yet belong to any specific branch. Committed files belong to the branch where the git commit command was run. If you switch to a different branch after committing changes, you will not see those changes in the other branch. Understanding how Git sees files and treats them can help you know what to expect when you're switching between branches. Let's have a look at file states. I'll initialize this empty directory and add and commit a file to main. Now we'll create a sample branch from main. We'll list our available branches then add a new file. At this point, if we run git status, we'll see that new.txt is an untracked file. If we switch back to main, notice when I list the directory, new.txt is there. That's because it's still an untracked file in the directory. We can run git add to stage the file for commit, and if we switch back to the sample branch, notice that the file is still there and it is still in the staged state. At this point, let's commit the file to the sample branch. Now if we run git status, we'll see that there is a clean working tree and both files exist in this directory. If we go back to main, notice how when we list the files, new.txt is no longer there. We've committed it in the sample branch, so Git is tracking it in the sample branch. Main is clean with no untracked files and nothing staged. Way to go! You now know the basics of branching, creating, renaming, deleting, and how to move from branch to branch. You've also seen some cases where branching can help you out in your day‑to‑day work. We covered the oops case, when a quick fix into something bigger and you want to isolate it; the problem‑solving case, where branches can help you preserve different approaches to a problem; and how to use branches to easily switch tasks when you get blocked. Finally, you saw how Git manages files on your file system and the difference between untracked, staged, and committed files. Join me in the next module to learn about merging, which helps all of the work you do in branches come together.

## Merging Made Easy

### Comparing and Merging Branches

Hi, this is Craig Golightly and welcome to Merging Made Easy. Recall our happy path, where you figured out the solution to a problem on a branch off of main. You may have been asking yourself, okay, Craig, that's great if I figured out the solution on a branch. How do I get that back into main? The answer is merging. If the word merging makes you think of something like this, don't worry. Merging in Git is easy, and this scenario is the easiest of them all. If no additional work has occurred on main, then Git can perform a fast‑forward merge. This is where the reference for main is simply moved forward to the last commit on the branch you are merging. Think of it this way. You started watching a movie on your TV at home. You got 30 minutes into it and had to leave. This would be the TV branch. While you were gone, you were able to use your phone to watch another 15 minutes from where you left off. So now you're at minute 45 in the movie on the Phone branch. When you get home, your TV will fast forward to minute 45 so you can resume watching as if you had been watching on the TV the whole time. Fast‑forward merges are the same way. Because there was no conflicting activity in the main branch after the branch was created, it's like we were working there the whole time. You can think of the commits all happening in a linear sequence. We'll be using the terms target and source throughout our merging examples, so let's make sure we're clear on the meaning of those terms. Target refers to the branch you want to modify by merging changes in. It is the target of the merge. In this example, the main branch is the target. Source refers to the branch that has the changes you want. It is the source of the merge. In this example, the solution branch is the source. When you merge the source branch into the target branch, only the target branch changes. The source branch remains the same. In this example, main changed as a result of the merge. Even after adding new commits to main, you could switch to the solution branch, and it will be the same as it was before the merge, as long as you haven't committed anything else to that branch. Behind the scenes, it would look more like this. The solution branch continues to reference the last commit that was made in that branch. When the merge happens, main also references that same commit. Once a new commit is made in main, then main will move on to reference that commit, while the solution branch stays the same. To merge one branch into another branch, first switch to the target branch by using git checkout then the name of the target branch. Next, merge in the source branch by using git merge, followed by the name of the source branch. For example, if you were on the ticket1 branch and wanted to merge it into main, you would run git checkout main, then git merge ticket1. Let's perform a simple fast‑forward merge. We'll create a branch off of main and do some work in that branch, committing our changes. We'll then switch back to main and merge the new branch into main. The result of the merge will be all work combined in the main branch. I'm in a new directory. So let's initialize it as a Git repo. We'll create a file called begin.txt, add some content, and commit the file. (Typing) Next, we'll create a branch called solution. Remember git checkout ‑b and the name of the branch. Let's create a file called solution.txt. We'll add some text, then add and commit the file on the solution branch. This is a very simple example of the scenario where you created a branch to do some work, you finished the work, and now you're ready to merge that work back to main. First, we'll switch back to the main branch to perform the merge. Remember we can use git checkout or git switch. I'll use git switch. Now we're on the main branch. Notice we only have one file, begin.txt. Solution.txt is checked into the solution branch, and we want to bring that over to main. To merge solution to our current branch, we'll simply run git merge solution. Notice in the output how it performed a fast‑forward merge. This was possible because nothing else happened on main, so it simply added those changes to main, and now we can see both files.

### Understanding Git Diff

You may be thinking, hey, Craig, I don't want to just run git merge and hope things work out. Can I at least see what's going to happen first? You sure can. Let's look at how to compare branches. Git diff can be used to compare branches. The syntax is git diff, then the name of the two branches you want to compare. For example, if you want to see what is different between main and a branch named ticket1, you would run git diff main ticket1. Diff is going to show you any changes between the tips of these two branches, meaning whatever is committed in each branch. Running diff in this way doesn't change anything in your file system, so don't worry about messing anything up. This will simply produce a report of what is different between the two branches. Git diff is a powerful tool with many different options, so let's dive a little deeper so you can understand it better. When you run git diff from the command line, the output is in a certain format. Let's go through the different pieces and what they mean. The first line shows the names of the files being compared in the diff. Git labels them a and b. In this case, we're looking at two different versions of the same file, example.txt. Next, Git shows the hash of the a file and the b file. If you want to view just the a or b file by itself, you can run git show and pass in this hash. The last number has to do with file permissions, and we're not going to use that right now. The next two lines show the file markers that will be used in the diff. File a will use dashes, and file b will use plus signs. Dev/null will be listed instead of a file name if the file has just been created or deleted. This last part is probably the most confusing thing when you see it for the first time. The double @ symbols are just markers for the chunk header. This line here is called the chunk header. The diff may contain multiple chunks, and each one will have its own header. The numbers correspond to lines in the file. The first pair of numbers with a dash indicate that there are six lines displayed from the a file, starting with line 3 of that file. Recall that the file marker for the a file is the dash. The next pair of numbers with a plus sign indicate that there are 7 lines displayed from the b file, starting with line 3 of that file. Recall that the file marker for the b file is the plus sign. The lines that don't have a dash or a plus in front of them are the same in both files. Most IDEs have built‑in support for diff so that differences are graphically displayed with colors and side‑by‑side comparisons. These tools can be helpful when trying to determine differences in your codebase. We're using the command line for these examples in order to understand the information provided by diff and to give you a tool that will work in any environment, as you may not always have access to a graphical Git client.

### Using Git Diff

Let's work through some common scenarios you'll encounter when working on projects and see how the git diff command can help you answer questions about what has changed. You'll compare files and examine both staged and unstaged changes. You'll see how to ignore whitespace changes in your diff, as well as how to compare specific commits in your Git commit history. You'll use the git show command to output what a file looked like at a specific commit in history, then you'll use git diff to compare branches. Both the current state of branches, as well as comparing what changes have happened since a branch was created. We'll start this demo in an empty directory. As a reminder, you can download the exercise files for this course and see all of the commands that I'm running in these demos to help you follow along and review things later. We use git init to initialize this as a git repo and our initial branch is main. Let's create a file called colors.txt. Let's add some content to start out with, save the file. Since we'll be going through a lot of different options for git diff, I'm going to use a list of colors. This allows us to easily see the changes and focus on the mechanics of git diff versus a specific programming language or problem. We'll save and commit the file, then start making some changes so we can use git diff. Let's add a few lines to colors.txt. Now when we run git diff, we'll see the new lines that have been added to the file. Let's stage these changes for commit by running git add colors.txt. Now, if we add one more line to the file, when we run git diff, notice how we only see that last change. This is because gray is the only unstaged change. If we want to see only the staged changes, in other words, what we've already staged using git add, we'll add the ‑‑cached flag to git diff. This output shows us only the staged changes. To see all staged and unstaged changes, we'll use git diff HEAD. Now we can see all of the changes in the file. Let's go ahead and commit all of the changes that we've made. Git status shows that we have a clean working tree, so now if we've run git diff, there is no output. This is because there are currently no differences. We've committed everything. Let's make a few more changes and commit them so we can build up a commit history for colors.txt. (Typing) You'll notice I'm using a combined ‑am flag for commit. This is because I want to just commit all changes. Now let's add a couple more files to our directory. We'll add a file with some letters, (Typing) and we'll add a file with some numbers. If we stage these two files using git add, then run git diff, notice how there is no output. This is because git diff, by default, is only looking at unstaged changes. To see the staged changes, we need to add the ‑‑cached flag. Notice dev/null in the File markers section since letters.txt and numbers.txt are new files. Also notice that each file is listed separately in the diff. On the top, we have letters.txt, and on the bottom, we have numbers.txt. As your diff output starts to get larger and larger depending on your machine and your command line, Git will open a default editor. In this case, it's vi so I may need to use the up and down arrows to scroll through the diff and hit the Q key to quit and go back to the command line. Notice in the output that it indicates no new line at end of file for both letters.txt and numbers.txt. Let's go ahead and add a new line to the end of each of those files, we'll add it to numbers and add it to letters. Now if we run git diff, we'll see that it picks up our unstaged change where we've added a new line to the end of each file. If we want to ignore whitespace changes, we'll add the ‑w flag to git diff. Now there is no output because the only changes were whitespace changes. Let's go ahead and commit our changes up to this point. (Typing) We'll run git status to just check that were clean. And now we've built up a bit of a history that we can use some of our other diff commands. To view the history run git log ‑‑oneline. Here, we can see the four different commit SHAs since starting main, adding the colors, changing some colors, then adding our numbers and letters file. Note that the commit SHAs that you'll have on your machine are going to be different than the ones you see here. Let's make some more changes to colors.txt. Suppose you got the question of how many colors were in the original file versus how many do we have now? We can see that there are 12 colors now, but how do we find out what the original file looks like? Let's run git diff with a commit SHA to compare the initial commit with what is in the current working directory. From our git log oneline output, we can see our first commit, start of main. Run git diff, then the commit SHA from your first commit. Again, your SHA will be different, so just copy whatever you have for that first commit. Now we're seeing the git diff output that we went over earlier. Notice the chunk header in the diff. The original a file is showing six lines and the current b file is showing 12. Seeing both versions of the files can sometimes be a little bit confusing. If you'd like to see just the original file, the index line gives you the SHA that you need to use the git show command. Go ahead and copy the A version, which is the first one after index, exit out of your diff, which again, mine is in vi so I use the up and down arrows to scroll through the diff and Q to exit back to the command line. Now I can run git show, then the index of the A file for colors.txt, which is the original one checked in. I've got that on my clipboard so I'll paste it, and now we can more easily see that there were six lines in the original commit. Let's go ahead and stage the changes that we have up to this point using git add colors.txt, then make a few more changes to the file. Git diff shows us the changes that have not been staged for commit, in this case, four new colors have been added. And again, to view the changes that have been staged for commit, add the cached flag so git diff ‑‑cached. Here, we can see the changes that were made and a couple of colors added. Let's commit all these changes and now we have a file with 16 colors listed. Suppose you got a requirement to replace all of the original colors that were listed in the first commit. How are you going to find out that information? We'll use git diff to compare the current version of colors.txt to the version of the initial commit. We'll run git log oneline to get that SHA again. We'll want this one, start of main, go ahead and copy that, then run git diff with the SHA. In the diff output, notice those lines that do not have any file markers, orange, yellow, and blue. These are the lines that are the same in both files and this answers our question of which original colors do we still need to change? Let's exit out of the diff with Q, then go change those colors. Save the changes, then run the same git diff command against the original commit SHA. Notice now that there are no colors without a file marker. We've replaced all of the original six colors. I'll hit Q to exit out of the diff, and at this point, let's go ahead and make a new branch called colors 2.0 to continue our work. Remember, git checkout ‑b, then the name of the branch. Let's add a couple of more colors and commit the change. (Typing) To see the difference between what is currently in main and what is currently in the colors2.0 branch, we'll run git diff colors2.0 main. We can see the changes that we made to the original colors, as well as the ones that we've added. Suppose that while you're doing your work in the colors2.0 branch, some other work happens in main. We'll go in and do that ourselves now by switching back to main, git switch main, and notice that our colors.txt has changed to what is actually committed in main. Suppose there was a requirement that all color names needed to be one word only. And then that change was made and committed in the main branch. (Typing) Let's switch back to the colors2.0 branch and we'll use git diff to find out what has changed in main since the colors2.0 branch was created. This will help us know what's going on and decide what coordination needs to happen or what changes we may need to pull into this branch. Recall that that is git diff colors2.0...main. Here we can see the change that was made in main where all colors were required to be one word only. Light green was changed to lime and dark brown was changed to chocolate. While in the colors2.0 branch, let's say you found an error in one of the files. The numbers.txt has some duplicate code. Lines 4 and 5 are duplicate. I'll go ahead and remove Line 5, the duplicate 4, save and commit my changes. We can use git diff to compare the numbers.txt file in main with the numbers.txt file in the colors2.0 branch. We'll run git diff colors2.0 main, then the file name numbers.txt. Now we can see the file in main has the extra duplicate 4. As a reminder, you can download the slides in the exercise files for this course, as well as a readme with the commands from the demos. Let's review some of the commands you used in the demo. You can use this page as a reference to remember some of the common diff commands. Git diff will show you what has changed that is not staged for a commit. If you add the ‑‑cached flag, then you'll see what is staged for commit that is different from the last commit. In other words, what would be committed if you ran git commit. Git diff HEAD will show you what has changed since the last commit, or in other words, what would be committed if you ran git commit ‑a. You also used the ‑w flag to ignore whitespace differences. These commands can help you decide what to add to a commit and to double‑check things before making a commit. Next, let's look at some commands that take a commit SHA as a parameter. If you pass in a single commit SHA, you'll see the difference between the specified commit and the current working directory. If you add the ‑‑cached flag, you'll see the difference between the specified commit and what is staged. You can also pass in two different commit SHAs to see the difference between those two specific commits. These commands can help you compare specific items that have already been committed. These next commands take branch names as parameters. I'll use feature and main as example branch names. To see the difference between the tips of two branches, simply list the two branch names. You can also see what has changed in main since the feature branch was created from main by adding ... in between the two branch names. To view the difference of a specific file between two branches, add the file name after listing both branches. These commands can be helpful when preparing to merge a branch or to see what has happened since you created a branch.

### Resolving Merge Conflicts

If you're working alone, merging is pretty straightforward. However, when you're working on a team, you're more likely to encounter conflicts when you merge. A conflict is simply something that has changed in both branches and Git needs you to tell it how to combine the two changes before it can finish the merge. For a conflicted file, Git will put all of the content from both versions into a single file with file markers to indicate where that content comes from. You'll then go in and edit the file into a final version that combines the changes correctly. When you're finished resolving the conflicts, you'll add and commit the changes, then Git will finish the merge by making a merge commit. A merge commit is the commit that combines the content from what is being merged and will have two parent commits, one from each side of the merge. When you open a file with conflicts, you'll see one or more sections with the file markers less than, equal, and greater than. In this example, information under the less‑than signs is the information from the file in your current branch. The equals signs separate the content from the two different versions of the file, and the information above the greater‑than signs is the content from the file contained in the branch you are merging in. For this example, it's coming from the main branch. To resolve the conflicts, choose what each line should be. When you're finished, save and commit the file, and don't forget to remove the file markers. Like diff, most IDEs have built‑in support for merging, and files are graphically displayed with colors and windows to interact with. These tools can make some merges even easier. We'll stick to the command line for these examples to understand what is going on behind the scenes, but I encourage you to explore some graphical merging tools available for your environment to see what works best for you. Let's walk through a merge that has conflicts on the command line to get a feel for how the process works. We'll resolve the conflicts and complete the merge. Let's continue in our directory with colors, numbers, and letters. I'm on the main branch with no changes, and if we take a look at colors.txt, there are 16 colors. Let's take a look at the log to see the last commit in main. Remember git log ‑‑oneline. We can see that the last commit in main was changing all of the colors to one word only. Let's switch to the colors2.0 branch. If we take a look at colors.txt, notice that there are 18 colors, and some of the colors are different. We can check the log to see the last commit in the colors2.0 branch. Here we can see it was removing a duplicate line. If we run git status, we can see that the working tree is also clean. Let's go ahead and merge main into the colors2.0 branch by running git merge main. Notice the message that there is a conflict in colors.txt. We'll need to fix the conflicts then commit the result. When we open colors.txt, we'll see that Git has updated the contents of the file with file markers indicating what is in the colors2.0 branch and what's in main. In this example, there was only one section that Git was unable to resolve. We'll go ahead and pick which one we want for each line. I'll keep nectarine and delete orange. I'll remove yellow, remove light green, and remove blue. For this merge, I've kept three items from my current branch and one from main. Before we finish, be sure to remove the file markers. Save the file. Then suppose at this point you got interrupted and went away from your desk. When you sit back down to do some work, depending on your terminal environment, you'll see a hint that you're in the middle of merging. Git Bash adds MERGING to the name of the branch at the end of the command prompt. You can also run git status and see that there are unmerged paths. Let's stage colors.txt and see how the status message changes. Notice now that it says all conflicts are fixed, but we're still in the merging state. We need to run git commit to conclude the merge. We'll go ahead and run that, and Git will bring up the message for the merge commit. It's populated with a default message, Merge branch 'main' into colors2.0, and I don't really need to add any additional information, so I'll go ahead and leave that. The rest of this file contains some additional information and instructions. Everything with a hashtag in front of it will be ignored on the commit message. Save and exit that file, then Git will automatically conclude the merge. Notice that my command prompt no longer says MERGING, and if I run git status, we'll see that we're in a clean working tree. Let's take a look at the log to see what happened. Notice the commit from main, only one word colors allowed; the commit from colors2.0 removing duplicate line; and finally, the merge commit where main was merged into colors2.0. Now that we've merged main into the colors2.0 branch, let's compare colors.txt in the 2.0 branch with main. (Typing) Notice that the colors.txt file in main is not changed because colors2.0 was the target branch of the merge, and main was the source. In order to change main, we need to merge the colors2.0 branch back into main. To do that, we'll switch to the main branch, double‑check that nothing has happened with git status, then perform the merge, git merge and the name of the branch we want to merge into main, which at this point is colors2.0. Notice that it was a fast‑forward merge because we had already resolved the conflicts in the version of the file In the colors2.0 branch. If we take a look at the log again, we can see that there is no additional merge commit because it was simply a fast‑forward merge. Main is now pointing at the same commit as the colors2.0 branch. As you do work in a branch that you know you need to merge back into another branch, pulling updates from that original branch more frequently helps your merges be more simple and your conflicts easier to resolve.

### Aborting a Merge

There may be times when you're in the middle of a merge, diffing files and resolving conflicts, that you just need to bail. Maybe you found a new bug to fix or you forgot to add a requirement. Sometimes you may need to get a team member to help you resolve some conflicts. You can run git merge with the ‑‑abort flag to stop the merge and return the directory to the previous state it was in before the merge started. As you work through merges, remember, just take it one step at a time. Look at the current conflict, decide what to do, then resolve the conflict, and you'll have the file merged in no time. Beware of the evil merge. An evil merge is one that introduces new content as part of the merge. A merge commit should only combine existing content. New content introduced in a merge commit can be confusing, as there's no context about where it came from or why it's there. If you realize something is missing during the merge, you can abort the merge, make the change in a new commit, then restart the merge. Let's go to the command line and start a merge, abort the merge to commit some new code, then restart and complete the merge. We'll pick up where we left off in main with our colors, numbers, and letters files. Let's make some changes to the colors file. We'll delete a couple of lines, add something that's not correct, move something around, and add another change. We'll commit the changes, (Typing) and check our status. Now let's make some conflicting changes in the colors2.0 branch. We'll switch to colors2.0, then open the colors.txt file, and let's make a few changes in similar places. We added something to the first line in the main branch, so let's do that same thing here, only with a different value. We'll delete something else, add something on a similar line, then remove a couple of other lines. We'll add and commit. (Typing) Now let's go ahead and merge main into colors2.0. Not surprisingly, we get a conflict message. Let's take a look at the merge conflicts. Notice now that we have more than one section, that first line, and more toward the middle. Suppose while you're going through this merge, you notice that orange is not only duplicated, but the second one is misspelled. Now, if you were to make that change now as part of the merge, you would be introducing new information into this commit. The correction didn't happen on either branch, so that is what would be nicknamed an evil commit, in that it's kind of hard to track down what happened or why. The good news is we can abort this merge, fix it, then come back to it. Run git merge ‑‑abort, and this will get us out of the merging state and restore the colors.txt file to the state that it was before we started the merge. We can see that the misspelling didn't happen in the colors2.0 branch, it's in main. We'll switch back to main. Now that we're in main, we'll fix the mistake, save the file, and commit the change. (Typing) If we check the log, we can see both commits that we've just done in main, modifying colors in main followed by fixing color mistake. Let's go back to the colors2.0 branch and start up the merge again. (Typing) We get the conflict message, and see the file markers in the colors.txt file. Notice now that the second orange is now fixed. We'll go ahead and resolve these changes. I want to keep all three of these colors, but silver should be after gold and iron. Note that things like reordering are okay because I'm not adding any new information, I'm simply combining the information in the way that is now correct. Remove the file markers. For this conflict, I'll just keep the one from main, double‑check that everything looks correct, then save the file, run a git add and git commit. I'll accept the default message, and the merge is complete. We'll review the log and see both commits from main, as well as the new merge commit that just happened. As you've seen from this example, aborting a commit isn't a big deal. It's actually kind of nice to be able to reset all of your files to the state that they were in before the merge happened, particularly if you get into some really difficult conflicts that need to be resolved. You can abort the merge, take a step back, gather some additional information or even make some corrections on the specific branches that you're trying to merge. Now that you've completed this module, hopefully merging feels more like a tool than a monster. You've seen how the fast‑forward merge works for simple cases and several different ways to track changes using git diff and git show. You've resolved merge conflicts and even aborted a merge to properly add new content, then restart and complete the merge. Join me in the next module to learn more about remotes, pull requests, ignore files, and other strategies for using branches with your team, so that you can spend less time resolving conflicts and more time getting your work done.

## Using Git Branches with Your Team

### Setting up Remotes

Hi. This is Craig Golightly, and welcome to Using Git Branches with Your Team. Let's get into the details of how to use branches to more easily work with your team. We're going to cover remotes, pull, and push, as well as pull requests and how those can help your workflow. We'll look at some of the built‑in features of Git that help avoid conflicts like the gitignore file. Then we'll discuss some best practices for team members as they use Git. You'll see what can go wrong and how you can avoid these traps, things like whitespace errors and breaking the build. A key piece of using Git with a team is the use of remote repositories or remotes. A remote is simply a location where a version of your project is hosted. Many teams use services like GitHub, Bitbucket, and GitLab to host their remote repositories, but a remote could be pretty much anywhere. A remote is typically the version of your project that everyone will use as a reference for overall team progress so things like tracking team milestones, code coverage for testing, and code reviews tend to revolve around remotes. We'll use GitHub for the demos in this course. If you'd like to follow along, you can use the fork feature which creates a copy of a repository in your own account. This allows you to use your own forked copy of the project as a remote so you can experiment and make changes. We'll use the fork feature to start off the demos for this module so you're not creating everything from scratch. Let's go to an existing GitHub project and get the information needed to clone the repository to a local machine. Once we have the repository cloned, we'll verify the remote setup. If you'd like to follow along the demos for this module, I've created a starting repository that you can fork. Go to github.com/seethatgodemo/widget, then click on the Fork button. GitHub will ask where you want to fork this repo and I'll select my personal seethatgo account. Now I have my own copy of the widget project in my own account. When you fork the project to your account, you'll be able to do the same things that I'm doing in the demo. The first thing we need to do is clone this repository down to our local machine. Under the Code button, you'll have different options to clone the repo. I'm going to go ahead and use the HTTPS option. Click the clipboard next to the URL, then go to the directory where you'd like to clone the repository. I'm going to clone it in this directory called remotes. Right now, there is nothing in it. Run git clone, then the URL you copied. If I list the files now, I see a widget directory and the files that are in that remote repo. If you run git status, you'll see that you're on the main branch and if you run git remote ‑v, you'll see the URL for the account that you cloned the widget project into. For me, it's seethatgo. You have successfully cloned the remote project down to your local workstation. Let's review some of the common commands used when working with remotes. A remote will have a URL that you can use to create a local copy. You can run git clone followed by the remote URL and Git will automatically pull down the code and set up a local main branch to track the remote main branch. By default, Git names the server you cloned from origin. If you want to choose the name for a remote, you can run git remote add followed by the name for the server and the remote URL. To see the list of remotes and URLs you have set up, you can run git remote ‑v.

### Using Remotes with Code

The nice thing about Git is that every developer also has their own copy of the repository where they're working. This gives them the control to choose when to integrate changes from everyone else and when to share what they have been working on with the team. Sharing code to and from remote is accomplished with a few simple commands. To receive changes from remote, you can use git fetch or git pull. Fetch will download any information from remote that isn't already in your local copy. You can then evaluate it and decide what you want to do. Pull is a combination of fetch and merge, so it will download the information and initiate a merge on your local branch. To share changes from your copy of the repo with remote, you'll run git push. Note that Git will not allow you to push if you have unresolved conflicts with remote. You must resolve any conflicts first, then you can push. Let's use the repo you cloned in the last demo to pull a change made to the remote repo down to your local repo. You'll then make a change locally, then push that change to remote so it's available to other users of the repo. Finally, you'll see what happens when you try to push changes to a remote that has changed while you were working and how to resolve that scenario. Navigate to the page in your account where you cloned the widget project. Select Add file, and we're going to add a config.txt file. This simulates a change made to the remote repository that is not in your local copy. Go ahead and commit the file directly to the main branch, and now we see it listed in the main page. If you run git status and don't get any messages, you may not be up to date with remote. Run git remote update so that Git is aware of changes that have happened on remote. Now when you run git status, notice that it gives a message that your branch is behind origin/main by 1 commit, and you can run git pull to update your local branch. On my local branch, I'm missing the config.txt that was added remotely. When I run git pull, note that it's added config.txt by performing a fast forward merge. Let's make some changes to the config.txt file. We'll add some additional configuration, save the file, and commit. Now when we run git status, Git lets us know that now your local branch is ahead of origin/main by one commit and that we can run git push to publish those local commits to remote. Let's push our commits to remote by running git push. Now when we run git status, we're all up to date. If you refresh your project web page, note the change to config.txt along with the commit message, added some additional config files. Let's simulate another change on remote that's happening at the same time as us working locally. We'll add a folder for feature3. In the feature3 folder, add a feature3.txt file with an implementation. Commit the file, and now you can see the changes on remote. Locally, let's go ahead and make some changes to the config.txt file. Save and commit. Let's attempt to push our changes to remote using git push. Notice the message from Git informing you that something else has been pushed to remote and that you may need to run git pull before you can push. By running git pull, remember that git pull is a combination of fetch and merge, so we're going to do the merge, and now if we list the files, we can see the new feature3 folder with the feature3.txt that was added remotely. If we check our status, we can see that we're ahead by two commits. If we take a look at the log by running git log ‑‑oneline, we can see the last two commits of adding additional config values, then the merge commit that happened when we pulled down the changes from remote. If we refresh the project page, showing the status of our remote repository, note that the new changes aren't there because we haven't pushed them, all we did was pull changes from remote to local. We still have a change local that we need to push to remote. Run git push. Now if we go refresh remote, we can see the additional config values that have been pushed.

### Using Remotes with Branches

You can also utilize remotes for your branches. Suppose you've done a lot of work on a local branch, but it's not ready to merge just yet. You'd like to have a copy of it on remote so you can continue the work on another machine. You can push your branch to remote. This will create a copy of your branch in the remote repository. You can then pull that branch down on a different machine and continue your work. When you do this, be aware that now anyone with access to the repository can pull it. Some teams have conventions for naming developer branches like certain prefixes or postfixes to clearly distinguish what is and is not considered part of the code base. Generally, people know which branches everyone on the team is working on, so this isn't usually a problem. Code reviews and troubleshooting are some other scenarios where pushing a branch to remote is useful. We'll cover this aspect more when we talk about pull requests. Let's walk through pushing a branch to remote, then accessing it from a different location. You'll create a local branch with some changes, then push it to remote. From a different location, you'll then list the branches on remote, then fetch and check out the branch you pushed. To simulate pulling down a branch from a different location after it was pushed, first clone the current remote widget project into a different directory than where you'll execute these first steps. I've created a folder under my remotes called otherMachine to simulate a different machine. Right now it's empty, so I'll go, and I'll grab the clone URL from my remote project. Recall that on GitHub, it's under the Code button. Copy the HTTPS URL, then run git clone, followed by the URL. If I go into the widget folder, I can see a copy of the current state of the project. I'm going to switch back over to my first machine now, and let's create a branch to work on, feature4. Remember, git checkout ‑b and the name of the branch, feature4. Now that I'm in the feature4 branch, I'll create a folder, go into that folder, and create a feature4.txt file. We'll simulate starting some work on this feature, save and commit the file. And now let's check the branches that are on remote by running git ls‑remote. Notice that there is no feature4 branch yet. We need to push the feature4 branch to remote so that we can fetch it from a different location and continue working on it. Run git push ‑u origin feature4. Now if we list the remotes again, notice that the feature4 branch is listed. Now let's switch to the directory that simulates being on a different machine. Now I'm in my otherMachine directory. If I run git branch, notice that I only have a main branch on this machine. Let's verify the remote for this repository by running git remote ‑v. I can see that it's pointed to the correct URL that I'm expecting for my copy of this project. If I list the remote branches from here, I can see the feature4 branch listed. To fetch the feature4 branch from remote to local, run git fetch origin feature4. If you run git branch at this point, you may be wondering what's going on. I thought I just fetched the new branch. If you add the ‑a flag to git branch, you'll see that now it is listing both remote tracking and local branches. We need to set up a branch in the local repository to track the remote feature4 branch. We'll use the git checkout command for this and pass in the ‑‑track flag with origin/feature4. Notice that we've changed to the feature4 local branch. Now when we run git branch, we can see the feature4 branch. And if we list the files, we can see the feature4 folder with feature4.txt. Let's review some of the commands from the demo. To push a local branch to remote, use git push ‑u origin, then the name of the branch. Note the ‑u flag sets origin as the upstream branch. To list the branches on remote, run git ls‑remote. To list remotes and their URLs, run git remote ‑v. To fetch a branch from remote to local, run git fetch origin, then the name of the branch. To list both remote tracking and local branches, add the ‑a flag to git branch. Finally, to set up a local branch to track the remote branch, run get checkout ‑‑track origin/, the name of the branch.

### Using Pull Requests

When you create a branch, do some work in it, then are ready to merge that branch back to main, you can utilize a pull request to facilitate the review and testing process. Sometimes, you may not even have access to merge directly into main, so a pull request would allow the maintainers of a project to review, then merge your code. Let's review the basic workflow for a pull request. First, push your branch to remote. Then, when it's ready, open a pull request. The repository that hosts your remote generally has functionality for pull requests in their web console. You can do things like see the commits, add reviewers, and discuss changes with your team. During the review process, you may have some fixes to make before the branch can be merged. No problem, just make the changes in your local branch, then push the commit to remote and it will automatically get added to the pull request. Remember that you should not rebase after you've pushed your branch to remote. If you're not sure what rebasing is, don't worry, we'll cover that a little later in this course. Once your pull request has been approved, it can be merged into main, and the branch deleted. Let's use a feature branch to implement some new code. You'll open a pull request for feedback and revisions, then merge and delete the feature branch once you're finished. Let's start back in the root directory of the widgit project on the main branch. Currently I have three features. Let's create a branch called utility to implement a new feature, git checkout ‑b utility. Let's create a util folder with three files, (Typing) config.txt, feature.txt, and utility.txt. I've included the sample text for these files in the README for the demos. Add the files, double‑check what we're adding, and commit. Push the utility branch to remote by running git push ‑u origin utility. (Typing) On the widgit project page for the remote, notice how we just push the utility branch. Let's create a pull request for that. Because this is a forked project, it defaults the pull request to the forked project. For this demo, we don't want to do that, we want to just keep it in our own copy of the repository. So change the base repository from seethatgodemo/widget to just seethatgo/widget. Your project will be called whatever your repository is where you forked the demo project. I'll add a better title and description to the pull request, add another user as a reviewer, then create the pull request. When the other user logs in, they'll see the open pull request. They can view a summary on the initial page, then look into each of the files changed. Reviewers can add comments to one or more lines of code. I'll add another comment about some expansion for this implementation. The reviewer can check if they viewed each file, then finish their review. A review can either be comments only, approval or a request for changes. I'll select Request changes. The reviewer can add additional notes and instructions, then submit the review. When the author goes back to the pull request page, they'll see the additional comments. Back on your local machine, go ahead and make the fixes. Config.txt needed a new value and feature 2 needed some additional capabilities. We'll check the status of our changes, and see that we've modified the two files. Let's add, commit, and push the changes to remote. (Typing) Remember git push origin utility to push the changes to remote. If you go to Commits, you can see the latest commit that was made, Modifying code based on review. We can see the changes that were made, and even respond to the conversation. If you put the @ sign in front of a user name, it will notify that user directly about the comment. I'll make this review simply a Comment and submit the review. The reviewer will also get an email notifying them about the comments. The original reviewer can go back and review the changes that you pushed to make the corrections. There's the 567 and the additional implementation. At this point, the reviewer can now approve the changes and submit their review. The original submitter will see the approval and can now merge the pull request. At this point, you can delete the branch, because all changes have been merged into main. This will delete the remote branch only. Back on your local machine, switch to main, then run git pull. Note that my command prompt tried to stay in the util folder, which doesn't exist in main because we haven't got the changes. We'll fix that, and notice that main shows up, and try the pull again. We can see the three files that were added, and if we run git branch, we still have our local copy of the utility branch. We can delete that with git branch ‑d utility. Running git branch again confirms that we've deleted our local utility branch.

### Ignoring Files

The .gitignore, or ignore file for short, is a way to tell Git what files you want it to well ignore. All files in Git are either tracked, you've added or committed them, untracked, Git is going to tell you they are untracked with every status, or ignored, Git won't say anything about them or do anything with them. You'll typically want to ignore files that are generated by your code base, things like compiled code, log files, build directories, caches, hidden system files, and any individual configuration files like preferences for your IDE. Ignoring a file ensures it won't be added to a branch or pushed to a remote, and since they aren't added to a branch, they won't appear or disappear whenever you switch branches. The gitignore file uses a regular expression like syntax to specify patterns of files and directories to ignore. These first few metacharacters function as you might expect. Asterisk matches anything, except a slash, question mark matches one character, except a slash, exclamation point is a negator. The slash is always used as a directory separator and the square brackets can be used to specify ranges. You can add comments to the ignore file with a hashtag. The double asterisk will match a directory anywhere in the repository. In this example, any bin directory anywhere in the repository will be ignored. Using a single asterisk for file names or extensions will also match any file in the repository. This would ignore any zip files anywhere in the repository. Any expressions with directory separators will be evaluated relative to the directory where the .gitignore file resides. It's a good idea to have an ignore file at the root of your project to ensure that everyone is ignoring the same things. It can be a real pain when someone checks in a bunch of files that are supposed to be ignored. If that happens, you can use git rm ‑‑cached then the file name to delete the file from your repository or git rm filename to delete it from both your repository and your local file system. If you have any additional ignore patterns that will help you, but not the project, you can add those in the .git/info/exclude file. That file will apply those patterns to your local repository only. Let's go to the command line and see the gitignore file in action. You'll create a file that is ignored by the project's .gitignore file to see how Git ignores it, then see how to customize ignore rules that will only affect your local workspace. Finally, you'll see where you can find several example .gitignore files for various programming languages. Let's take a look at the gitignore file for the widget project. This one is very simple. It's ignoring all files that end in a .log extension. Let's create a file called test.log to see what happens. We have a clean working directory. I'll create the test.log file, save it, note that it is in the directory, but when I run git status, it tells me that there is still nothing to commit. The ignore file is doing its job and not cluttering up our messages with things that we don't want to add to Git and we don't want Git to tell us about. Let's now add a text file. We can see the file and now on git status, Git identifies it as an untracked file. Suppose you wanted to add some text files to your machine and not have them tracked by Git, but you don't want that setting to affect other members of the project. From the root of your repository, go to the .git directory, then info, and exclude. This uses the same syntax as the ignore file. I can go ahead and save star.txt to ignore text files, go back to my directory, (Typing) and now git status ignores the text file. If you go to github.com/github/gitignore, you'll see several examples of ignore files for various languages and tools. For example, the Python gitignore contains different things that you typically would want to ignore in a Python project. These can be a good starting point for some of your projects.

### Setting Team Conventions

There's a story about a village that was near a cliff. People visiting would get too close to the edge of the cliff and fall off. The people in the village wanted to do something to help, but they were divided. Half of them wanted to buy an ambulance and put it in the valley so that when people fell down, they would have a way to try to help them or at least get them to the hospital. The other half wanted to put a fence on top of the cliff to prevent people from walking off the edge. In Git, the cliff represents the potential conflicts when working with a team. The ambulance represents the various merging tools available for you to work through difficult conflicts, and the fence represents team conventions and best practices that help you avoid falling off the cliff. Let's take a look at some ways to reduce the ambulance in the valley situations on your team. First, have a common .gitignore file for each project. This will help reduce confusion and prevent unwanted files from being added to the repository. Next, use a README file for important information about the project. We won't cover markdown in this course, but it is much easier to read than a plain text file. Stay connected with what's going on in the codebase. In other words, don't go in a cave for a month. Frequent smaller commits are going to be easier to manage than one giant commit once a month. Finally, coordination. Sometimes your project may need a large refactor. If a commit is going to affect a large percentage of the project files, coordinating with the team members to merge to main before the change can help reduce confusion compared to just making the large change while dozens of people have branches in flight in the same project. Many teams have automated tests that run before or after a new code is checked into the repository. Typically, these will be unit tests to ensure that no existing functionality is broken by a new commit. If everyone is working on main, a commit will usually kick off a timer where anything in the next 5 minutes or so gets built, then the tests run against the new build. If teams are using branches, typically main is merged into the branch, the tests run, then if successful, the branch gets merged to main. Utilizing branches for changes makes testing easier since the change is isolated in the branch and can be verified before the merge to main. Once the tests are run, an email usually goes out to the team with the results, pass or fail. The two main types of errors that can happen are syntax errors and unit test failures. We'll talk about some ways to avoid those here in just a second. If you do break the build, take ownership. A quick message with the words on it lets the team know you are aware and looking into it. Get help from others if you're not sure why it broke and report back with a message to the team when it's fixed. When merging branches that you intend to push to remote and share with your team, make sure you have the latest changes from any remote before you start the merge. To avoid syntax errors, build the code on your machine before you push or merge to main. To avoid unit test errors, run the unit tests on your code before you merge and fix any broken tests or code that unintentionally broke a test. That sounds pretty obvious, but I've been places where some individuals were constantly breaking the build because they did not do those two simple things before merging code. That caused problems and delays for dozens of other teammates.

### Creating Code That Can Merge

There are a few things you can do to help your team when you make commits. First, commit related changes. Group features together in a commit that stand on their own, not half of two different, unrelated things. This makes merging easier. If there is an overlap on a feature, or if a particular feature needs to be moved into another branch, your commit won't have part of an unrelated feature going along with it. Next, use a good subject line for your commit message. Think of it like the subject in an email. Could it answer the question of what this commit will do if applied? Try to keep it around 50 characters or less so team members can get a quick, concise insight when browsing through a list of commits. Finally, add a body to your commit when appropriate. Not all commits will be quick, simple, one‑line subjects. Sometimes, you need to provide more context around why the change was made to the code. This could really help someone reading through the project a year or more down the line who is trying to reconstruct something and understand a series of changes. People can use the diff to see what changed, use the commit message body to tell them why it changed. A feature flag can be as simple as an if statement that surrounds new code. If the flag is true, then the new code will run. Feature flags are a great way to isolate your changes. They make the new code easy to identify, since it's surrounded by an if block. The existing code should be placed in an else block, or the equivalent default for whatever language you're using. If the feature flag is not true, then the existing code should run. Using feature flags allows you to submit buildable code with each commit, while the new code is still in progress. It also helps you maintain frequent commits while working on a larger feature. Many projects keep feature flags in the code for the initial release of a feature. If there's a problem, rollback is easy, just change the flag to false, instead of completely having to redeploy a different version of the code. You can also use the feature flags for a gradual rollout where only a certain percentage of the users get the new feature right away, then that percentage is gradually increased until it's at 100%. Commented out code can be a bit of a mystery in a code base. You may come across a block like this, where it's not apparent why this wasn't just deleted. If it's not going to run, why is it still here? As long as the code has been committed at least once, then Git will remember it. Don't let your code base become a junkyard of items you might need later. Commented out code costs the time of every person who comes across it and wonders why it's there. Let Git do its job of remembering your history, and clean up your code base from commented out code. So what's the big deal with whitespace? Nobody sees it, right? Wrong. Git sees everything, including whitespace. While you can tell Git to ignore whitespace, if different team members are running different code formatters on the same piece of code, it creates a lot of extra noise and diffs, and can drown out the real changes in a sea of red lines changed. I once came back from vacation to see an email summary from Git that over 1000 lines of code had been changed by a new team member. I thought, what in the world have they been doing? It turns out, they were just running format on every file they touched before they checked it in, but they hadn't bothered to notice that the default format settings in their IDE did not match the format of the code in the repository. So, pick a formatting standard and stick to it. Some teams even have automated processes to ensure code is formatted according to team standards before it is committed to the repository. The other troublesome whitespace you'll encounter are line endings. Line endings can be different for different operating systems. If there are different line endings in the same code base, this can cause problems for some programs. If you've ever encountered a weird bug that turned out to be related to line endings after lots and lots of digging, you know what I mean. If not, consider yourself lucky, and warned. A line ending standard should be defined by the team and enforced to ensure everyone is using consistent settings for any code they're adding to the Git repository. Let's recap some of the ways that branches can enable your team to collaborate on code. You've seen how to use and manage remotes and how the pull request is a great tool for making code reviews easier and coordinating merges. You've also covered how to avoid breaking the build by checking syntax and tests locally before you merge, and how frequent branch updates and smaller merges can be easier for your team to manage. Finally, you learned some important ways to avoid conflicts by using a .gitignore file, feature flags, and ensuring consistent whitespace settings for your code. Join me in the next module to learn some advanced merging methods, including rebase, squash, and cherry‑pick.

## Advanced Merging Methods

### Rearranging with Rebase

Hi, this is Craig Golightly, and welcome to Advanced Merging Methods. Let's look at the rebase feature in Git to see when you might use this to squash several commits into a single commit or replay commits from another branch into the branch you are currently working in. We'll also look at the cherry‑pick feature for times when you may need a particular commit from another branch without merging the entire branch or you have a bug fix to apply to several branches. Sometimes you have to make a mess to create something. If you ever look at your kitchen after preparing an amazing meal, there are probably several pots, pans, dishes, and other things to clean up. The same thing can happen when writing code. You might forget things along the way, create checkpoint commits, and have several attempts that don't work. Git rebase can be used to clean up your local history and focus on the end result. The goal is to increase the accuracy and clarity of what the code is doing and why changes were made so that someone else reading the history or trying to perform a merge has the information they need. You could think of it as the difference between reading through a rough draft versus the final copy of an article. Keep in mind that rebase is an advanced feature. It is by no means a mandatory part of your daily use of Git. When working in a distributed environment with other team members, rebase can cause problems, so let's look at when you should not use rebase. Do not use rebase on a public branch. Let me say that again. If you rebase on a public branch, you're almost always going to cause confusion, and you may even cause others to lose their work, so do not use rebase on a public branch. There's a fair amount of debate around the use of rebase. Some teams may choose not to use rebase anywhere, so be sure to check your team guidelines. In this course, we're going to touch on rebasing branches with a couple of examples. There's another full course on Rewriting Git History, so check that out for a deep dive into all aspects of rebase. There's a couple of scenarios where you might want to use rebase with branching. The first is cleaning up your local history before sharing a branch. Maybe you want to do this for personal reasons, or maybe the project you were trying to share your code with wants you to clean up the history. Either way, you can use rebase to accomplish this. The other is pulling changes into your branch from main without performing a merge.

### Squashing Multiple Commits

Suppose you are working on a branch and you make three related commits. The actual end result you wanted to build is finished up in the last commit and you want to have all three commits appear as one. You can use Rebase to squash or combine the second and third commits into the first so it appears as though you did all of the work in one commit. What Git actually does is create a new commit, then copy the changes from each commit into that new commit. You can then choose the commit message for that new commit and it will appear as one commit in your history. Let's use Rebase to squash several commits into a single commit. I'm in a new directory so let's initialize it as a git repo. We'll add a file to main, then add and commit the file. (Typing) Next, let's create a ticket1 branch, git checkout ‑b ticket1. We'll create a file, then perform several changes and commits to that file. (Typing) Add and commit this first checkpoint. (Typing) Let's go ahead and do some more work on the ticket, check that in, (Typing) then finally, finish up work on the ticket. We'll commit the end product. (Typing) And now let's take a look at our history using git log ‑‑oneline. Right now, this repo has three commits, the initial file in main, then the three commits we just made while working on ticket1. Suppose you were getting ready to merge the ticket1 branch back to main, but you didn't want to have all three commits from working on ticket1. Let's use the git merge‑base command to determine where we should start the Rebase from to squash all of our work from ticket1 into a single commit, run git merge‑base ticket1 main. This returns the full commit SHA. Notice how the first few characters match the initial commit first file in main. We can start the interactive Rebase with git rebase ‑I, then the first few characters of the commit SHA. Git will present this file with each commit that is part of the rebase. We're going to squash these three commits into one commit. We'll do this by changing pick to squash for the second and third commits. Note also in the comments that Git provides several different options for the rebase command. We're done here so we'll Save and Exit. Next, Git will show the commit message from each commit involved in the rebase. I'm going to just remove the comments from the first two and leave the finished ticket1. If you needed to also edit this, you could add additional information. Once you're satisfied with the commit message, save and close the file. Note the message about the successful rebase. If we take a look at the log now, note that there is only one commit, instead of the previous three commits. Let's review some of the commands we used to squash several commits into one. Recall that git log ‑oneline shows you your branch history. Git merge‑base ticket1 main shows the original base of the ticket1 branch created from main. Now for the rebase command itself, the ‑i flag is for an interactive rebase and the commit SHA tells git where to start the rebase. Git opens a file with a list of commits for you to work with. There are several different options of what you can do at this point in the rebase. To squash multiple commits into a single commit, type squash in front of the commits you want to combine, instead of pick. Once you save and close this file, you'll be presented with the commit messages from the three commits involved in the squash. You can choose to keep one or more messages or combine them into something new. When you save and close this file, then Git will complete the rebase.

### Rebasing from Main

Suppose you're in a branch off of main and you make a couple of commits. If you run git rebase main, then any changes that happened in the main branch will be pulled over to your branch, then your commits will be replayed on top of the last commit from main. This can help sequence your changes with main and create a more linear history when you merge back to main. Note that rebase can't just move your old commits. It will actually create a copy of them so that it can change the parent. The original commits are eventually garbage collected by Git, so keep in mind that rebase creates a copy of your commits so that it can change the parent. Let's look at rebasing changes from main into a feature branch. You'll see how Git sequences the commits from main before commits in the feature branch. We'll then compare rebase with merge. I'm starting in an empty directory, so let's initialize it and add some content. (Typing) Add and commit the file. (Typing) And we'll create one more. (Typing) We'll add and commit the other file, and now let's create a ticket2 branch to make some changes. (Typing) We'll add two files to this, and commit each one, ticket2.txt (Typing) and ticket2helper.txt. (Typing) Note the current state of our repository. We have two files that were created in main and two files created in the ticket2 branch. If we look at our log, we can see the two commits in main and the two commits in the ticket2 branch. Let's switch back to main and add some more content. (Typing) Add and commit main3. We're now at a point where we could go to the ticket2 branch and rebase to get the changes from main, or we could perform a merge. I'm going to make a copy of this directory so after we rebase we can go back and merge and compare the two in terms of what shows up in the history of Git. I'll create a new r2 copy directory, then copy all of the contents over there. (Typing) If we go to that directory, we can see our files in main, our ticket2 branch, and the files unique to the ticket2 branch. Let's go back to our original folder. To rebase main into the ticket2 branch, we need to first switch back to the ticket2 branch. Currently this branch does not have the main3 file. Run git rebase main. Note the message that it successfully rebased and updated our branch. Now we can see the main3 file, and if we look at the log, we can see that the commit for the new file in main has been placed before the two commits in the ticket2 branch. For even more detail, you can run git reflog. This is read from bottom to top, and provides the history of what has actually happened. The initial two commits in main creating the ticket2 branch, the two commits on ticket2 going back to main, committing the third file in main, going back to the ticket2 branch, then starting the rebase, then the two commits in the ticket2 branch. By rebasing main into the ticket2 branch, the commits from main are grouped together before the commits in the ticket2 branch. Now let's go to the copy that we made. We'll check our status. We're in the ticket2 branch, and this is the same state as our other repo before we did the rebase. Let's go ahead and merge main into the ticket2 branch by running git merge main. We'll save the commit message, exit the file, then see that the merge was made by the recursive strategy. We can now see the new file from main, and if we look at the log, notice the difference from rebase. We have the two commits from main, the two commits in ticket2, the new file in main, and the merge commit when we merged main into ticket2. We can compare this to the rebase, (Typing) and see the difference in sequence and that there is no merge commit when you do a rebase. There's really no right or wrong, it's just two different ways of keeping track of your history and organizing your commits. Let's touch on another tool you may need to use if someone upstream from the branch you are working on decides to rebase. Suppose you made a branch from some upstream branch at a certain point, then continued work on that branch. Meanwhile, someone rebased the upstream branch, which they shouldn't do, but they did, squashing several commits, including the commit where you started your branch. On top of all that, they added something new. If you ran git pull at this point, you would have several conflicts that would be difficult to resolve. You can instead run git pull ‑‑rebase, which will look at the reflog for each commit to find the common ancestor, even if it has been squashed, then sequence the commits in your branch. Hopefully you won't have to use this very often, but if you do encounter this scenario, remember to use git pull ‑‑rebase.

### Using Cherry Pick

Imagine that the cherries on this branch represent several commits in a Git branch. What do you do if you only want to bring one of those commits into another branch? You cherry pick it. Git cherry‑pick appends any commit to the current working head of the branch you are in, effectively making a copy of the commit in a different branch. In reality, cherry‑pick does not move the commit, it creates a copy of the commit, but with a new parent in the branch you're cherry picking into. The original commit stays the same and is not removed from the branch the way a cherry would be if you picked it. A common case for cherry picking is a bug fix that applies to multiple versions of a product. If each product is on its own branch, you can make the commit to fix the bug in one branch, then cherry pick that commit into the other branches to apply the fix. You could also use cherry‑pick to capture a few commits from a branch that isn't going to be merged, but some of the commits may still be useful. There may be times when parallel development is happening in different branches and you need something from another branch to work on your task, but the branches aren't ready to merge yet. You can cherry pick the commit you need from the other branch, then continue your work. Keep in mind that cherry‑pick does create a duplicate commit, so if the branch you picked from is merged into the branch where you placed the cherry‑pick, it could cause confusion if the person merging is not aware of the cherry‑pick. Like rebase, cherry‑pick is an advanced feature and it should not replace regular merging. Let's look at an example where you're working on a feature branch and need to figure out a couple of different variables in order to move on. You create a branch for x and try a few things before arriving at a solution. You do the same thing for y. You don't really need to merge everything back into your feature branch, you just need the final commit for x and y. You can cherry pick the commit from x and from y into the feature branch. This will create a copy of those commits in the feature branch. You can then move on with development of the feature. Here's a code base where the current version in development is 3.0. There's a branch for version 1.4 with some changes for that version, and another branch for version 2.1. Both of the previous versions are still in production and have active customers. While developing on the 3.0 branch, a critical security bug was discovered and fixed. This same vulnerability was determined to exist in both the 2.1 version and the 1.4 version. You can't merge the 3.0 branch into the previous versions because it would break functionality. How can you apply the fix into the other versions? You can cherry pick the commit from the 3.0 branch into the other two branches. Git will copy the commit and you can test it in each code base to verify that it is fixed in all versions of the product. Let's go through the basic commands for a cherry pick. First, you'll use git log to see the list of commits and identify the commit you want to cherry pick. Note, that you can view the log from a different branch than the one you are currently on by adding the branch‑name. Next, check out the branch where you want to make a copy of the commit. Now you can use the cherry‑pick command to pass in the commit SHA and Git will append that commit to the HEAD of the current branch.

### Cherry Pick Demos

Let's simulate solving a difficult problem where you have two different things you're trying to discover, x and y. You'll make a branch to work on solving x and another branch to work on solving y. Once you find a solution for x and y, you'll cherry‑pick the solution for each back to main. Let's initialize the empty directory as a git repo, add some content to start out, (Typing) add, and commit. (Typing) Now let's create a branch to solve x. We'll create three different files to simulate looking for the solution for x. We'll commit each file. (Working) At this point, the branch has served its purpose in that we found the value for x. Let's go back to main and create a new branch to find the value for y. (Working) We'll do the same thing here. We'll create three files to simulate looking for a solution and commit each file. (Working) At this point, we've found the value for y. Let's take a look at the history for the y branch. Note that it has the original commit in main, then the three different commits to find the value. The commit we're actually interested in is the final one, where we found the value for y. Go ahead and copy this commit SHA, because we'll need it for the cherry pick. Now let's switch to the x branch to get the commit SHA for the solution for x. (Working) Here's the commit we want that has the solution for x. Save that off. Now let's switch back to main. With the two commit SHAs, we can cherry pick those to commits into main. Notice that right now we don't have the solutions, and our main history has only the single file that we created. Let's first cherry pick the value for x by running git cherry‑pick, then the commit SHA for x. Note that your commit SHA will be different. Now we can see the solution for x, and the commit has been copied into the history for main. Now let's cherry pick the solution for y. Now we have the value for y, and the history reflects the added commit. If we look at the history for the x branch, note that the commit SHA is different. This is because cherry‑pick makes a copy of the commit into the new branch. Even though the content is the same, the commit on the x branch is different than the commit on the main branch. We can run the log for the y branch and see the same thing. The value for y in the y branch has a different SHA than the value for y in the main branch. By utilizing branches to find the solution, then cherry picking the solution to main, the history for main is a lot easier to read and understand than if we had merged the entire x and y branch into main. Let's see how cherry‑pick can help you manage a repo that has multiple active branches, one for each version of a product that you're supporting. The product has some changes with the newest versions that make it unable to fully merge newer product versions into branches for older product versions. When you discover a bug in the latest version that also exists in previous versions, you'll see how cherry‑pick can help you apply that fix to all branches of the product. Let's initialize this empty directory as a Git repo, git init, add some content, (Working) and add and commit the file. Next, let's create a branch for the 1.4 version of the product, git checkout ‑b 1.4. We'll add and commit a file for this branch. (Working) Now we'll switch back to main and create a branch for the 2.1 version of the product. Normally, there would be several commits between 1.4 and 2.1, but that doesn't affect this example, so we'll just jump right in. Let's add and commit a file to the 2.1 branch. (Working) Now if we list the branches with git branch, note that there are active branches for both the 1.4 and 2.1 version of the product. Let's go back to main and create a branch for the 3.0 version. (Working) Suppose that while working on this branch we found a critical bug fix. (Working) We'll add and commit that to the 3.0 branch, and at this point, we discover that the same bug exists in the 1.4 and 2.1 branches of the product. Rather than redoing or copy/pasting that into the other branches, let's cherry‑pick the commit that we made in 3.0 to fix the branch over to the other two branches to test and apply the fix there. We first need to get the SHA for the commit that we just made in the 3.0 branch. Run git log ‑‑oneline to get the commit SHA. We'll go ahead and copy that. Now let's switch to the 1.4 branch. (Working) Note that everything is how you left it. There's no bugfix.text in the branch yet. Let's cherry pick the bug fix commit from 3.0 into 1.4. Run git cherry‑pick, then the commit SHA. Now when we list the files, we can see bugfix.text is in the 1.4 branch, and the log shows that commit as well. Now let's switch to the 2.1 branch. We can run the same cherry‑pick command here, which will copy over the file and add the commit to the history for 2.1. You can now independently test and verify the fix in each branch, then when you're done, switch back to 3.0 and continue your work. If we look at the log on 3.0, note again that cherry‑pick creates a copy of the fix. Notice that the commit SHA in the 3.0 branch is different than the commit SHA in the 2.1 branch for the critical bug fix. Congratulations! You've seen how Rebase can help you tidy up your history with squash commits, as well as how to pull in changes from another branch and have them sequenced with your changes. You've also worked through a couple of use cases for the cherry‑pick command; like when you only need specific commits from a branch or when you need to apply the same commit to several branches.