

A summary note of Git

Book

Installing Git. Click [here](#)

First-Time Git Setup

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

Your Identity

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

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

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

Your Editor

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

Your default branch name

By default Git will create a branch called master when you create a new repository with git init. From Git version 2.28 onwards, you can set a different name for the initial branch.

To set main as the default branch name do:

```
git config --global init.defaultBranch main
```

Git Basics

Getting a Git Repository

- You typically obtain a Git repository in one of two ways:
 1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or
 2. You can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work

1. Create a Directory

So Let us create a directory called **Hamid**

You need to type - `mkdir Hamid` , where `mkdir` means “Make Directory” and go to this directory. - `cd Hamid`, where `cd` stands for “Change Directory”. Now we can have our files in it.



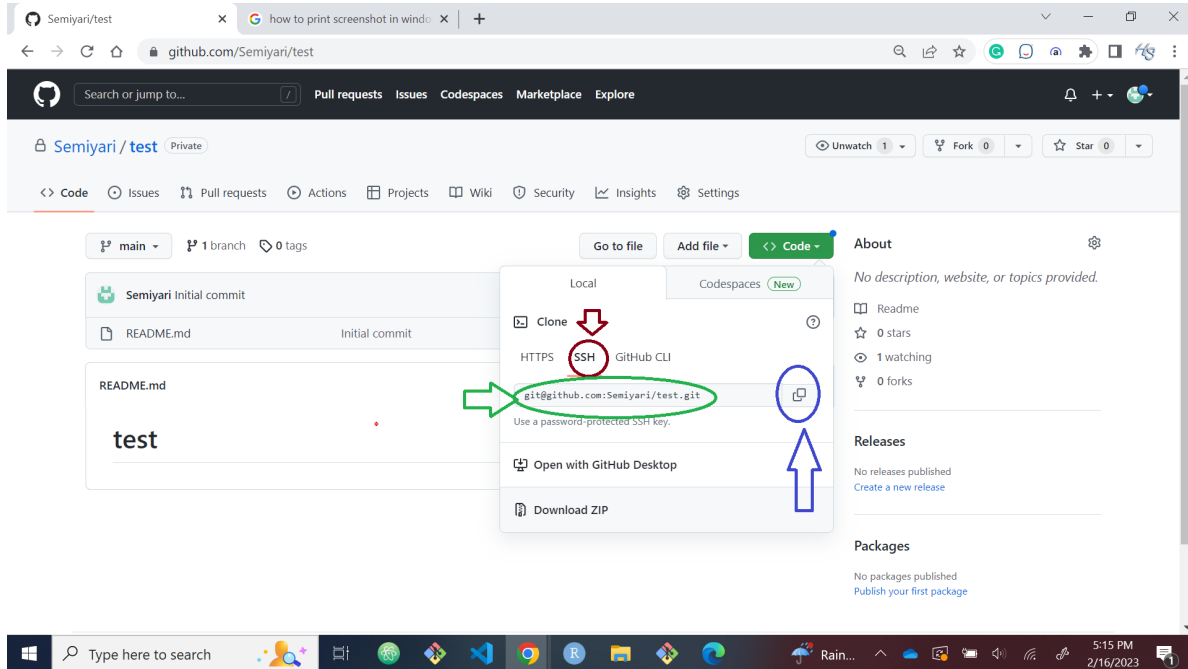
A screenshot of a MINGW64 terminal window. The title bar at the top reads "MINGW64: c:/Users/semyaii/Hamid". The terminal shows the following commands and output:

```
semyari@AU302413 MINGW64 ~  
$ mkdir Hamid  
  
semyari@AU302413 MINGW64 ~  
$ cd Hamid  
  
semyari@AU302413 MINGW64 -./Hamid  
$
```


A large red arrow points from the right towards the prompt "semyari@AU302413 MINGW64 -./Hamid". A red horizontal line is drawn under the prompt, and a small red dot is positioned below it.

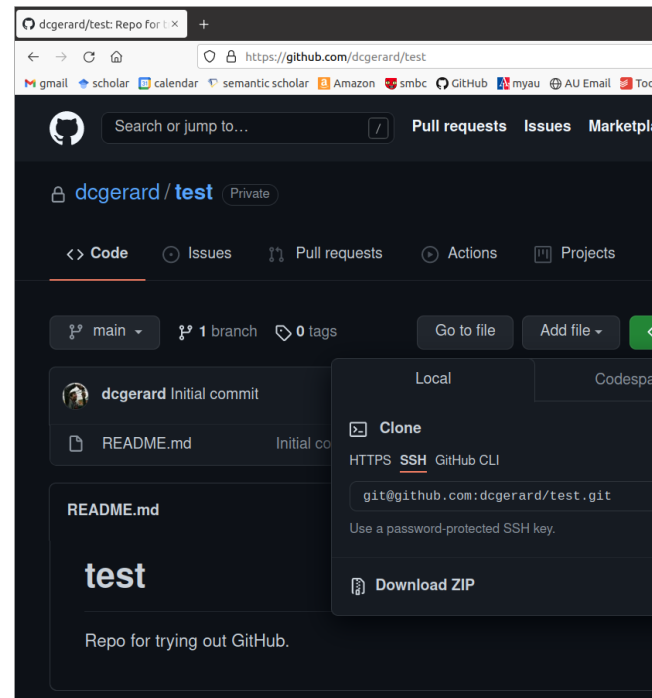
We want to add files to a Git repository. We have to initialize a new empty repository. - So we type `git init` - The `git init` is a one-time command you use during the initial setup of a new repo. Executing this command will create a new `.git` subdirectory in your current working directory.

2. Clone a Repository using “SSH”



- Make sure that “SSH” is highlighted.

- Then click on the  button to copy the link.



- In the terminal, navigate to where you want to download the repo, then clone it with `git clone`

```
git clone git@github.com:dcgerard/test.git
```

Make sure to change the link to what you copied (don't use my link above).

- Then move into your new repo

```
ls  
cd test
```

Recording Changes to the Repository

At this point, you should have a bona fide Git repository on your local machine, and a checkout or working copy of all of its files in front of you. Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Checking the Status of Your Files

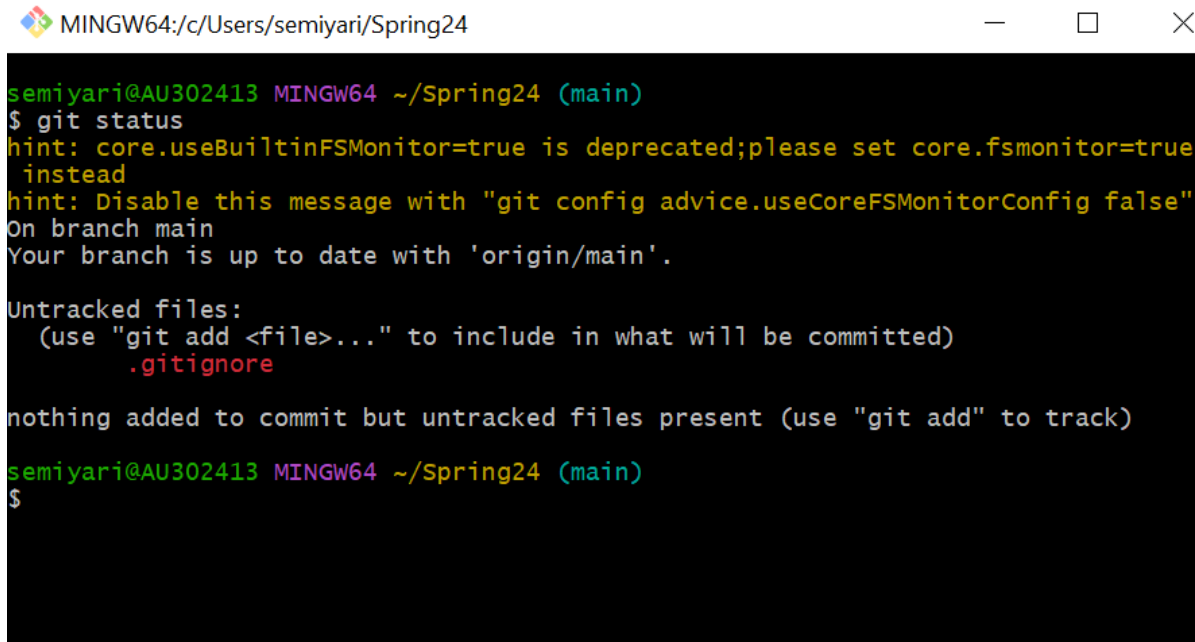
The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
```

```
On branch master  
Your branch is up-to-date with 'origin/master'.  
nothing to commit, working tree clean
```

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server.

I have created a directory called it Spring24 and added the `.gitignore` to my project folder. If I run the `git status` command I will get

A terminal window titled 'MINGW64:/c/Users/semyari/Spring24' with standard window controls. The terminal shows the output of the 'git status' command. It includes a hint about 'core.useBuiltinFSMonitor=true' being deprecated, a message stating the branch is up to date with 'origin/main', and a section for 'Untracked files' listing '.gitignore'. A final message states 'nothing added to commit but untracked files present (use "git add" to track)'. The prompt '\$' is shown at the end.

```
semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true
instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
semyari@AU302413 MINGW64 ~/Spring24 (main)
$
```

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

Tracking New Files

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

```
git add .gitignore
```

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

```
MINGW64:/c/Users/semiyari/Spring24

semiyari@AU302413 MINGW64 ~/Spring24 (main)
$ git add .gitignore
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true
instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
warning: in the working copy of '.gitignore', LF will be replaced by CRLF the ne
xt time Git touches it

semiyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true
instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore

semiyari@AU302413 MINGW64 ~/Spring24 (main)
$ |
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the subsequent historical snapshot.

Short Status

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

```
semyari@AU302413 MINGW64 ~/Spring24 (main)
$ cat README.md
# Spring24
semyari@AU302413 MINGW64 ~/Spring24 (main)
$ echo "DATA 413/613" >> README.md

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status -s
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
A .gitignore
M README.md

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status --short
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
A .gitignore
M README.md
```

staging area

Working Tree

Right column Working Tree

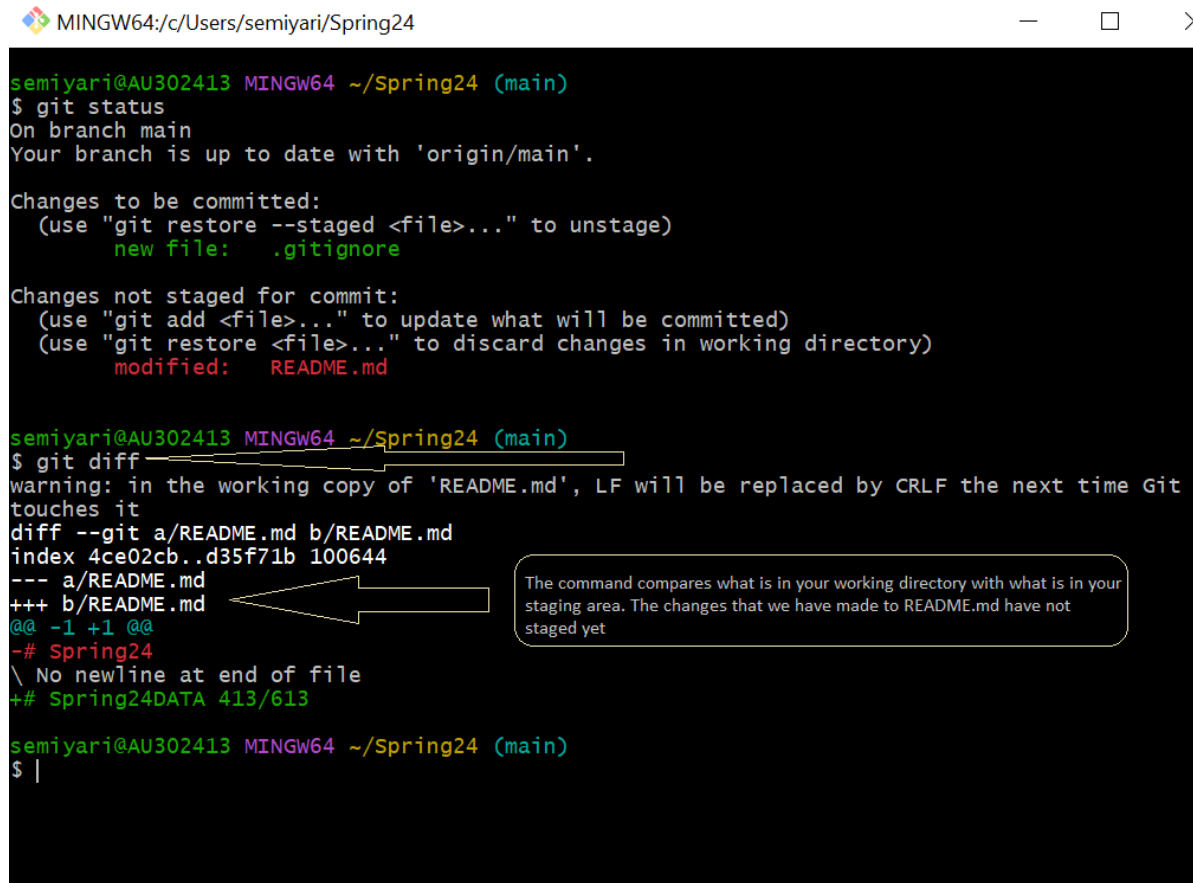
Left column Staging Tree

New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.

Viewing Your Staged and Unstaged Changes

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


```
git diff
```



```
MINGW64:/c/Users/semyari/Spring24

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git diff
warning: in the working copy of 'README.md', LF will be replaced by CRLF the next time Git
touches it
diff --git a/README.md b/README.md
index 4ce02cb..d35f71b 100644
--- a/README.md
+++ b/README.md
@@ -1,1 @@
-# Spring24
\ No newline at end of file
+# Spring24DATA 413/613

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ |
```

The command compares what is in your working directory with what is in your staging area. The changes that we have made to README.md have not staged yet

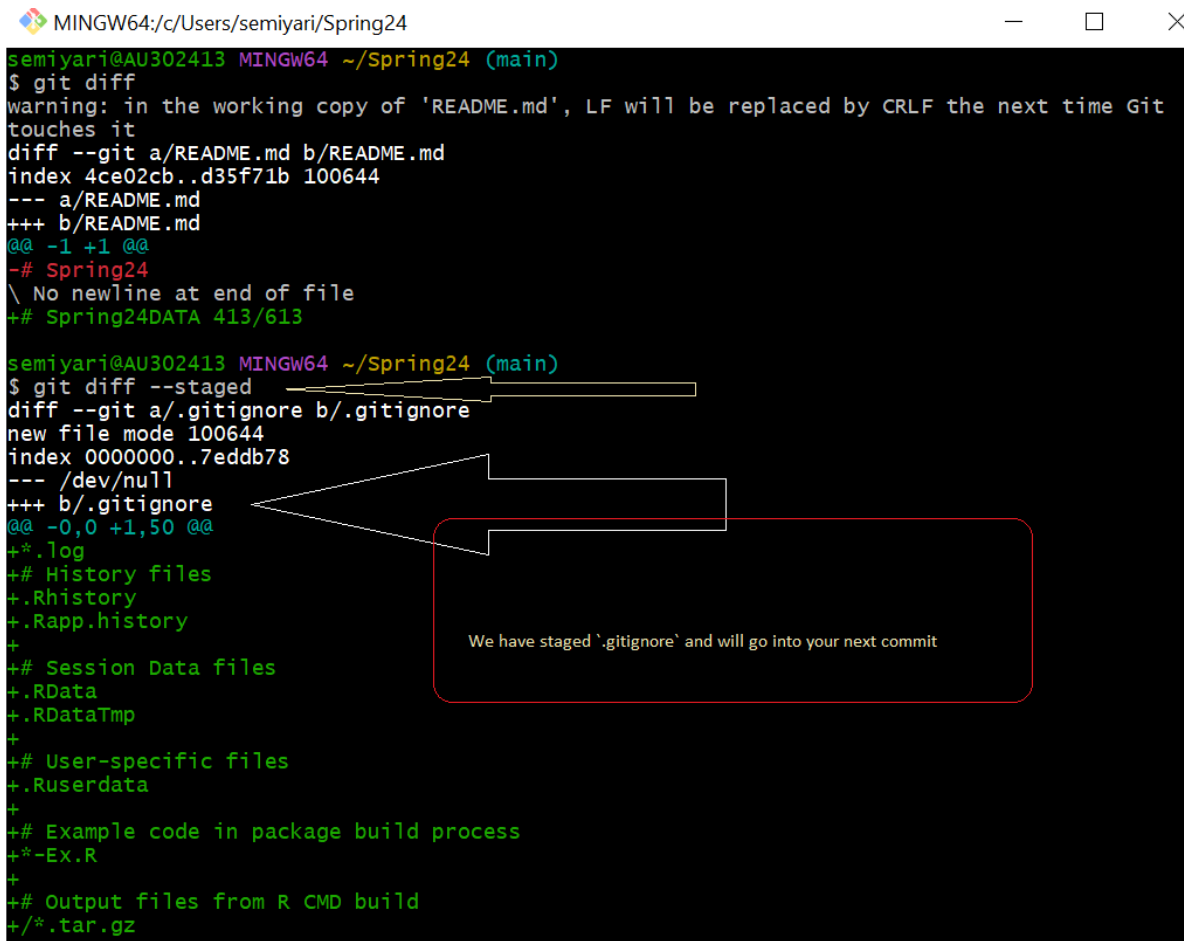
That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

- Lines after b “+” are being added. Lines after a “-” are being removed.
- `git diff` won’t check for changes in the staged files by default. But if you want to see what you’ve staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
git diff --staged
```

```
MINGW64:/c/Users/semyari/Spring24
semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git diff
warning: in the working copy of 'README.md', LF will be replaced by CRLF the next time Git
touches it
diff --git a/README.md b/README.md
index 4ce02cb..d35f71b 100644
--- a/README.md
+++ b/README.md
@@ -1,1 @@
-# Spring24
\ No newline at end of file
+# Spring24DATA 413/613

semyari@AU302413 MINGW64 ~/Spring24 (main)
$ git diff --staged
diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..7eddb78
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,50 @@
+*.log
+# History files
+.Rhistory
+.Rapp.history
+
+# Session Data files
+.RData
+.RDataTmp
+
+# User-specific files
+.Ruserdata
+
+# Example code in package build process
+*-Ex.R
+
+# Output files from R CMD build
+/**.tar.gz
```



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

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged — any files you have created or modified that you haven't run `git add` on since you edited them — won't go into this commit. They will stay as modified files on your disk.

The simplest way to commit is to type `git commit`:

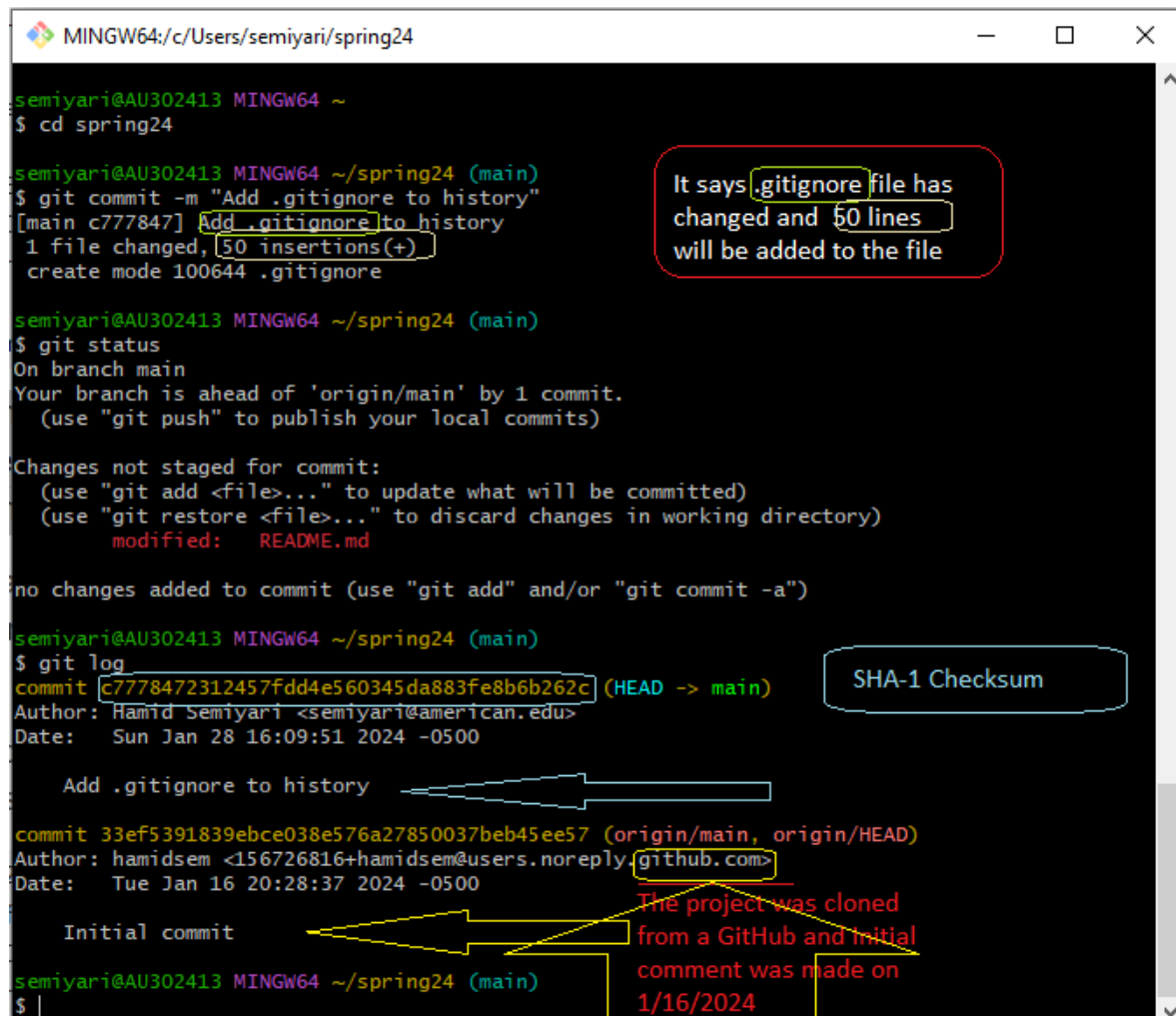
```
git commit -m "Write a concise, and describe what has been changed since the last commit"
```

- Let us commit the .gitignore

```
git commit -m "Add .gitignore to history"
```

Viewing the Commit History

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



```

MINGW64; c:/Users/semiyari/spring24
semiyari@AU302413 MINGW64 ~
$ cd spring24

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git commit -m "Add .gitignore to history"
[main c777847] Add .gitignore to history
1 file changed, 50 insertions(+)
create mode 100644 .gitignore

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

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

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git log
commit c7778472312457fdd4e560345da883fe8b6b262c (HEAD -> main)
Author: Hamid Semiyari <semiyari@american.edu>
Date:   Sun Jan 28 16:09:51 2024 -0500

    Add .gitignore to history

commit 33ef5391839ebce038e576a27850037beb45ee57 (origin/main, origin/HEAD)
Author: hamidsem <156726816+hamidsem@users.noreply.github.com>
Date:   Tue Jan 16 20:28:37 2024 -0500

    Initial commit
  
```

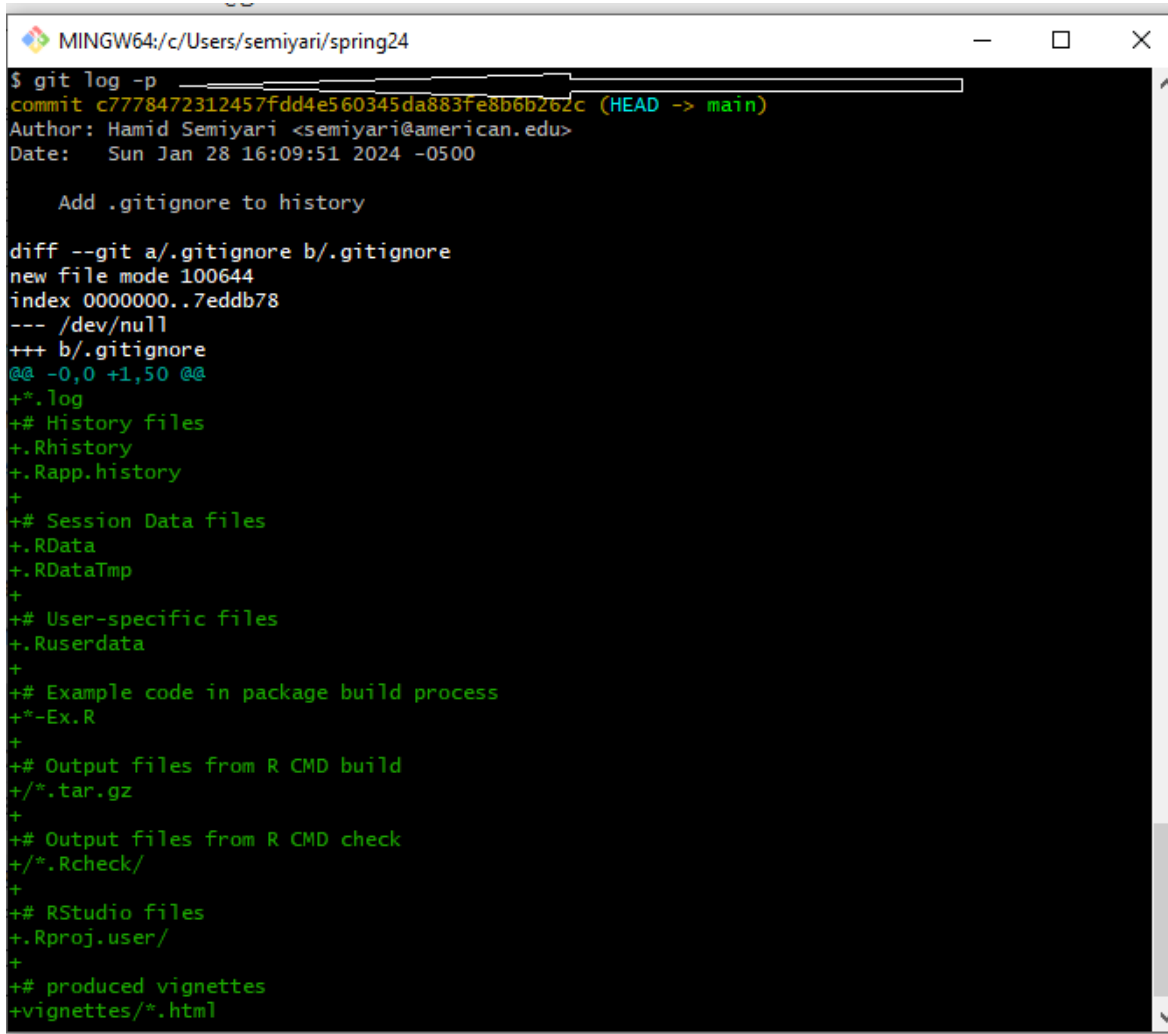
Annotations in the screenshot:

- A red box highlights the commit message "Add .gitignore to history" and the output "1 file changed, 50 insertions(+)", with a note: "It says .gitignore file has changed and 50 lines will be added to the file".
- A blue box highlights the SHA-1 checksum "c7778472312457fdd4e560345da883fe8b6b262c" with the label "SHA-1 Checksum".
- A yellow box highlights the "Initial commit" message, with a note: "The project was cloned from a GitHub and initial commit was made on 1/16/2024".

- By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can

see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

- One of the more helpful options is `-p` or `--patch`, which shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using `-2` to show only the last two entries.



```
$ git log -p
commit c7778472312457fdd4e560345da883fe8b6b262c (HEAD -> main)
Author: Hamid Semiyari <semiyari@american.edu>
Date:   Sun Jan 28 16:09:51 2024 -0500

    Add .gitignore to history

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..7eddb78
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,50 @@
+*.log
+# History files
+.Rhistory
+.Rapp.history
+
+# Session Data files
+.RData
+.RDataTmp
+
+# User-specific files
+.Ruserdata
+
+# Example code in package build process
+*~Ex.R
+
+# Output files from R CMD build
+/**.tar.gz
+
+# Output files from R CMD check
+/**.Rcheck/
+
+# RStudio files
+.Rproj.user/
+
+# produced vignettes
+vignettes/*.html
```

Undoing Things

- At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

AMEND

- `--amend` One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the `-amend` option:

This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message. + Case 1: Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit's message without altering its snapshot. i) The command line without `-m`

```
git commit --amend
```

The default text editor will be opened up in which you can replace the message from your older commit. Save and exit the text editor and your change will be made.

ii) the command line

```
git commit --amend -m "Your Message"
```

enter the your correct message.

- Case 2: Say that your last commit is missing a crucial file. Without it, your project worth nothing
- Add the missing file.
- Stage the changes:

```
git add <missing_file>
```

- Now, amend the last commit we use

```
git commit --amend
```

Or


```
git commit --amend --no-edit
```

This command will change the files in your last commit. It will not change the message associated with the commit because we have not used the `-m` flag.

- For more read [2.4 Git Basics - Undoing Things](#)

Working with Remote

- To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.
 - To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you’ve specified. If you’ve “cloned” your repository, you should at least see `origin`
 - You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

 MINGW64:/c/Users/semiyari/spring24

```
semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote
origin

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote -v
origin  git@github.com:semiyarih/Spring24.git (fetch)
origin  git@github.com:semiyarih/Spring24.git (push)
```

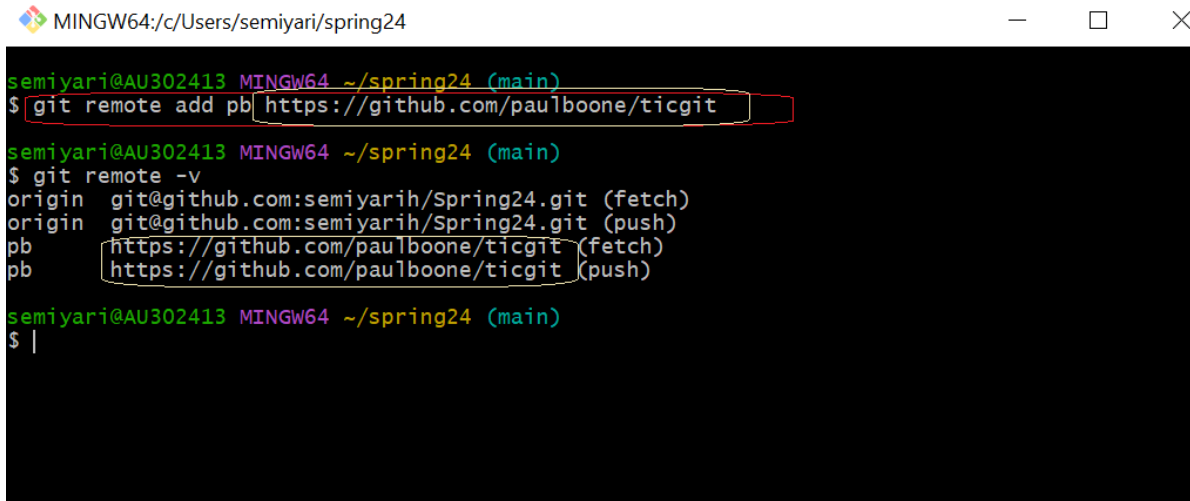
If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

```
$ git remote -v bakkdoor https://github.com/bakkdoor/grit (fetch) bakkdoor https://github.com/bakkdoor/grit
(push) cho45 https://github.com/cho45/grit (fetch) cho45 https://github.com/cho45/grit
(push) defunkt https://github.com/defunkt/grit (fetch) defunkt https://github.com/defunkt/grit
(push) koke git://github.com/koke/grit.git (fetch) koke git://github.com/koke/grit.git (push)
origin git@github.com:mojombo/grit.git (fetch) origin git@github.com:mojombo/grit.git
(push)
```

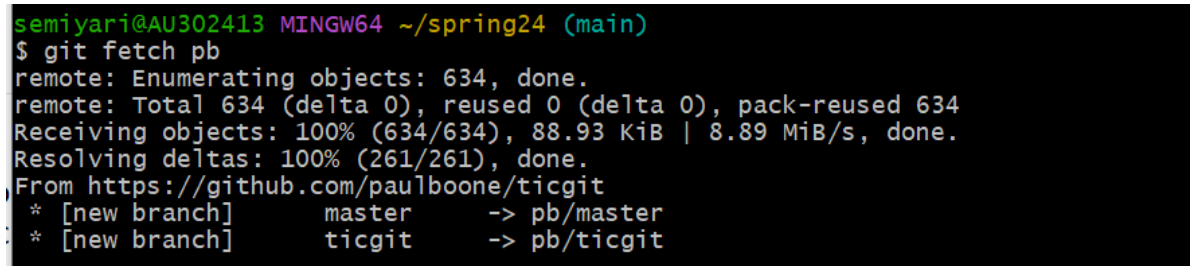
This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can’t tell that here.

Adding Remote Repositories We’ve mentioned and given some demonstrations of how the “git clone” command implicitly adds the “origin remote” for you. Here’s how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run

```
git remote add <shortname> <url>
```

A terminal window titled 'MINGW64:/c/Users/semiyari/spring24' with standard window controls. The prompt is 'semiyari@AU302413 MINGW64 ~/spring24 (main)'. The first command is '\$ git remote add pb https://github.com/paulboone/ticgit', with 'pb' and the URL highlighted by red boxes. The second command is '\$ git remote -v', which lists the remote configurations: 'origin git@github.com:semiyarih/Spring24.git (fetch)', 'origin git@github.com:semiyarih/Spring24.git (push)', 'pb https://github.com/paulboone/ticgit (fetch)', and 'pb https://github.com/paulboone/ticgit (push)'. The 'pb' and the URLs are highlighted by red boxes. The prompt returns to '\$ |'.

Now you can use the string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `git fetch pb`:

A terminal window showing the output of the command '\$ git fetch pb'. The output is: 'remote: Enumerating objects: 634, done.', 'remote: Total 634 (delta 0), reused 0 (delta 0), pack-reused 634', 'Receiving objects: 100% (634/634), 88.93 KiB | 8.89 MiB/s, done.', 'Resolving deltas: 100% (261/261), done.', 'From https://github.com/paulboone/ticgit', '* [new branch] master -> pb/master', and '* [new branch] ticgit -> pb/ticgit'. The prompt returns to '\$ |'.

Paul's master branch is now accessible locally as `pb/master` — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it. We'll go over what branches are and how to use them in much more detail in **Git Branching Section**.

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run:

```
git fetch <remote>
```

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

- It's important to note that the `git fetch` command only downloads the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push <remote> <branch>`. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push. See **Git Branching Section** for more detailed information on how to push to remote servers. ### Inspecting a Remote If you want to see more information about a particular remote, you can use the `git remote show <remote>` command. If you run this command with a particular shortname, such as origin, you get something like this:

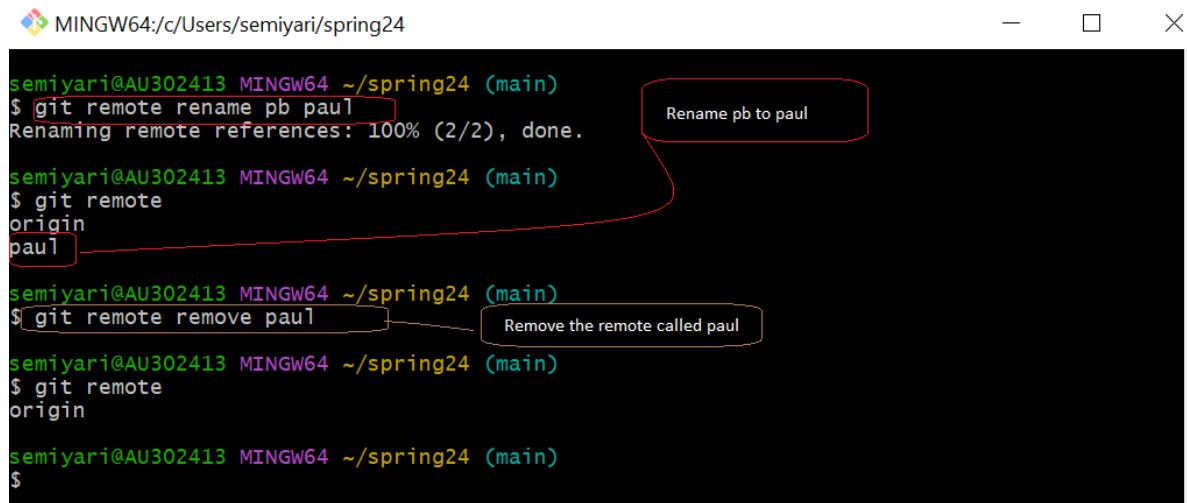
```
semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote show origin
* remote origin
  Fetch URL: git@github.com:semiyarih/Spring24.git
  Push URL: git@github.com:semiyarih/Spring24.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (fast-forwardable)
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the master branch and you run `git pull`, it will automatically merge the remote's master branch into the local one after it has been fetched. It also lists all the remote references it has pulled down.

Renaming and Removing Remotes

You can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`

- It's worth mentioning that this changes all your remote-tracking branch names, too. What used to be referenced at `pb/master` is now at `paul/master`
- If you want to remove a remote for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can either use `git remote remove` or `git remote rm`:



```
MINGW64; c:/Users/semiyari/spring24
semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote rename pb paul
Renaming remote references: 100% (2/2), done.

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote
origin
paul

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote remove paul

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git remote
origin

semiyari@AU302413 MINGW64 ~/spring24 (main)
$
```

Annotations in the image:

- A red box around `git remote rename pb paul` is connected by a red line to a red-bordered callout box containing the text "Rename pb to paul".
- A red box around `git remote remove paul` is connected by a red line to a red-bordered callout box containing the text "Remove the remote called paul".

Git Branching:

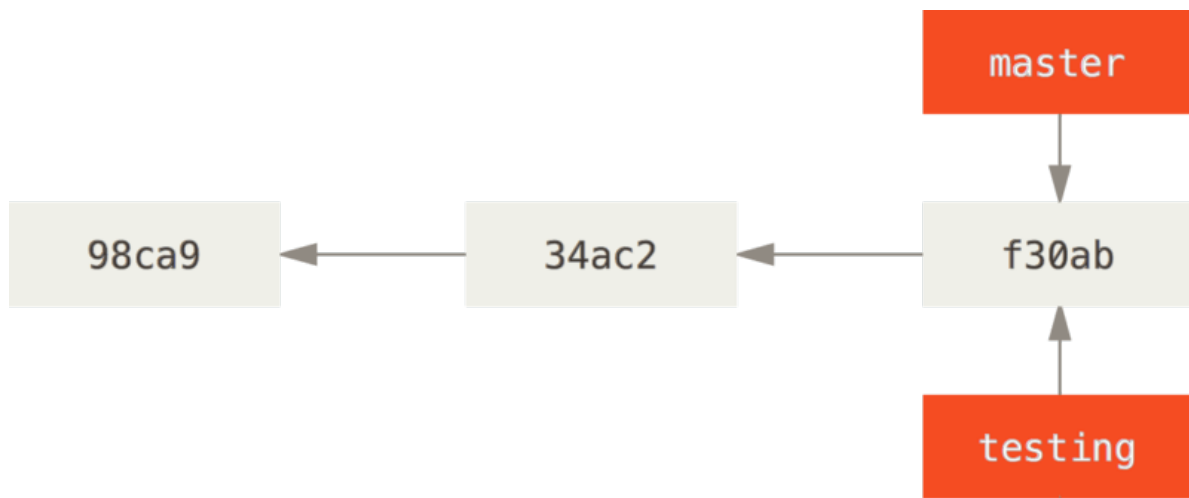
- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- A branch is an “alternative universe” of your project, where you can experiment with new ideas (e.g. new data analyses, new data transformations, new statistical methods). After experimenting, you can then “merge” your changes back into the main branch.
- Branching isn't just for group collaborations, you can use branching to collaborate with yourself, e.g., if you have a new idea you want to play with but do not want to have that idea in main yet.
- The “main” branch (the default in GitHub) is your best draft. You should consider anything in “main” as the best thing you've got.
- The workflow using branches consists of

1. Create a branch with an informative title describing its goal(s).
2. Add commits to this new branch.
3. Merge the commits to main

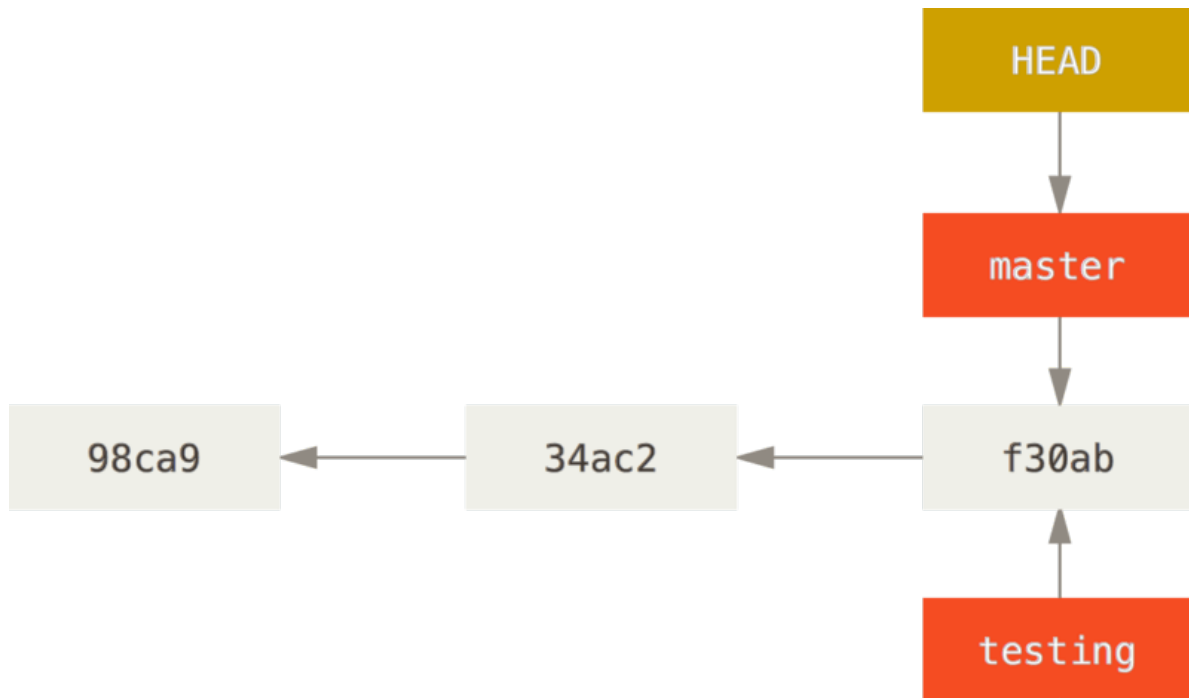
Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you want to create a new branch called testing. You do this with the git branch command:

```
git branch testing
```



- **How does Git know what branch you're currently on?** It keeps a special pointer called HEAD. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master. The git branch command only created a new branch — it didn't switch to that branch.



You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
git log --oneline --decorate
```

```
MINGW64:/c/Users/semyari/spring24

semyari@AU302413 MINGW64 ~/spring24 (main)
$ git branch testing

semyari@AU302413 MINGW64 ~/spring24 (main)
$ git log --oneline --decorate
db6dd67 (HEAD -> main, testing) hange the commit by --amend
a7ef36d Modify README.md
c777847 Add .gitignore to history
33ef539 (origin/main, origin/HEAD) Initial commit

semyari@AU302413 MINGW64 ~/spring24 (main)
$
```

You can see the master and testing branches that are right there next to the f30ab commit. *Note your Checksum (and mine) are different than f30ab*

Switching Branches

To switch to an existing branch, you run the `git checkout` command. Let's switch to the new testing branch:

```
git checkout testing
```

This moves HEAD to point to the testing branch.



What is the significance of that? Well, let's do another commit:

```
MINGW64:/c/Users/semiyari/spring24
$ git log --oneline --decorate
db6dd67 (HEAD -> main, testing) hange the commit by --amend
a7ef36d Modify README.md
c777847 Add .gitignore to history
33ef539 (origin/main, origin/HEAD) Initial commit

semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git checkout testing
Switched to branch 'testing'
D       file.txt

semiyari@AU302413 MINGW64 ~/spring24 (testing)
$ echo "Hello World. New line." >> file.txt

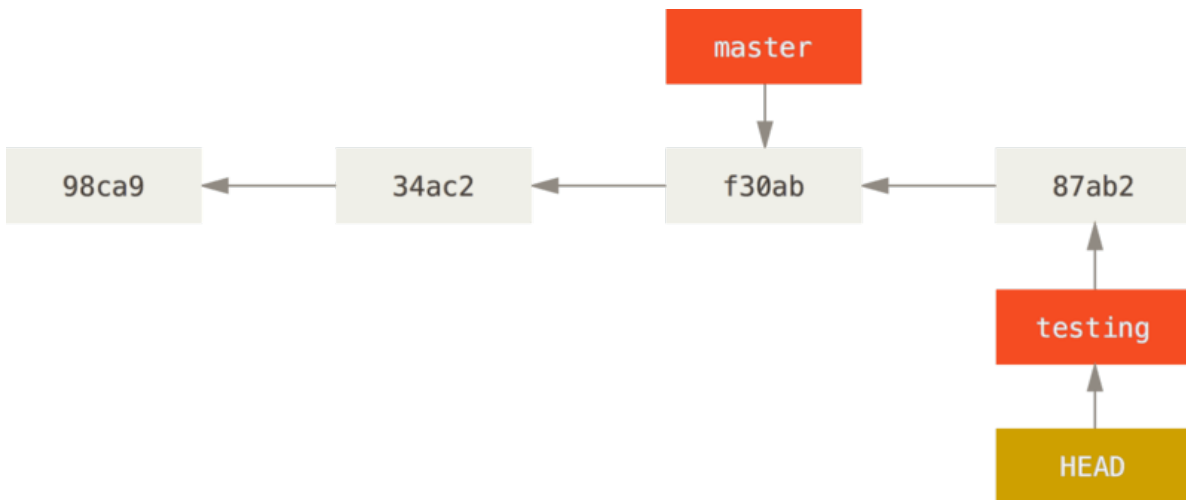
semiyari@AU302413 MINGW64 ~/spring24 (testing)
$ git status
On branch testing
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    file.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file.txt

semiyari@AU302413 MINGW64 ~/spring24 (testing)
$ git add file.txt
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next time Git touches it

semiyari@AU302413 MINGW64 ~/spring24 (testing)
$ git commit -m "Make a change - New Branch"
[testing 634aa3a] Make a change - New Branch
 1 file changed, 1 insertion(+), 1 deletion(-)

semiyari@AU302413 MINGW64 ~/spring24 (testing)
$ |
```



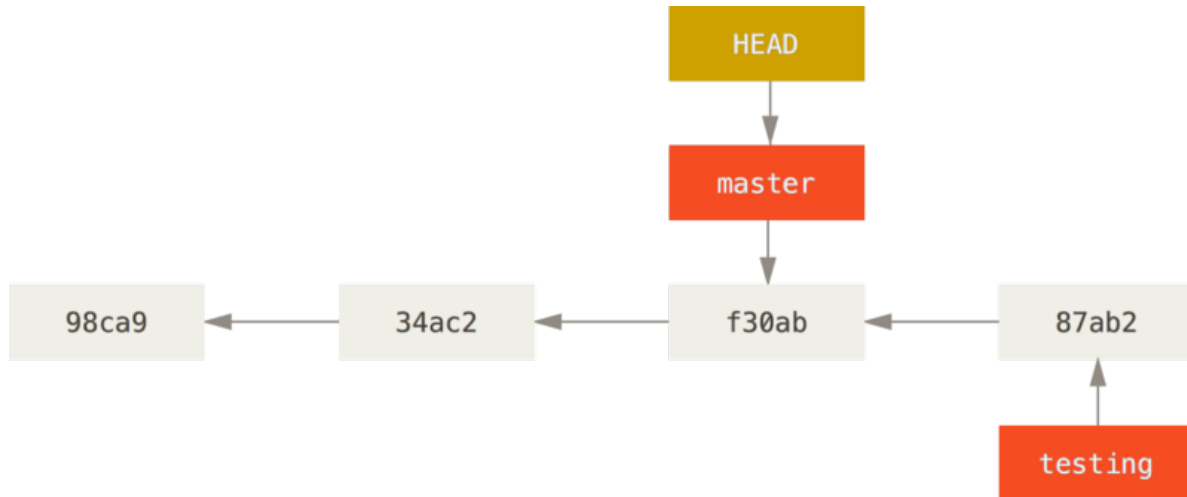
This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches.

Let's switch back to the master branch:

```
semyari@AU302413 MINGW64 ~/spring24 (testing)
$ git branch
* main
  testing

semyari@AU302413 MINGW64 ~/spring24 (testing)
$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 3 commits.
(use "git push" to publish your local commits)

semyari@AU302413 MINGW64 ~/spring24 (main)
```



That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your testing branch so you can go in a different direction.

Let's make a few changes and commit again:

```

$ echo "This line is added when I switched back to main branch" >> file1.txt

emiyari@AU302413 MINGW64 ~/spring24 (main)
ls
README.md  file.txt  file1.txt

emiyari@AU302413 MINGW64 ~/spring24 (main)
git add file1.txt
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the next time Git
touches it

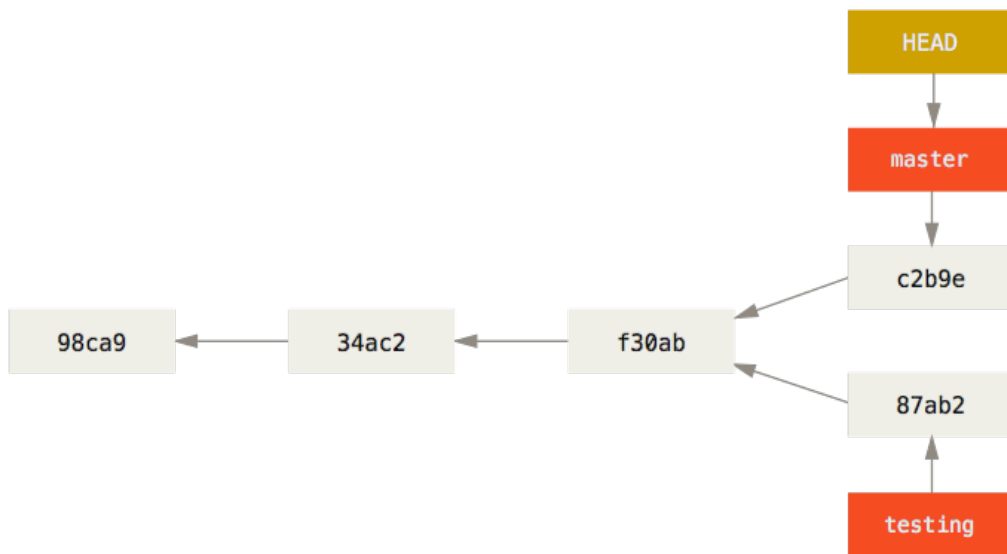
emiyari@AU302413 MINGW64 ~/spring24 (main)
git commit -m "Add file - Switch back to main branch"
main 3eb698c] Add file - Switch back to main branch
1 file changed, 1 insertion(+)
create mode 100644 file1.txt

```

Add a new file to Working Tree

Add file to staging Tree and then commit it.

Now your project history has diverged. You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple branch, checkout, and commit commands.



You can also see this easily with the git log command. If you run `git log --oneline --decorate --graph --all` it will print out the history of your commits, showing where your

MINGW64:/c/Users/semiyari/spring24

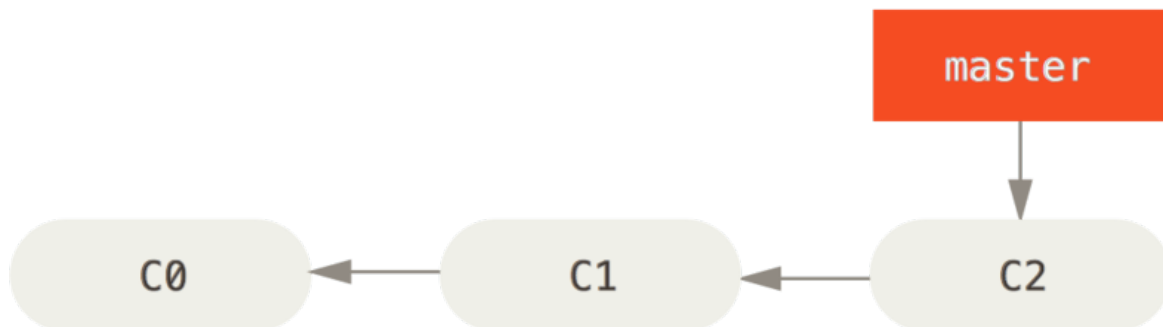
```
semiyari@AU302413 MINGW64 ~/spring24 (main)
$ git log --oneline --decorate --graph --all
* 3eb698c (HEAD -> main) Add file - Switch back to mai
| * 634aa3a (testing) Make a change - New Branch
|/
* db6dd67 hange the commit by --amend
* a7ef36d Modify README.md
* c777847 Add .gitignore to history
* 33ef539 (origin/main, origin/HEAD) Initial commit
semiyari@AU302413 MINGW64 ~/spring24 (main)
```

branch pointers are and how your history has diverged.

Basic Branching and Merging

Basic Branching

- First, let's say you're working on your project and have a couple of commits already on the master branch.



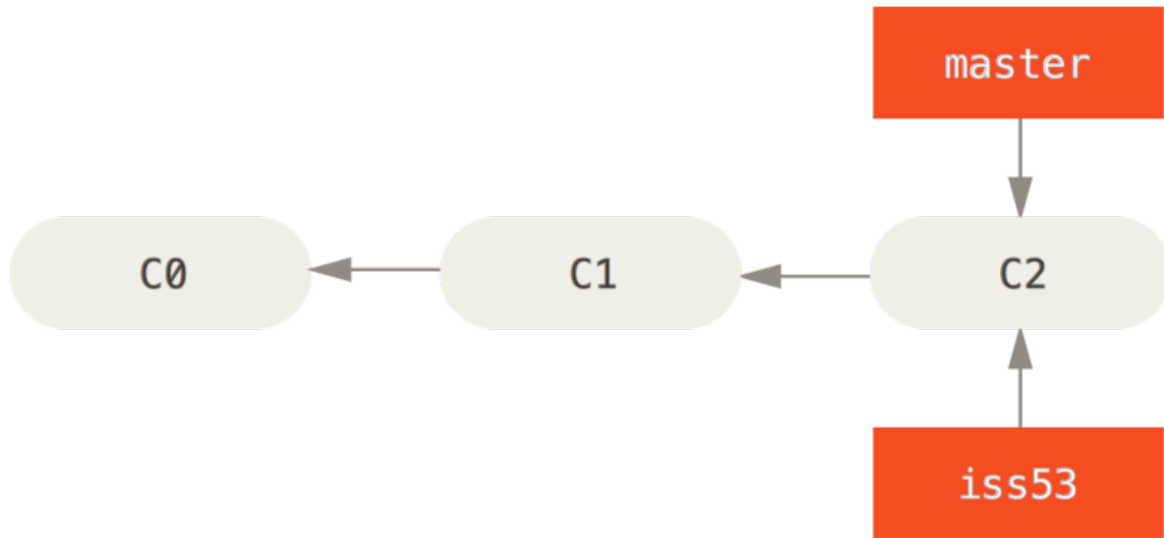
You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch (iss53) and switch to it at the same time, you can run

```
git checkout -b iss53    # Create "iss53" branch and switch branch to "iss53"
```

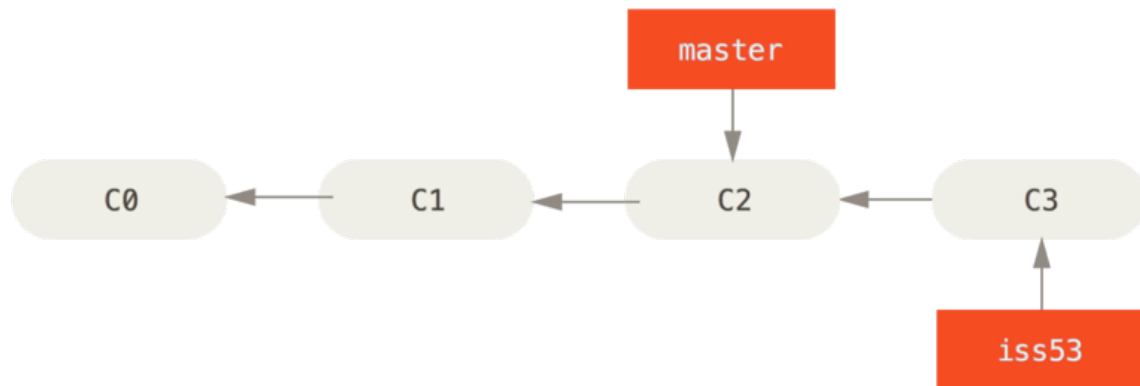
This is shorthand for:

```
git branch iss53        # Creat a new branch named "iss53"
```

```
git checkout iss53      # Switch to branch "iss53"
```

You work on your website and do some commits. Doing so moves the `iss53` branch forward, because you have it checked out (that is, your `HEAD` is pointing to it)



Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, all you have to do is switch back to your `master` branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches. Let's assume you've committed all your changes, so you can switch back to your `master` branch:

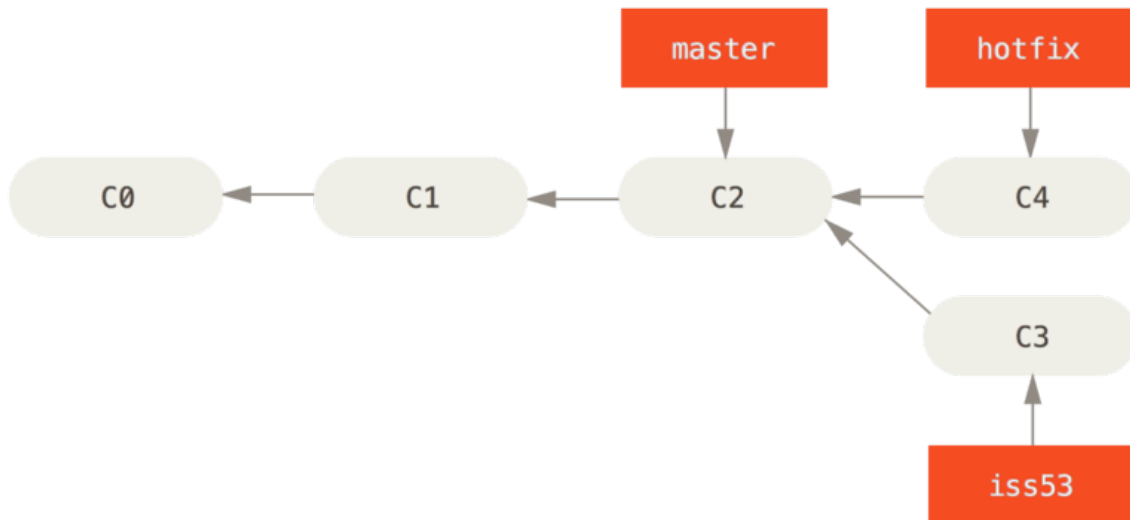
```
git checkout master    # Switch to branch "master"
```

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix.

- This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a **hotfix** branch on which to work until it's completed:

```
git checkout -b hotfix    # Create "hotfix" branch and switch branch to "hotfix"
```



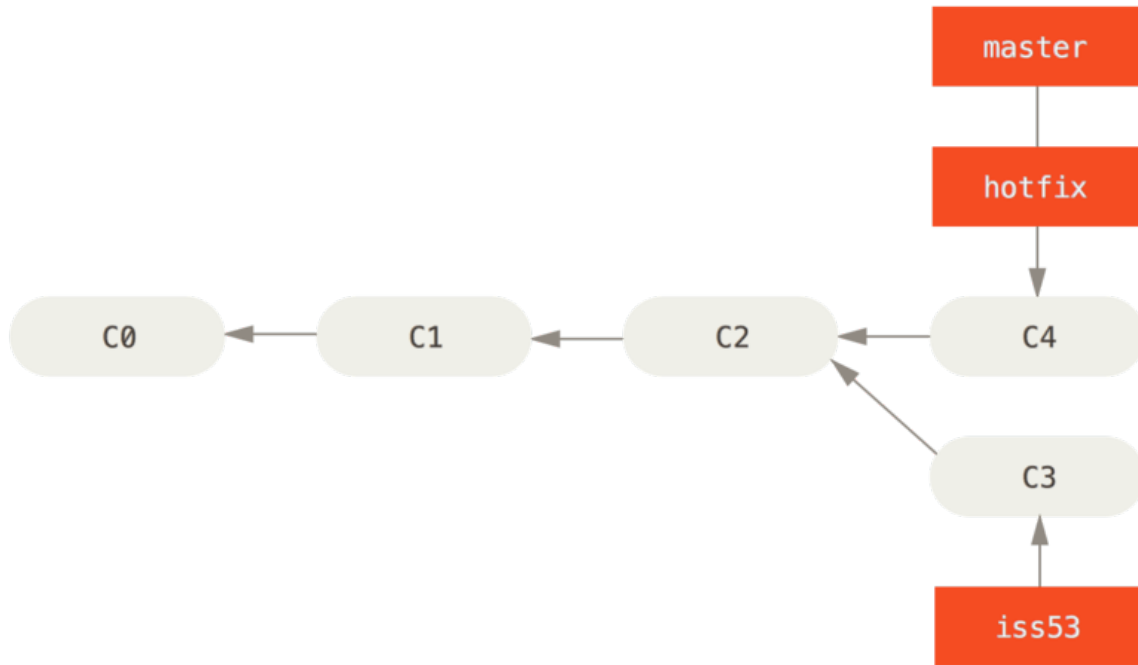
Now, you work on this issue after you fix it you want to run your tests to make sure the hotfix is what you want, and finally “merge” the **hotfix** branch back into your **master** branch to deploy to production. You do this with the `git merge` command

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

You'll notice the phrase “fast-forward” in that merge. Because the commit **C4** pointed to by the branch **hotfix** you merged in was directly ahead of the commit **C2** you're on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things

by moving the pointer forward because there is no divergent work to merge together — this is called a “fast-forward.”

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.

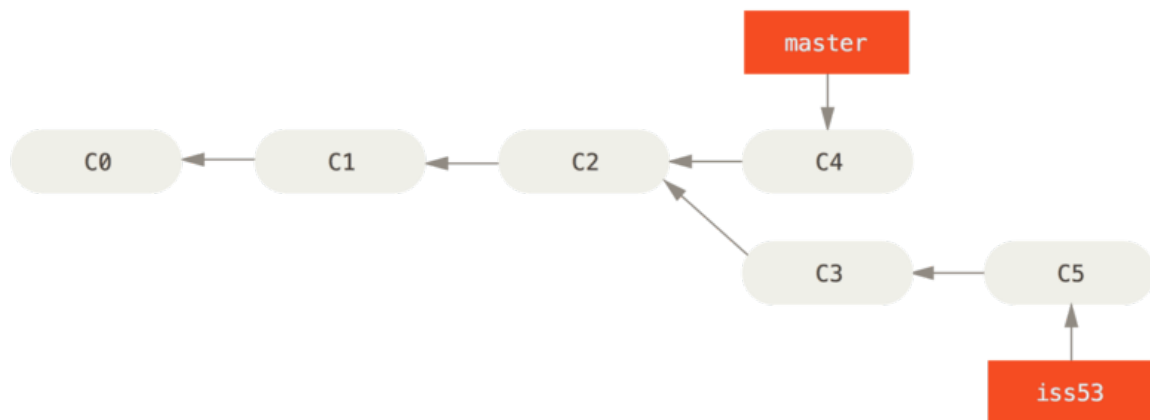


After your super-important fix is deployed, you’re ready to switch back to the work you were doing before you were interrupted. However, first you’ll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the `-d` option to `git branch`:

```
git branch -d hotfix
```

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

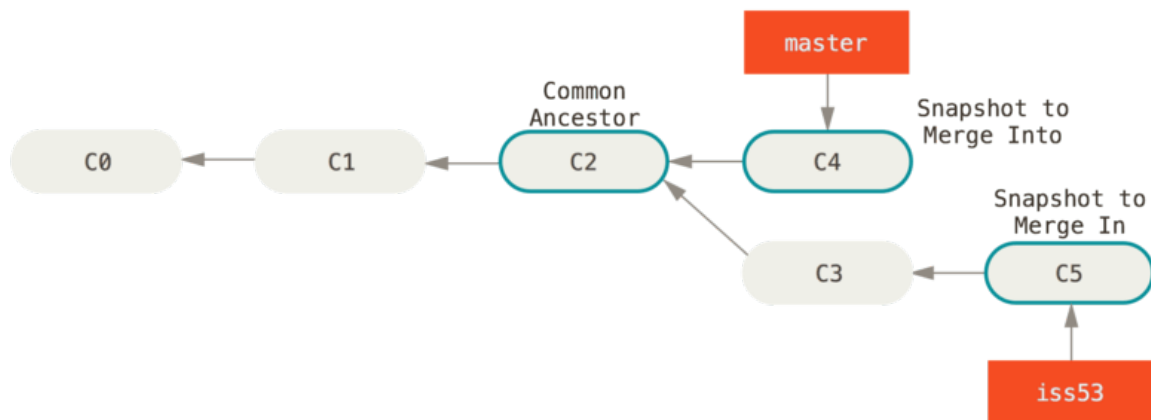
```
git checkout iss53
```



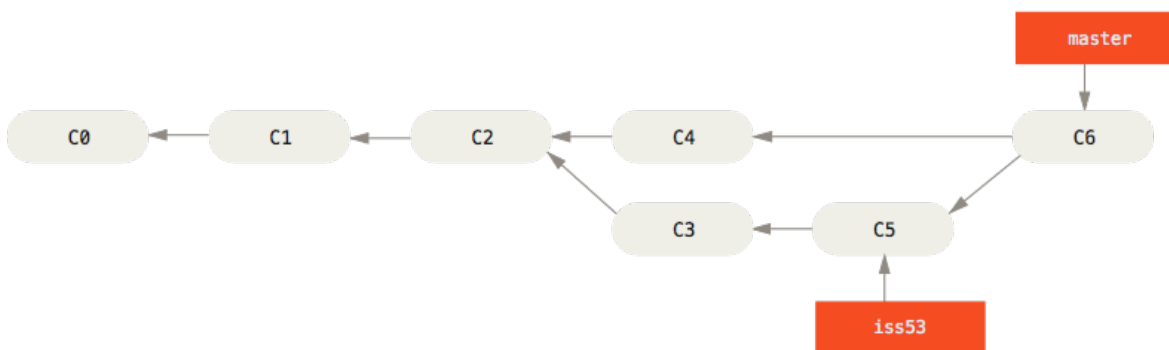
Basic Merging Suppose you've decided that your issue #53 work is complete and ready to be merged into your **master** branch. In order to do that, you'll merge your **iss53** branch into **master**, much like you merged your **hotfix** branch earlier. All you have to do is check out the branch you wish to merge into and then run the **git merge** command:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

This looks a bit different than the **hotfix** merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple *three-way merge*, using the two snapshots pointed to by the branch tips and the common ancestor of the two.



Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this *three-way merge* and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.



Now that your work is merged in, you have no further need for the `iss53` branch. You can close the issue in your issue-tracking system, and delete the branch:

```
git branch -d iss53
```

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly. If

your fix for issue #53 modified the same part of a file as the `hotfix` branch, you'll get a *merge conflict* that looks something like this:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   index.html
```

no changes added to commit (use “git add” and/or “git commit -a”) Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

This means the version in `HEAD` (your `master` branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the `=====`), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution has a little of each section, and the <<<<<<, =====, and >>>>>> lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ git mergetool
```

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge a

Merging:

index.html

Normal merge conflict for 'index.html':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (opendiff):

If you want to use a merge tool other than the default (Git chose `opendiff` in this case because the command was run on macOS), you can see all the supported tools listed at the top after “one of the following tools.” Just type the name of the tool you'd rather use.

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run `git status` again to verify that all conflicts have been resolved:

```
$ git status
```

```
On branch master
```

```
All conflicts fixed but you are still merging.
```

```
(use "git commit" to conclude merge)
```

```
Changes to be committed:
```

```
    modified:   index.html
```

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
Merge branch 'iss53'
```

```
Conflicts:
```

```
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

Branch Management

- `git branch` If you run it with no arguments, you get a simple listing of your current branches.

```
$ git branch
  iss53
* master
  testing
```

Notice the `*` character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to).

- To see the last commit on each branch, you can run `git branch -v`, `git branch -v`

```
$ git branch -v
  iss53    93b412c Fix javascript issue          **Last Commit in brach 'iss53'*
* master  7a98805 Merge branch 'iss53'          **Last Commit in brach 'master'*
  testing  782fd34 Add scott to the author list in the readme **Last Commit in brach 'testing'
```

- To see which branches are already merged into the branch you're on, you can run `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

- To see all the branches that contain work you haven't yet merged in, you can run `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

- If a branch contains work but you do not need it and you try to delete it with `-d` it fails.

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

- Rename the branch locally with the `git branch --move` command:

```
$ git branch --move bad-branch-name corrected-branch-name
```

This replaces your `bad-branch-name` with `corrected-branch-name`, but this change is only **local** for now. To let others see the corrected branch on the remote, **push** it:

```
$ git push --set-upstream origin corrected-branch-name
```

Pul - Fork

- please go over the note “8c_pull_fork.html”