

Git Tutorial

Amir Kooshky (kooshkya@gmail.com)

Maedeh Heydari (maedeh.heydari212@gmail.com)

Advanced Programming
Sharif University of Technology
Spring of 2023



What is Git?

Git offers countless features, one of which is giving you an **“undo on steroids”** button.

We will use this analogy a lot: Many times, before making a big change to the project, developers make a copy of their entire project directory and then make their desired changes to the original project. This way if they break something, they always have the contingency of bringing back the copy and restarting from there. One of the many applications of Git is facilitating this process. At its core, that's practically what it's doing for you! The advantage over your classic “copy” solution is the plethora of extra features that it offers.

Also, Git will allow you to collaborate with others on the same project and work on different parts of it simultaneously, and then eventually merge all the changes you made separately into a single, final version.

- Check git installation/version:

```
$ git -version
```

```
C:\Users\Logan>git --version  
git version 2.34.1.windows.1
```

- Change Author Identity (All repositories):

```
$ git config --global user.name "AmirKooshky"
```

```
$ git config --global user.email "kooshkya@gmail.com"
```

```
PS C:\Users\Logan\Desktop\College\Web\Rock-Paper_Scissors>
```

```
* History restored
```

```
commit 8ab78f1a79b1beaebbc1f6636604106f9017f749 (HEAD -> gui, origin/gui)
```

```
Author: Amir Kooshky <kooshkya@gmail.com>
```

```
Date: Fri Feb 10 16:16:58 2023 +0330
```

```
gui: further style weapon buttons
```

```
commit b828a6d80984725fd6fae49897e7f3ec4562e153
```


```
Author: Amir Kooshky <kooshkya@gmail.com>
```

```
Date: Fri Feb 10 16:13:49 2023 +0330
```

If you don't have git installed, refer to [this link](#)

Remove the --global option to isolate changes to the current repository

- **Change default branch name:**
- `$ git config --global init.defaultBranch main`

An orange cloud-like shape with a dark orange outline, containing white text.

Will be come back
to this once we have
covered branches

- **Check all of your git config**
- `$ git config --list`
- `$ git config <key name>` e.g. `$git config user.name`

What is a repository (repo)?

- Turn the current folder into a git repo by creating a hidden `.git` folder inside it.

```
$ git init
```

- Get the overall status of the git repository (to be explained later)

```
$ git status
```

```
$ git status --short : The short version
```

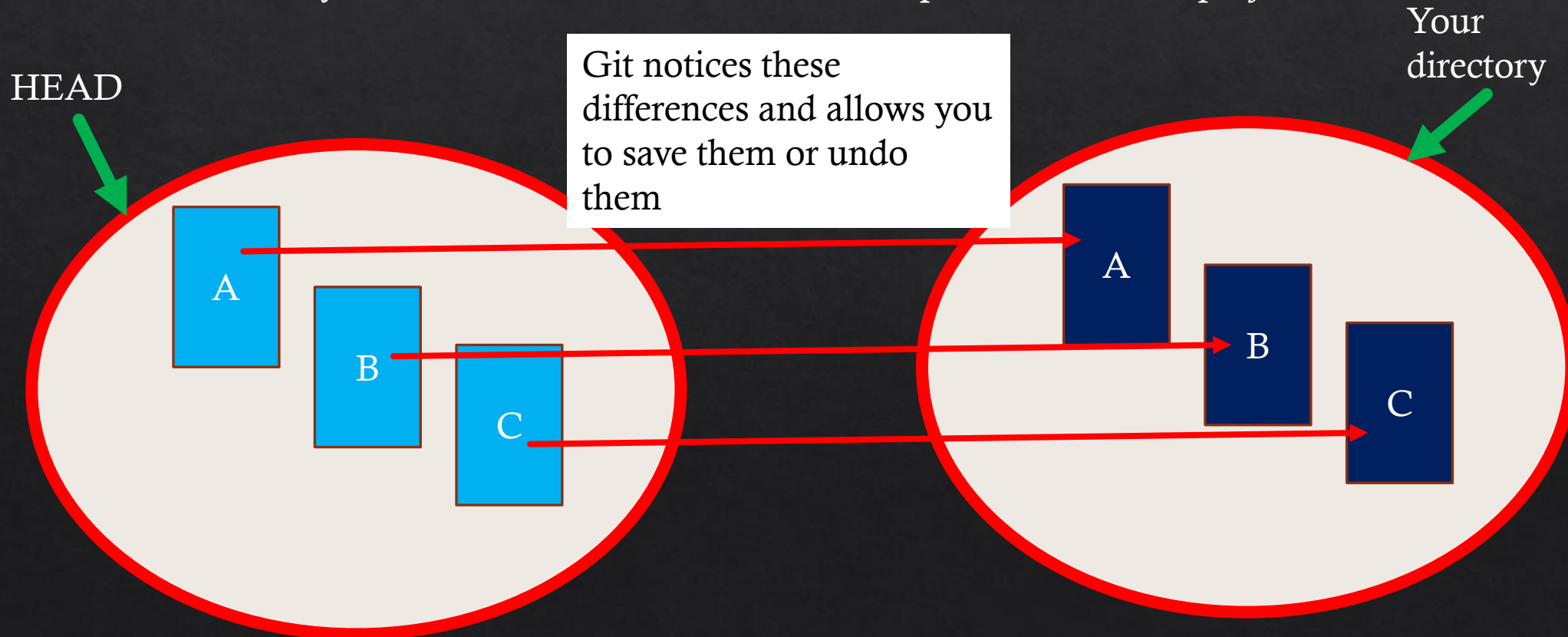
```
kooshkya@DESKTOP-Q0U5DEF:~$ mkdir mine
kooshkya@DESKTOP-Q0U5DEF:~$ cd mine
kooshkya@DESKTOP-Q0U5DEF:~/mine$ git status
fatal: not a git repository (or any of the parent directories): .git
kooshkya@DESKTOP-Q0U5DEF:~/mine$ git init
Initialized empty Git repository in /home/kooshkya/mine/.git/
kooshkya@DESKTOP-Q0U5DEF:~/mine$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
kooshkya@DESKTOP-Q0U5DEF:~/mine$ ls --all
.  ..  .git
```

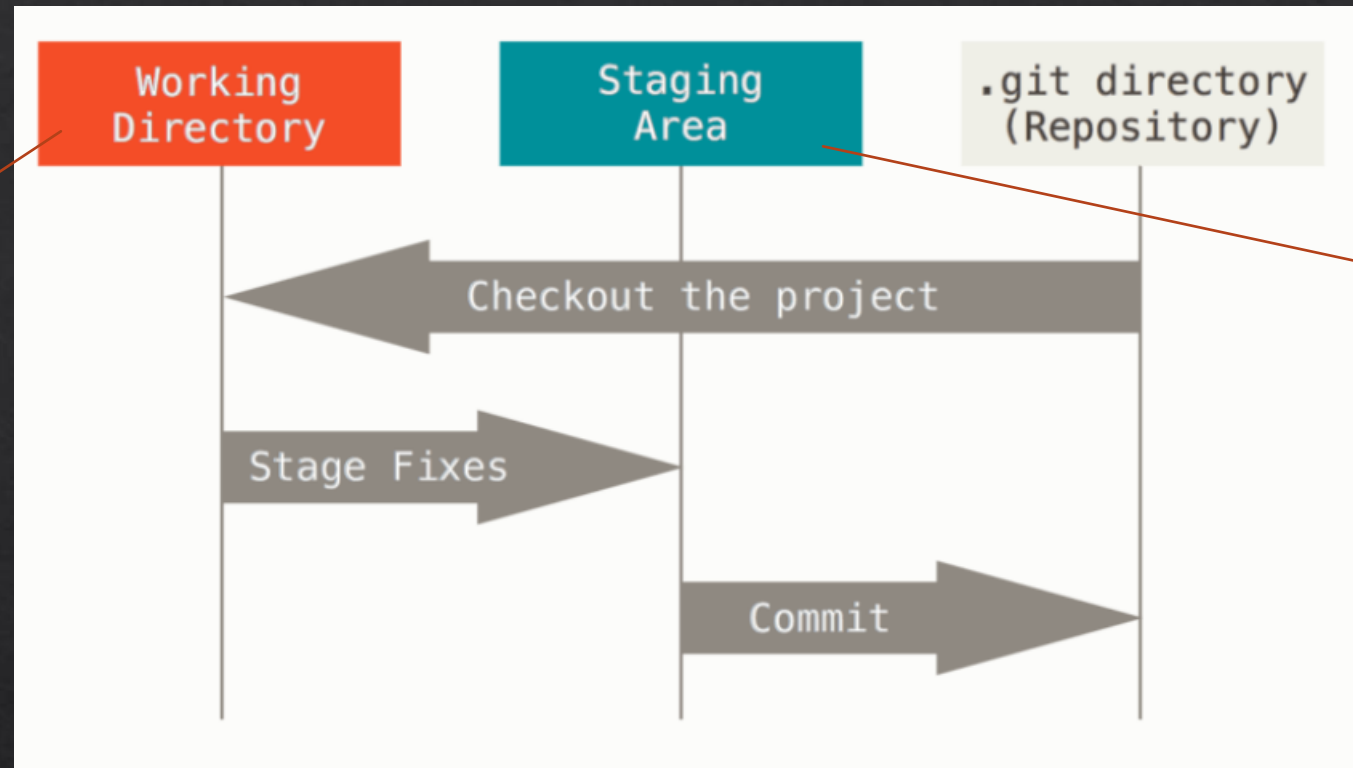

Simplified explanations of important Git components

In order to save the repo's state at a certain moment (equivalent to copying it into another directory so you can bring it back if you break it), we make **commits**. So each repo consists of many commits. There is a pointer called **HEAD** which refers to a certain commit. Git regards this commit as the “last save”. So it calculates your files' changes relative to their “image” in this commit. The equivalent of copying into another directory would be that HEAD refers to the copied folder of the project.



The Three States Of Git Files

1. **Unchanged (committed)**
2. **Untracked** (All the other types are technically “tracked”)
3. **Modified**
4. **Deleted**



Also called the “**Git Index**” or simply “**Index**”

Commands To Move Files Between States

```
$ git add <filename>           : stage modified/untracked file
$ git add .                     : stage all modified/untracked files
$ git add --all                 : stage modified/untracked file
$ git commit -m "commit message" : commit the entire staging area
$ git commit                    : commit the entire staging area
    but write "extended" commit message in editor.
$ git commit -a -m "commit message" : stage and commit the entire staging area (does
    not track untracked files
```

A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes and a soft, painterly appearance.

Project 0

- **Checking commit history**

```
$ git log
```

- Use Enter Key to scroll down when the content does not fit in the terminal.
- Use Q key to exit the scroll area.

- **Checking commit history (short version)**

```
$ git log --oneline
```

A large, stylized orange cloud shape with a thin dark outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 1

- **Change the editor git commit takes commit messages in**

Nano:

```
$ git config --global core.editor "nano"
```

VS Code:

```
$ git config --global core.editor "code --wait"
```

- **Check What git's current core editor is:**

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

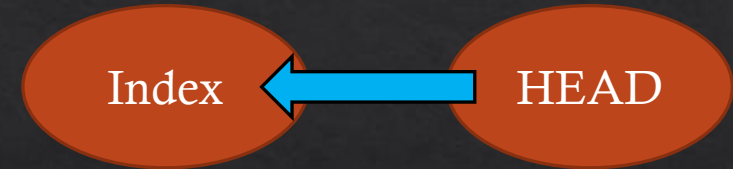
Undoing things in git

Restoring working directory changes

- Use restore to do all types of restorations

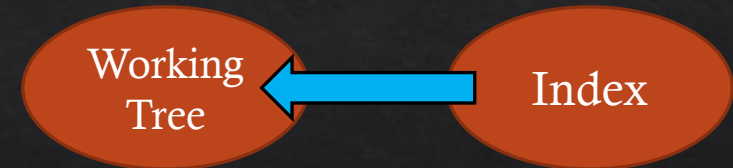
Unstage a staged file:

```
$ git restore --staged <filename>
```



Restore a modified/deleted file (restores it from the stage, NOT from the last commit)

```
$ git restore <filename>
```

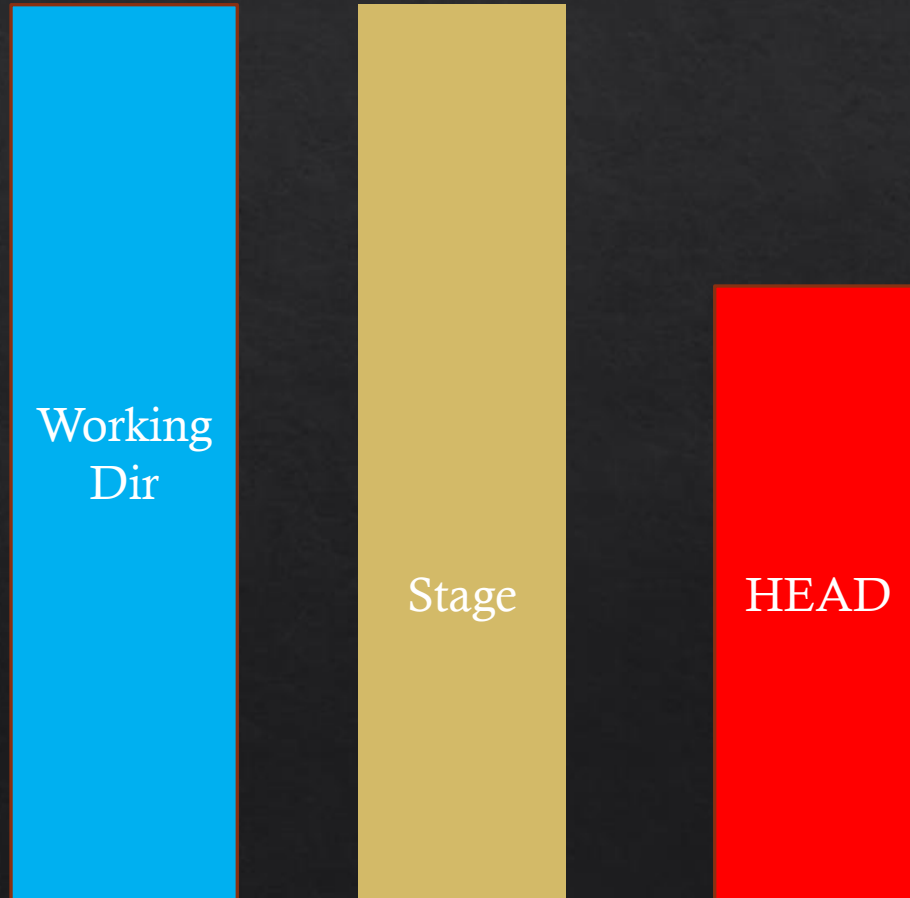


IDEs like VSCode and IntelliJ have git extensions that simplify the restoration process

Undoing things in git

Restoring working directory changes

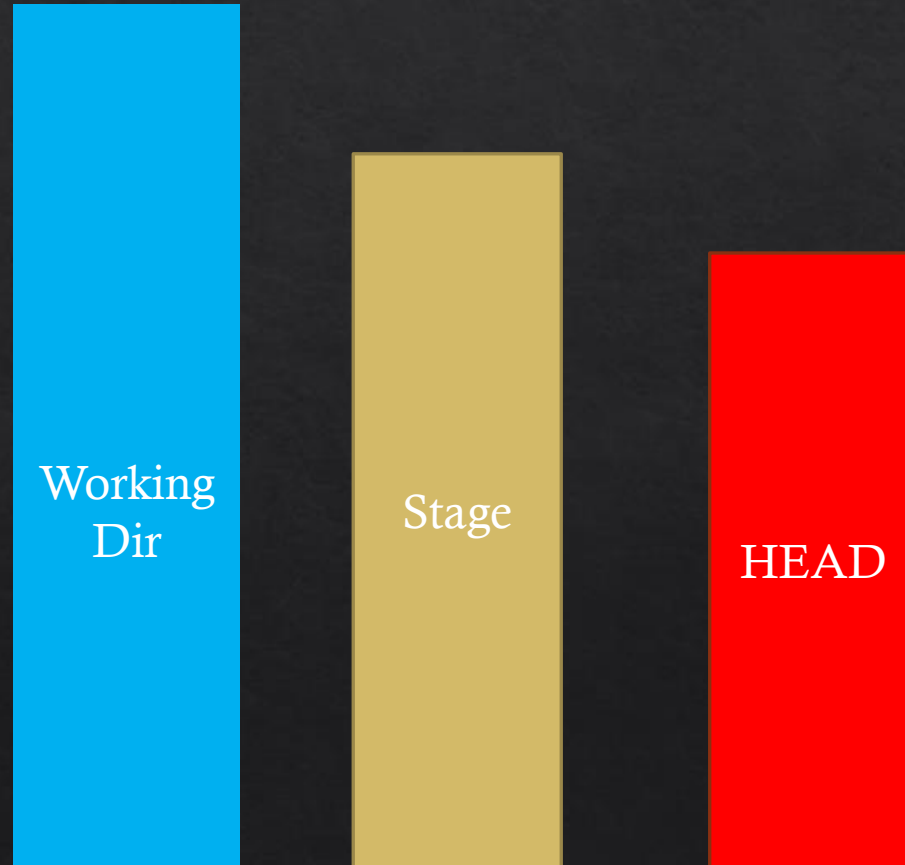
```
$ git restore --staged <filename>
```



Undoing things in git

Restoring working directory changes

```
$ git restore <filename>
```



Undoing things in git

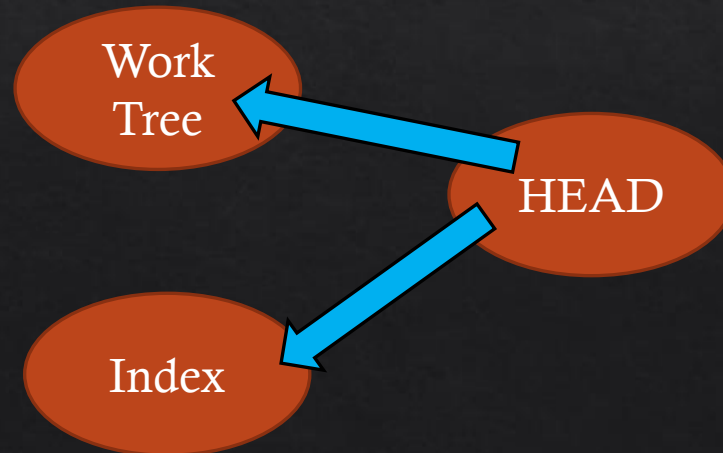
Restoring working directory changes

- Use restore to do all types of restorations

Do both of the previous ones:

```
$ git restore --staged --worktree <filename>
```

Restore docs



Undoing things in git

Undoing Commits

- Each commit is linked to the previous, **parent** commit. The differences made to the files between the two commits can easily be calculated by Git.
- Use **revert** to undo the changes made in a specific commit

```
$ git revert <ref>
```

<ref> can be a commit hash (read from git log), or other things which we do not yet cover.

Note that git revert undoes the changes in exactly the commit specified and does not work if you want to undo several commits at once

An example will be covered in Project 2

Undoing things in git

Undoing Commits (2)

- Use **reset** to take the **HEAD** back to a previous commit

```
$ git reset <commit hash>
```

Defaults to `--mixed`: index changed to match the destination commit but worktree is unchanged

```
$ git reset --soft <commit hash>
```

Neither index nor worktree is changed

```
$ git reset --hard <commit hash>
```

Index and worktree are changed

Refer to [this link](#) for more details

This is an example of a command that meddles with the commit history. Use of these commands can become troublesome. Try to avoid them and instead use checkouts and branching to undo changes but not lose any data (This method will be covered later)

An example will be covered in Project 2

Undoing things in git

Modifying the last commit

- **Use amend to modify the last commit**

```
$ git commit --amend -m "new commit message"
```

Commits the current index (anything you have staged) and renews message

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

Commit the current index but don't prompt a new message

This is another example of a command that meddles with the commit history. Use of these commands can become troublesome. Try to avoid them and instead use checkouts and branching to undo changes but not lose any data (This method will be covered later)

An example will be covered in Project 2

A large, stylized orange cloud shape with a thin dark outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 2

What is a tag?

Git Tags

Tag Types:

Annotated: hold information about the author, date created, message, etc.

Lightweight: Only holds info about the commit it's pointing to.

- **Creating and deleting Tags:**

```
$ git tag <tag name>
```

Creates a lightweight tag on the most recent commit (HEAD) with the specified name

```
$ git tag <tag name> <commit hash>
```

Creates a lightweight tag on any commit (specified by hash) with the specified name

```
$ git tag -a <tag name> -m "tag message"
```

Creates an annotated tag on the most recent commit (HEAD) with the specified message and name.

```
$ git tag -a <tag name> -m "tag message" <commit hash>
```

Creates an annotated tag on the hash-specified commit with the specified message and name.

```
$ git tag -d <tag name>
```

Deletes a tag locally.

- **Reading tags**

```
$ git tag
```

Lists the names of all tags

```
$ git show <tag name>
```

Show all the data the tag holds (including commit, message, author, date, etc.)

- **Pushing tags : git push origin does NOT push tags**

```
$ git push origin <tag name>
```

Push the specified tag

```
$ git push origin --tags
```

Push all tags

```
$ git push origin --delete <tag name>
```

Delete tag from origin (git push does not sync tags)

- **Checking out tags**

```
$ git checkout <tag name>
```

Check out the tag's commit

Git Ignore

- **Creating .gitignore**

```
$ touch .gitignore
```

Create the .gitignore file

You can use the wildcard * in your .gitignore directives:

*.txt => ignores all txt files (in ANY directory)

You can ignore entire directories:

hello/ => ignores the contents of hello/ directory ANYWHERE in the repo

/hello/ => ignores the contents of hello/ directory ONLY in the root

To prevent .gitignore's recursive behavior, you can create .gitignores INSIDE of directories. The rules specified in those will not apply to parent directories.

List of complete gitignore rules can be found [here](#)

A large, stylized orange cloud shape with a thin dark outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 3

Branches, an introduction

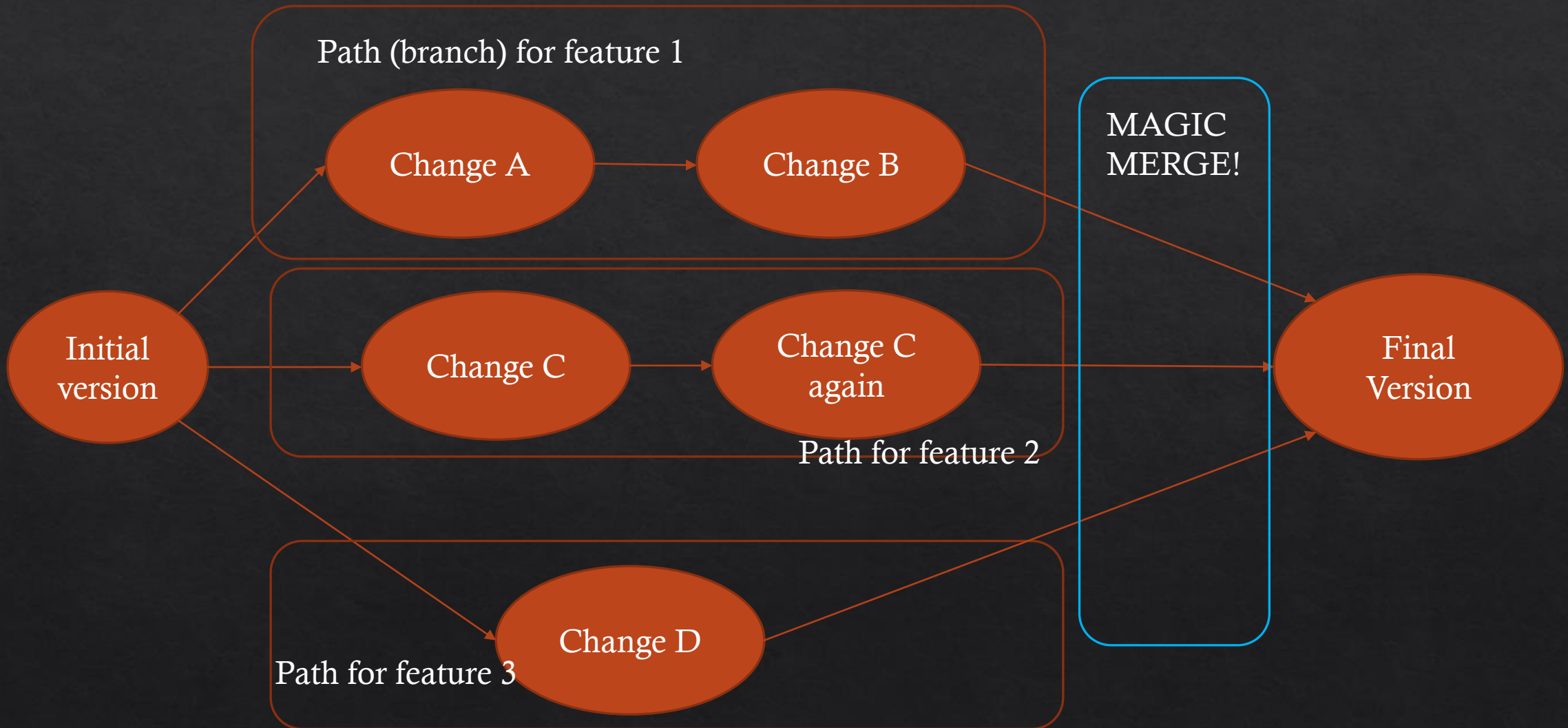
Branches are an incredibly powerful tool in Git. Until now, we have only had a linear development process, where commits are strung along in a line, each one building on the one before it.

But what if you need to collaborate with other people on a project? Say you need to split tasks and have each person work on a specific feature. By the time everyone is done, you will have 3 versions of the project, each including one of the features. You would have the extra (rather tedious) task of making those 3 projects into a single one that incorporates all three features.

The same issue arises even if there is no collaboration, but you wish to work on 3 different features at the same time without making all the changes to a single version of the project. This is a common case because you will probably want to keep the 3 development processes separate so you will have a less troublesome debug phase (having 3 unfinished features in a single project will prove to be quite a headache if bugs arise down the line.) Git branches offer a solution to the above problems, and more! Essentially, branches allow you to break your development path into separate paths (i.e., branches.) But what makes them practical is Git's ability to later merge those branches automatically, converging all the changes you made on the three different paths into a single commit, thus incorporating all the implemented features into a single, final commit.

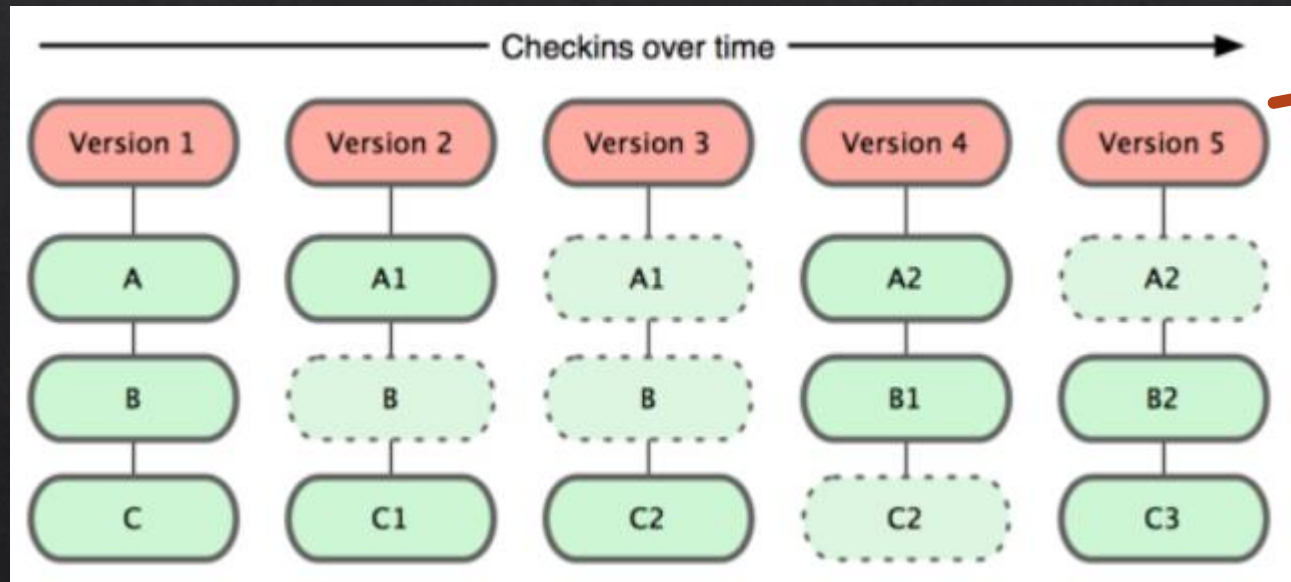
We will present the schematics of how branches typically work on the next slide, and then will delve into the inner workings and technicalities of branches. You absolutely need to learn these basics if you want to efficiently use Git (and not accidentally cause disastrous incidents with Git.)

Branches, an introduction



Git Commits, Revisited

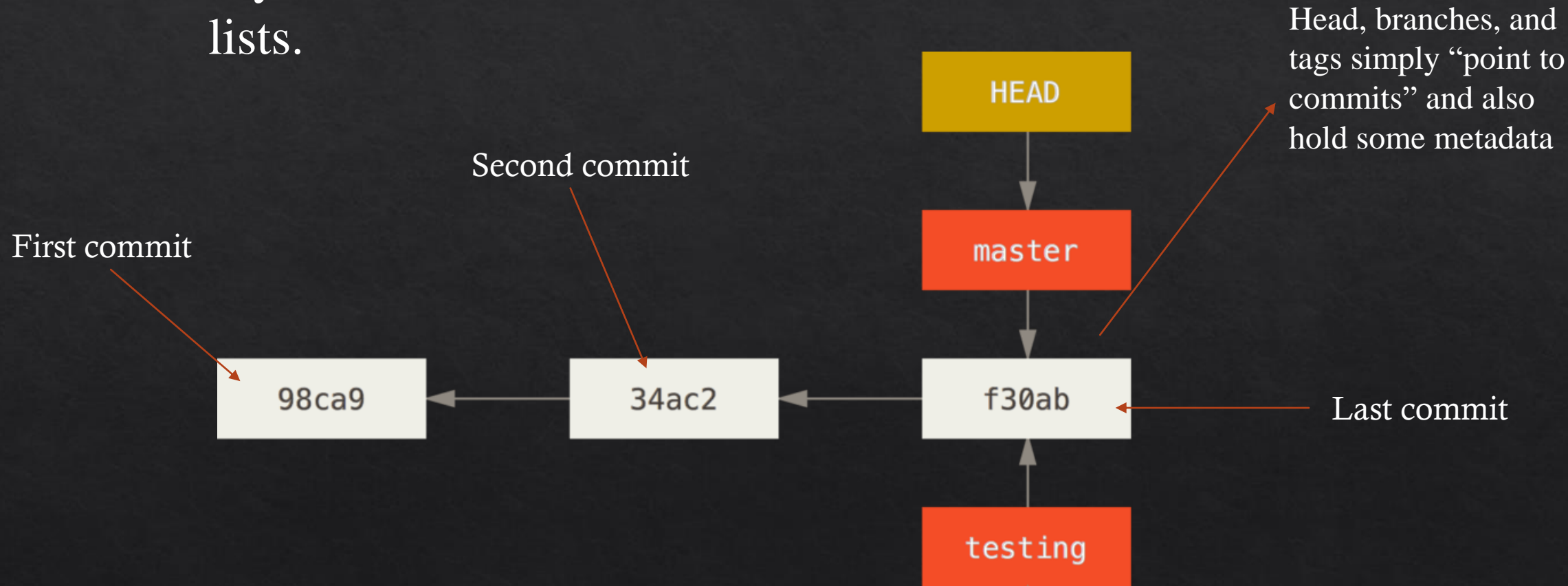
Git Stores “Snapshots” of the files in each commit



But it's also easy for Git to see the changes between any two commits

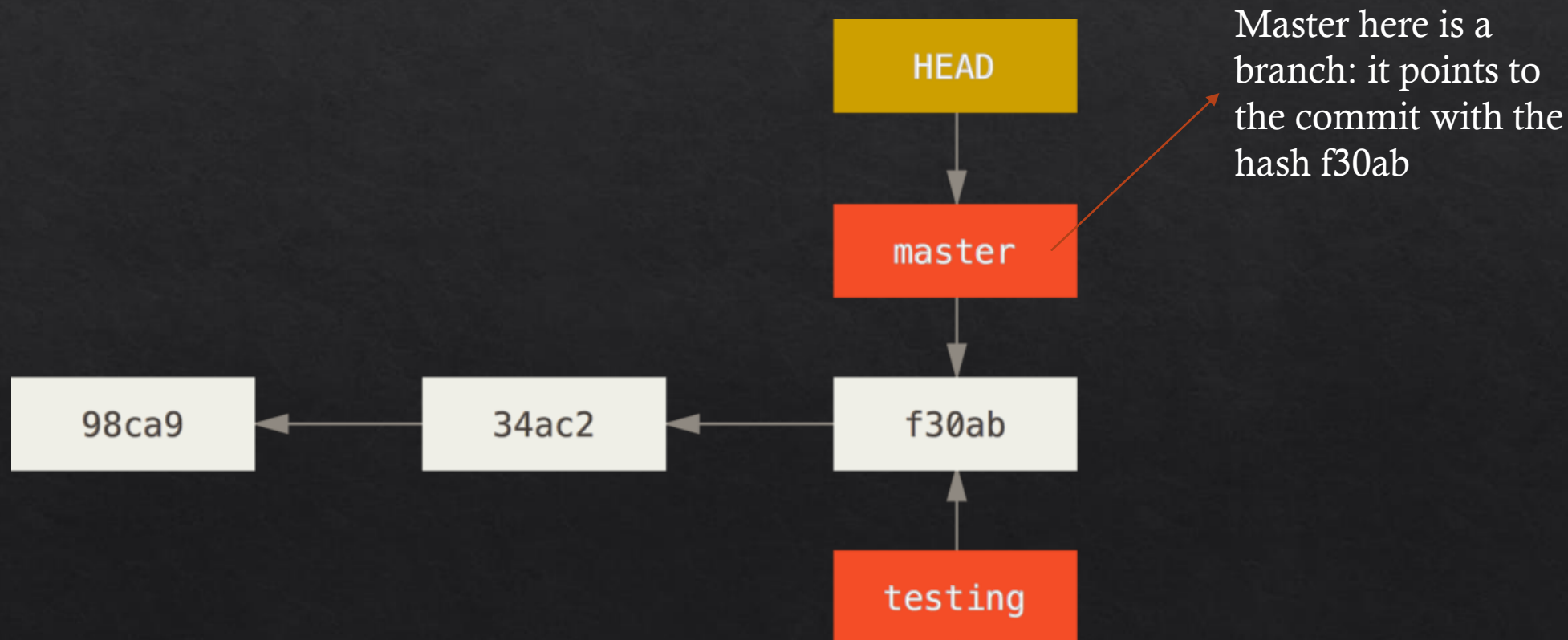
Git stores commits as a DAG (a directed acyclic graph) in which each commit “points” to its parent commit(s). Its structure is very similar to linked lists.

Git branches, and tags (and in specific cases HEAD) are references (pointers) to certain commits in the history



Branches

As we have already mentioned, branches are simply “pointers to commits”. When you make the first commit in your repository, a **default branch**, (names master or main) is automatically created and points to the commit you made.

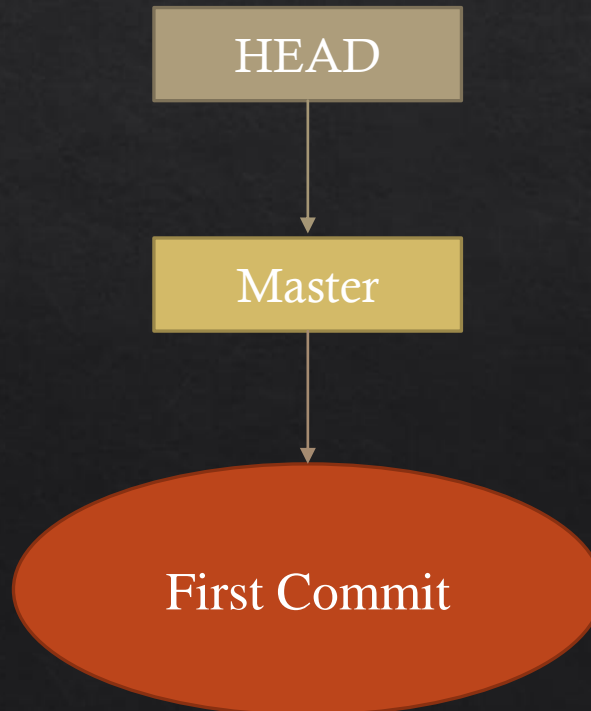


The HEAD

The head is yet another pointer! **It always points (either directly or indirectly) at a commit.** It specifies which commit/branch you are working on.

When you make the first commit in your repo, 2 things happen:

1. The default branch (called master or main) is created and points to that commit
2. The HEAD is updated to point to the default branch (it's a pointer to a pointer, in a sense). We can also say that the head is pointing (indirectly) at that commit



The HEAD (part 2)

A note on the staging area: the staging area does NOT change when you make a commit. What changes is what the HEAD points to. When you commit, a new commit is created that matches the content of the index. The head is then updated to point to this new commit. So when the content of HEAD is compared with the index, no differences are detected, and git status does not show any “uncommitted” changes.

So what we have been doing so far by committing has been:

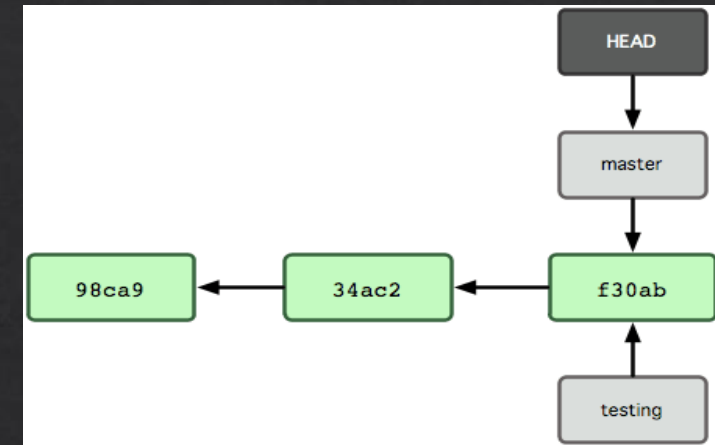
1. Creating new commits that match the current state of the index
2. Updating the master/main branch to point to them, thus causing the HEAD to indirectly point to them.

When the head points to a commit **C**, it means that the “last commit” that the index is compared against the commit **C**. If the index differs from this commit in any file, that difference is counted as an uncommitted change and will be committed if you use git commit

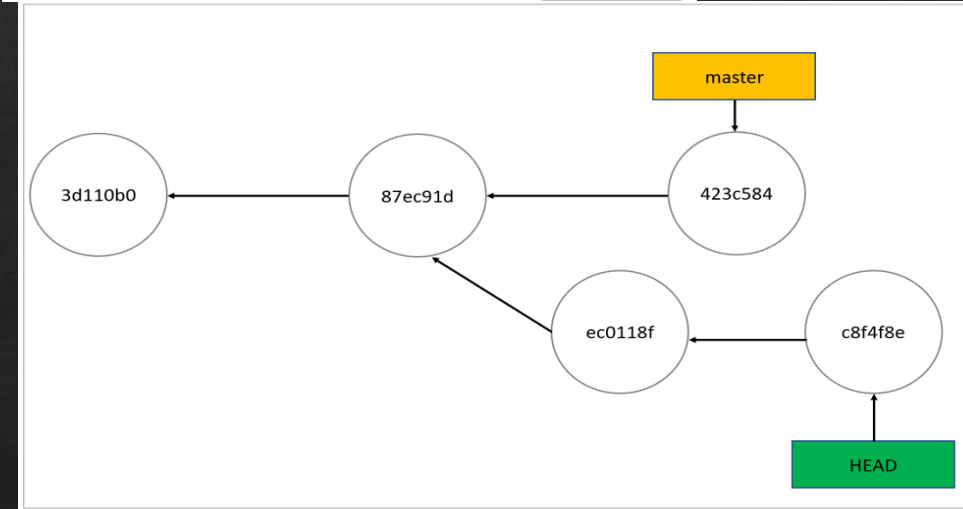
The HEAD (part 3)

The HEAD has two states:
Depending on whether it points to a commit or to a branch

Normal State: When head is pointing to a branch (for example master)



Detached State: When head is pointing to a commit, NOT a branch



If you make a new commit using the git commit instruction while in the **detached head state**, the HEAD will be updated to point to the new commit. If HEAD was pointing to a branch (meaning we were in **Normal State**), the branch will be updated to point to the new commit, and HEAD will keep pointing at the branch.

The HEAD (part 4)

If you are in **detached head state**, if you commit changes, head will be updated to point to the new commit, but no branch will move to point to that commit.

If you are in **normal head state**, HEAD and the branch it is pointing to will be updated to point to the new commit (the branch will point to the commit and the head will still point to the branch). This is called “normal” because you will always have access to that commit through your branch.

- **Updating HEAD to point to a new branch/commit**

```
$ git checkout <branch name>
```

```
$ git checkout <commit hash>
```

Git checkout updates the working directory AND staging area to the checked-out commit's state.

Note that in order to checkout to a commit/branch, you are **USUALLY** not allowed to have any unstaged or uncommitted changes. In short, **git status** should not return any unstaged or staged changes. Just commit your changes and go back to that branch.

Branch Commands

```
$ git branch
```

List all branches. Puts a * beside the branch that HEAD currently points to. If HEAD is detached, will print the commit hash that HEAD points to.

```
$ git branch <branch name>
```

Creates a new branch with the given name to point to whatever commit HEAD is pointing to.

```
$ git checkout -b <branch name> <commit hash>
```

Checks out the given commit and also creates a new branch with the given name to point to it (the HEAD will point to the newly created branch, NOT the commit, so it will not be detached)

```
$ git branch -d <branch name>
```

Deletes the given branch name.

```
$ git log --all
```

Show commits and branches and tags for ALL branches, not just the current one's

A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes and a soft, painterly appearance.

Project 4

What are branches for?

- **Making features:** When you want to make changes to the code but still have a backup of the current state of your repo (in case you break things!), you can create a new branch on the commit you are at right now and make the changes you want to make and commit them. As we have seen, the new commits will create their separate “branch” in the git DAG and the commit “path” of our main “branch” will be separate. If at any point we actually “break stuff” and wish to go back to before we had made our changes, we can simply checkout the branch we were originally on and revert any changes we made. But if we don’t break things and actually want to apply the changes we have made on the new branch to our main/master branch, there is a mechanism called “git merge” that we will cover later that allows us to do that.

What are branches for?

- **Better understanding reset:** we have introduced used git reset [here](#). Git reset is a simple git checkout with the difference that if HEAD is in normal state, the HEAD won't be updated to point to the specified commit, but the branch that HEAD is pointing to will be updated to point to the specified commit



A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud is composed of several overlapping circles, giving it a fluffy, organic appearance.

Project 5

Merging

Suppose you are on the master branch. You wish to add a feature so you create a new branch called **feature** and check it out. You make a few commits and are satisfied with the result so you want to apply the changes to your main branch. The git instruction to do this is “git merge”.

```
$ git merge <branch name>
```

Merges <branch name> “**into**” whatever branch HEAD is pointing to.

DON'T EXPECT TO FULLY
UNDERSTAND THIS
COMMAND JUST YET! THIS
IS ONLY A BRIEF SUMMARY
OF WHAT IT DOES

Merging (part 2)

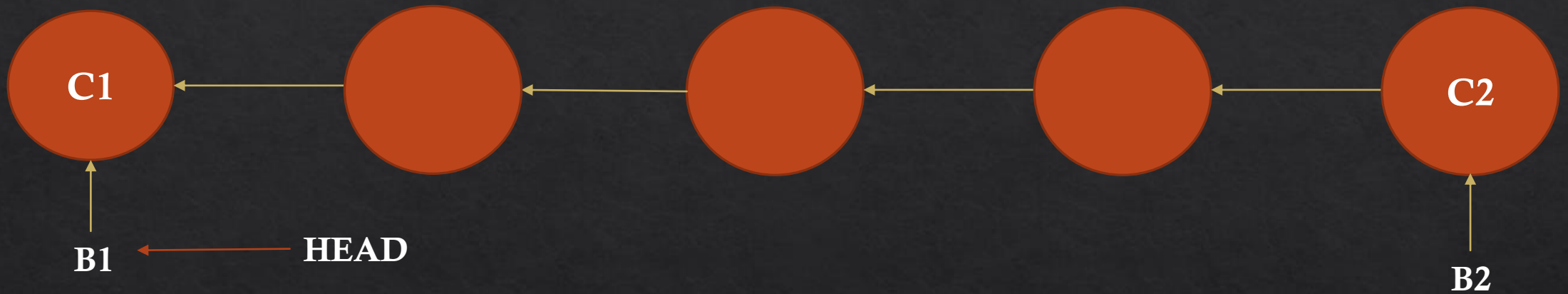
There are 2 types of merging:

1. Fast-Forward Merge
2. Three-Headed Merge

We will first consider the simpler case: fast-forward merge

Merging (part 3)

If HEAD is on branch **B1** pointing to commit **C1** and there is a branch **B2** that points to commit **C2** such that **C1** is an ancestor of **C2** (meaning **C1** was turned into **C2** through a series of commits):



Earliest Commit

Most Recent Commit

Then calling the command

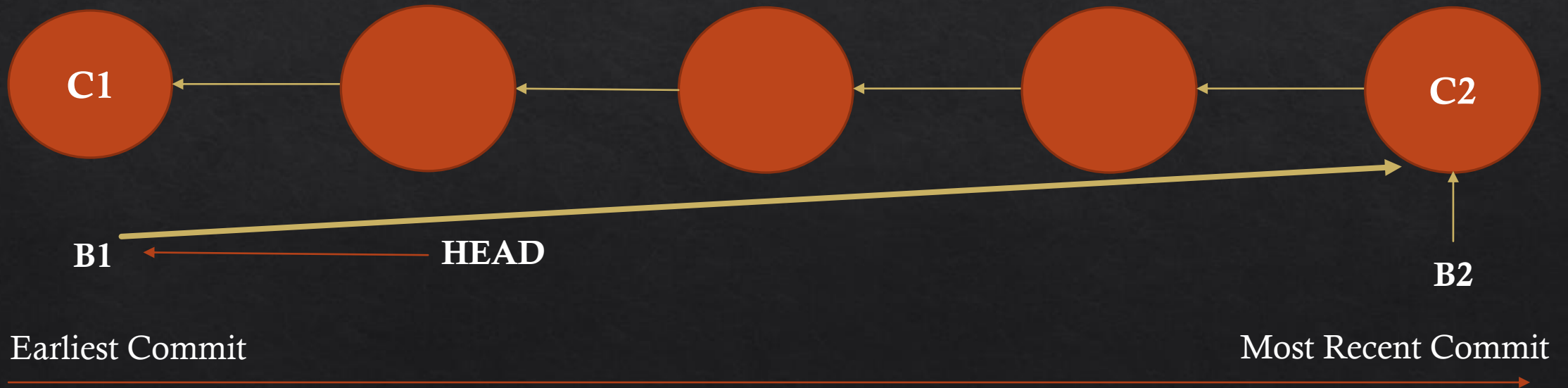
```
$ git merge B2
```

Will result in a **fast-forward merge**. In this type of merge, the branch that the head points to will simply be updated to point to the same commit as the branch that has been merged into it (in this case **B2**) See the result on the next page

Merging (part 4)

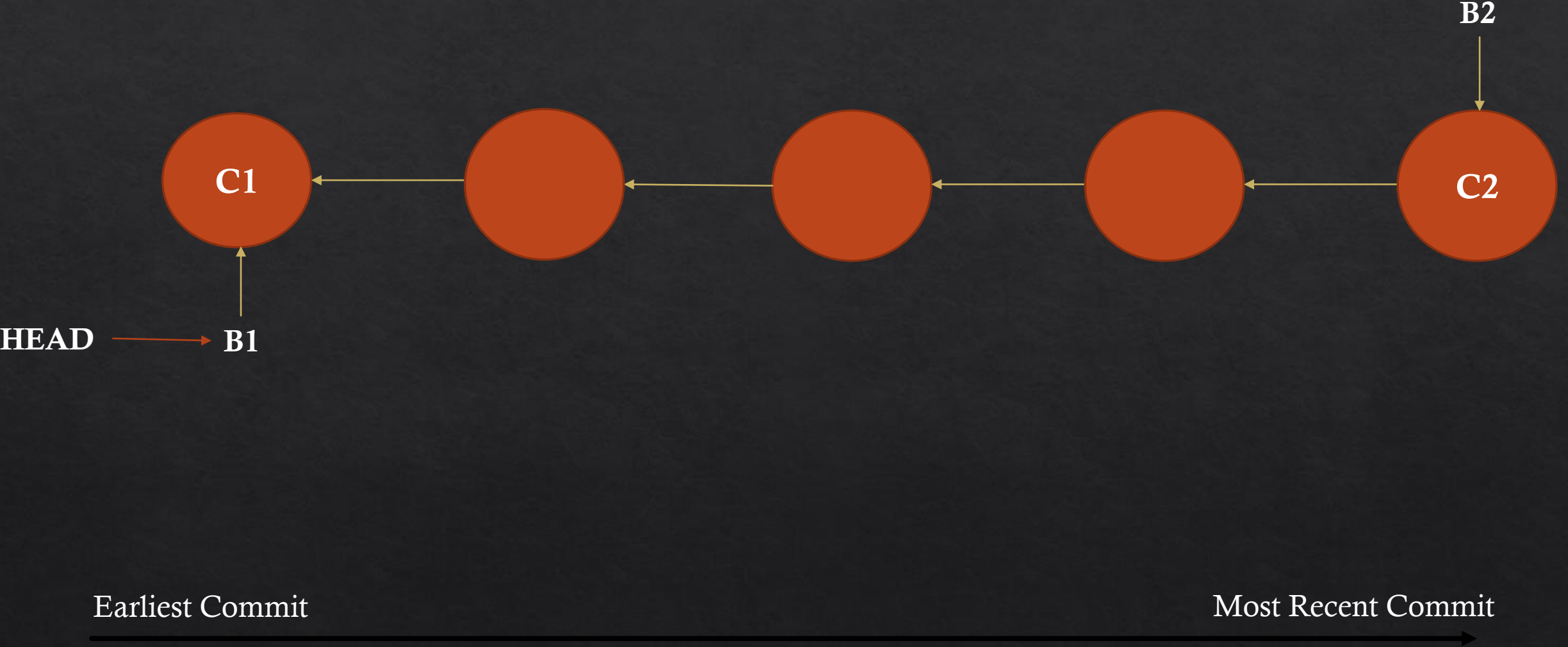
Result of calling the below command in the previous page

```
$ git merge B2
```



Merging (part 5)

Fast Forwarding



A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes and a soft, painterly appearance.

Project 5.1.

Merging (part 6)

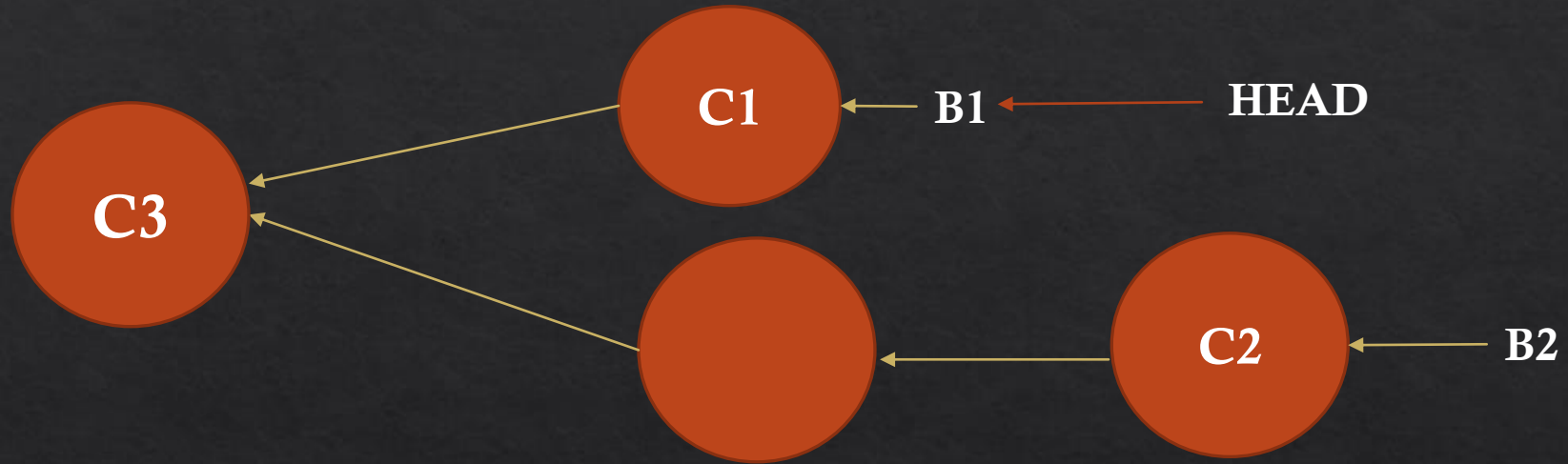
There are 2 types of merging:

1. **Fast-Forward Merge**
2. **Three-Headed Merge**

We will now consider the second case: **Three-Headed Merge**

Merging (part 7)

If HEAD is on branch B1 pointing to commit C1 and there is a branch B2 that points to commit C2 such that C1 is NOT an ancestor of C2 (meaning C1 and C2 had a most recent common ancestor called C3):



Earliest Commit

Most Recent Commit

Then calling the command

```
$ git merge B2
```

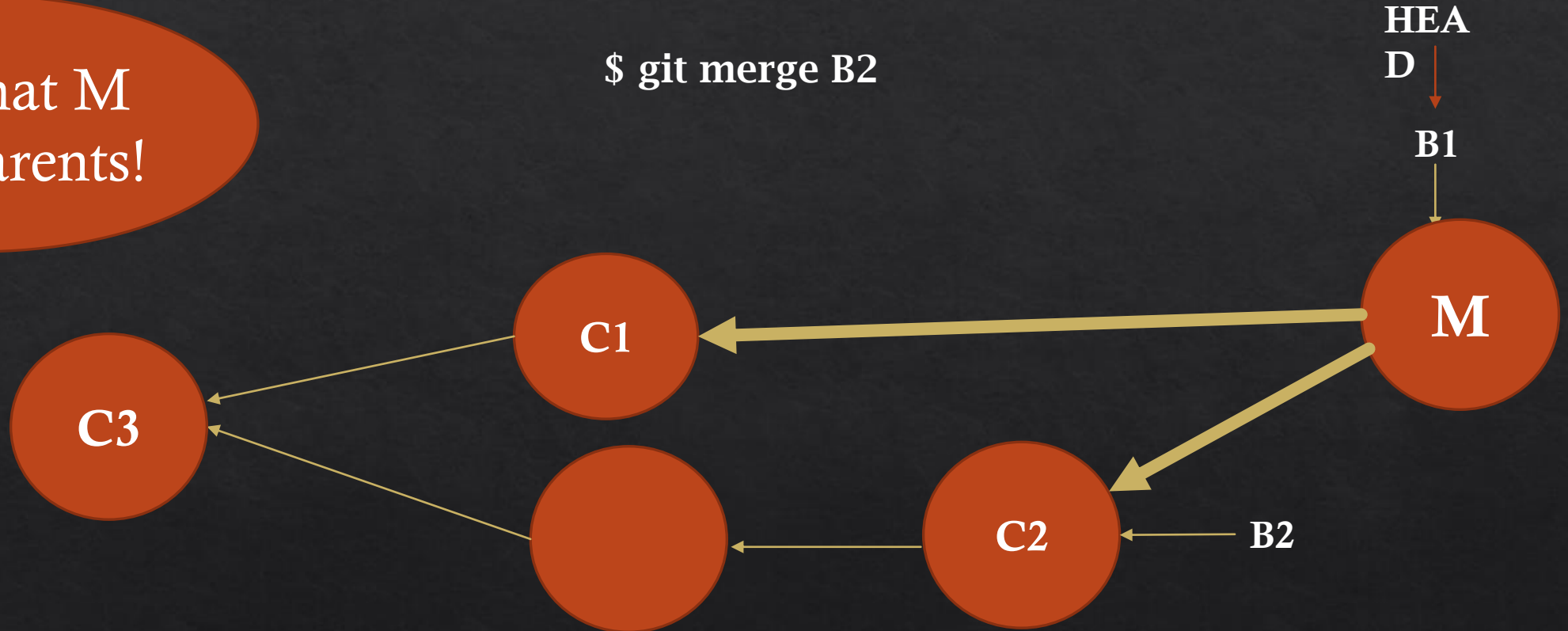
Will result in a **Three-Headed Merge**. In this type of merge, a new commit (M) will be created on the branch that HEAD is pointing to and all changes from C3 to C1 and those from C3 to C2 will be applied to it. (see result on the next page)

Merging (part 8)

Result of calling the below command in the previous page

`$ git merge B2`

Note that M
has 2 parents!



Earliest Commit

Most Recent Commit

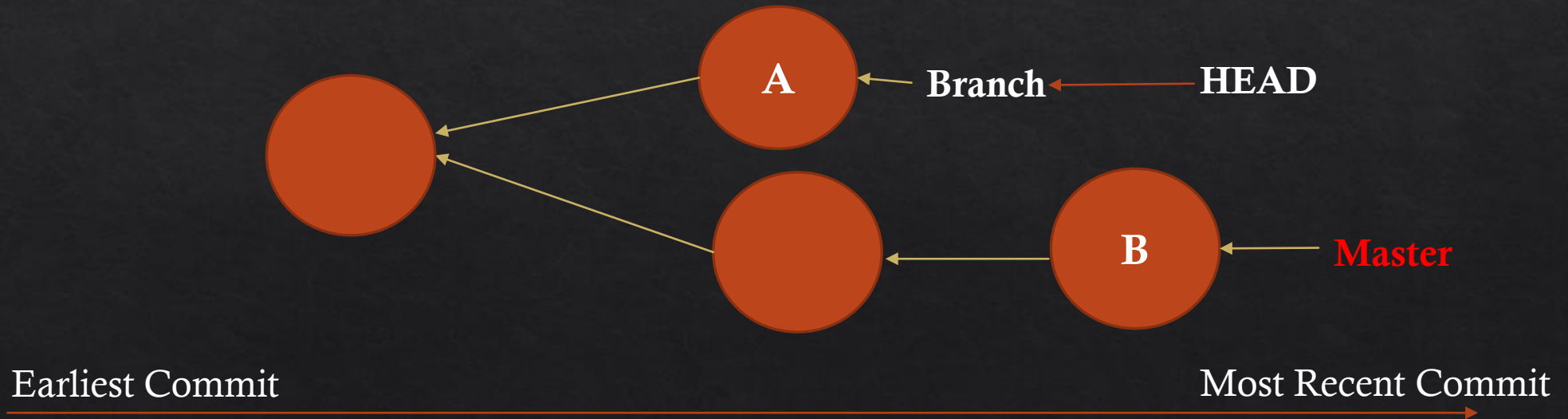
A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 5.2.

Merging (part 9)

Merge Conflicts

Suppose you have created a branch and have made a few commits to it. Meanwhile, the master branch has also advanced by a few commits (maybe another developer committed to master or you merged another branch into it.) Your repo will look like this:



Merging (part 10)

Merge Conflicts Continued

Now Also suppose that the commits on the master branch have edited line 2 of a certain file called file1.txt to be “hello master!” and your commits on **Branch** have also edited line 2 of file1.txt to be “hello branch!”. When you try to merge **Branch** into **master**, git does not know which version of the line to put in the resulting “merged” commit, so it uses the **Merge Conflict Mechanism** to let you decide which changes to keep and which ones to discard.

Git uses special notation to point out **conflicting lines in files**. If you open these in a plain text editor, you will see the exact notation. IDE’s like VSCode and IntelliJ, however, don’t let you see these raw notations and style them with colors and buttons to be more readable and easier to work with.

The process of removing the notations that git has intalled in your files and applying your desired changes is called “**conflict resolution**”. After resolving the conflicts, you should use git commit -m “custom commit message” or simply git commit to make the merge commit with the default message.

If you make a mistake in the conflict resolution process and want to undo all changes you made in the process of merging, use:

```
$ git merge --abort
```

Merging (part 11)

Merge Conflicts Notations

```
public static void main(String[] args) {  
<<<<<<< HEAD  
    System.out.println("feature1");  
=====  
    System.out.println("feature2");  
>>>>>>> feature2  
}
```

Simple git notation for conflicted lines:

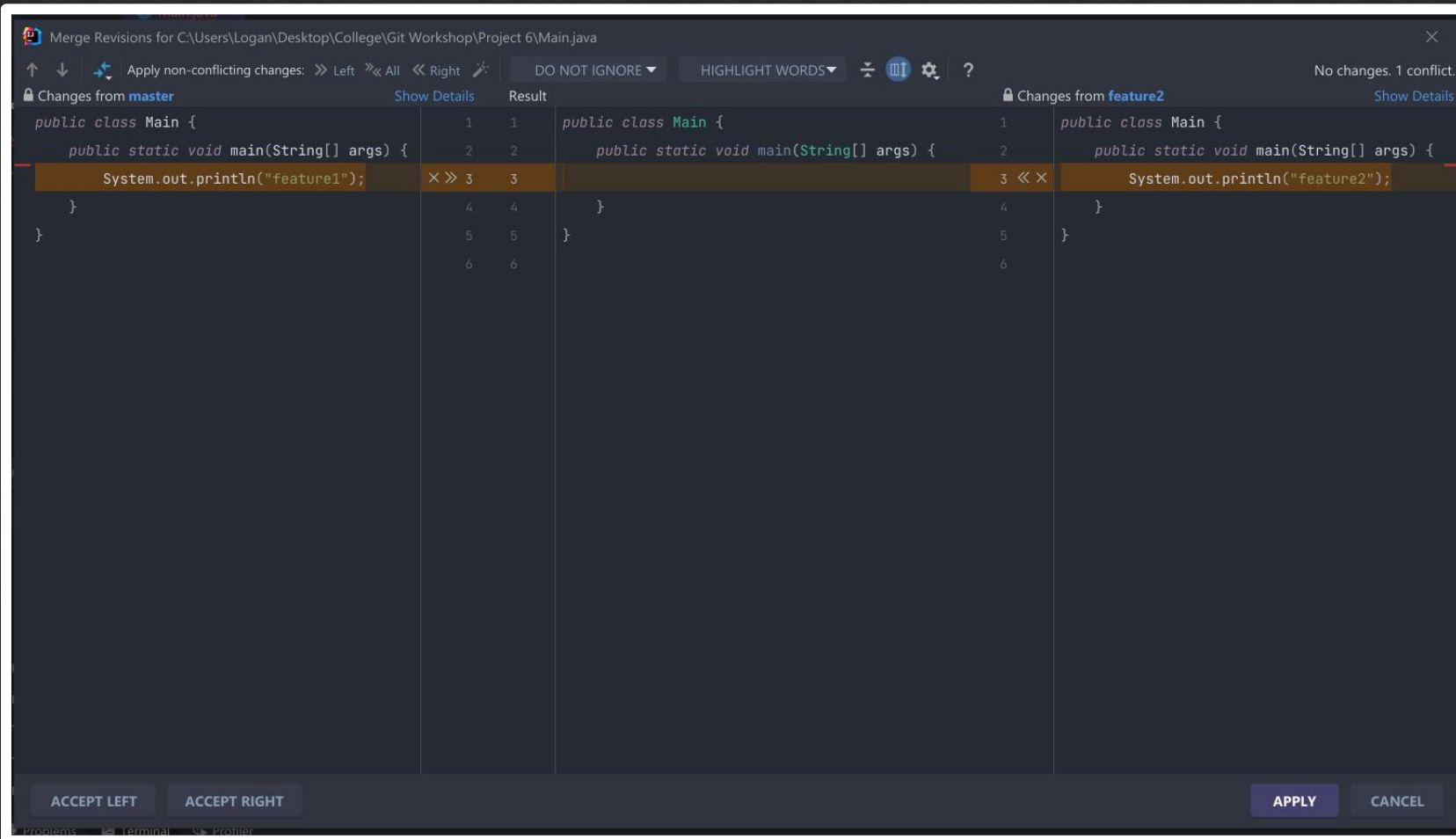
<<<<HEAD signifies the change that the branch that HEAD is currently on is wishing to make.
===== splits the two changes. The line(s) between ===== and >>>>>feature show the changes that the feature branch is wishing to make.
You decide how to manage these changes: you make the changes you want and remove the <<<< and ===== and >>>>> notations

```
2 | public static void main(String[] args) {  
   | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
3 | ✓ <<<<<<< HEAD (Current Change)  
4 |   System.out.println("feature1");  
5 | ✓ =====  
6 |   System.out.println("feature2");  
7 | ✓ >>>>>>> feature2 (Incoming Change)  
8 |   }  
9 | }
```

VSCode notation for conflicted lines: Exactly like simple git notation but has added buttons and coloring to make it easier for the user to resolve the changes

Merging (part 12)

Merge Conflicts Notations



IntelliJ conflict resolution tab:
Lots of coloring and options.
Make sure you familiarize
yourself with this feature of
IntelliJ if it's your editor of
choice.

A large, stylized orange cloud shape with a thin black outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 6

General Conflicts

Note that merging is not the only place you will encounter conflicts. We will see later on that conflicts arise when we are pulling from remotes as well.

Merge conflicts are not based on line numbers: Git is pretty smart about comparing files and changes. It is incorrect to think that the only way git compares to see if two branches have a merge conflict is by comparing their changes to identical line numbers in files. The Project on the next page will illuminate this concept.

A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 6.1.

GitHub

You have already worked with GitHub! It is simply a website to “**host**” your code. It stores your repository so you can download (clone) it to any device. It allows you to tweak the stored repository from any of your devices. It also offers countless other features which we will not have time to cover.

Remember Git is a **distributed Version Control System**, so its structure allows a single repository to be on several devices. It offers features that lets these “clones” of the repository stay in sync and communicate. A special case of this relationship is between the repository on your local device and that on GitHub (which can be thought of as just another device out there that stores the repo!)

In the next part of the tutorial, we will learn to use these features!

Cloning

Cloning means downloading a repository from GitHub onto your local device. When a repository is cloned, all of its commits, branches, tags, etc. will be downloaded onto your device. Note that if you download the repo from GitHub as a .zip file, it will only give you the files in one of the commits, but there will be no **.git** folder. Thus, you will not have downloaded a repo, you will have actually downloaded the state of files in a certain commit in the repo. This is not suitable for someone who wants to contribute to the project!

How to clone a repo from GitHub:

```
$ git clone <clone link>
```

You will learn how to obtain the clone link in project 7

Remotes

A **git remote** is simply an object that lets your local repository communicate with the repository on another device. Think of each remote as a link to the repo on another device. For example, if you “clone” a repository from GitHub, Git automatically creates a remote called **origin** that links your version of the repo to the one that is on GitHub. You will learn how to use **origin** to change the repo on GitHub in later sections.

With each remote comes an **access right**: there are varying levels of this. Your remote may allow you to only read from the remote repository it links to. Or it may give you read/write privilege, which allows you to also modify the remote repo it links to.

Git remotes store a **fetch link** (that is used to read from the remote repo) and a **push link** (used to write changes to the remote repo)

List all git remotes:

```
$ git remote
```

List all git remotes and their fetch and pull links:

```
$ git remote -v
```

Add a git remote:

```
$ git remote add <remote name> <remote link>
```

SSH link: <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh>

Project 7

What happens after you clone?

Suppose you clone a repository. You will get all the commits and all the branches. Now you go to master and make a change and commit. What will happen to the master branch of the remote repo on GitHub? Will the changes immediately apply there, too? The answer is no. The remote repo has no way of knowing what changes you have made to the repo unless you explicitly tell it about them. So what happens is you will have your own version of the repo. If you make a commit on master, your master branch will split into 2 branches (as you will see in git graph in project 7.1): master (or local master) and origin/master. Origin/master points to the commit that master used to point to when you cloned the repo. Origin/branch-names are all your device's "memory" of the state of the remote repo when you cloned it. Branches without origin/ at the beginning of their names, however, are branches that you have modified and that are on your device. origin branches can be checked out. But your HEAD **will NOT point to the origin/branches**. checking out the origin branch will place you in **detached head state** and HEAD will only point to a commit.

If you check out one of the remote branches without putting origin/ at the beginning, git will make your local version of that branch and point HEAD to it.

The master/main branch is the only branch that will have both local and origin versions after cloning. The other branches will only have the origin version. Git will create the local version when you check out the origin version.

The origin after cloning

Just as your local repo may change after cloning, the origin repo (on GitHub) may also change. For example, someone else may commit on its master branch, or another version of the repo on someone else's device may have pushed to it (you will see later what this means). As you may guess, the changes made to origin don't immediately apply to your local repo, because your repo can't be in constant communication with the origin to see when it has changed. So, your local repo's "memory" of what the origin repo looks like may get outdated after a while. The branch positions on the origin repo may change, and the origin repo may receive new commits. There is a command you can use to "tell" your local repo to update its data about the origin repo:

```
$ git fetch <remote name, e.g.: origin>
```

This command downloads all of the remote's new commits and updates all the origin/branch positions, and creates new origin/branches when there have been new branches on the origin. Effectively, this **updates your repo's memory of the origin**. Obviously, the remote may still change after you have git fetched, so frequent git fetches are important so you know what is going on in the origin!

Note that git fetch does NOT affect any of your local branches or commits, it only ADDS information. So it's a pretty safe operation to do.

A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes and a soft, painterly appearance.

Project 7.1.

Syncing up with the origin

Suppose that you have cloned a repository and have some changes to a branch **B**. So, your repo has an origin/B and a local B. Now suppose your git fetch origin and find out that branch **B** has received some new commits and advanced, so your **B** is out of sync with the origin. What do you do? What we are looking for here is an operation similar to “git merge” that can merge all the changes made to your local **B** and to the remote **B**. This operation is called **Git Pull**:

```
$ git pull origin <branch name>
```

First executes git fetch on <branch name> (so origin/<branch name> is up to date with the origin.) Then performs git merge origin/<branch name>

Effectively merging the origin version of <branch name> into HEAD (which should be on the local version of <branch name> otherwise things get messy!)

Note that after git fetch and during merge, you can regard origin/<branch name> exactly like a local branch. Also note that just as with regular merging, a fast-forward merge or a three-headed merge can take place.

Also, as with merging, there may be **pull conflicts** which are displayed and resolved **EXACTLY** like typical merge conflicts, so no cause for worry there! (All shall be clearer after Project 7.2. !)

A large, stylized orange cloud shape with a thin dark outline, centered on a dark gray background. The cloud has multiple rounded lobes of varying sizes.

Project 7.2.

Syncing up with the origin, the other way around!

Suppose that you have cloned a repository and have some changes to a branch **B**. So, your repo has an origin/B and a local B. Now suppose you are satisfied with these changes and want to update the GitHub repo to incorporate them. In a way, you want to merge your local branch **B** into **origin/B**. The git command to do so is **Git Push**:

```
$ git push origin <branch name>
```

Checks out origin/<branch name> and git merges with local <branch name>. So, this is effectively another merge. The only difference with git pull is what is being merged into what!

Note: In order for this command to work, your origin/<branch name> HAS to be in sync with GitHub, so always git fetch before running this command.

There are no push conflicts!: If your local branch conflicts with the origin branch, git push RETURNS AN ERROR CALLED “non-fast forward”, meaning your branch conflicts with the origin’s changes on the branch. What you need to do is first pull the origin branch INTO your local branch and then push so that no conflicts occur. (refer to project 7.3.)

A large, stylized orange cloud shape with a dark orange outline, centered on a dark gray background. The cloud has multiple rounded lobes and a soft, painterly appearance.

Project 7.3.