



Git

A distributed version control system



Version control systems

- **Version control** (or **revision control**, or **source control**) is all about managing multiple versions of documents, programs, web sites, etc.
 - Almost all “real” projects use some kind of version control
 - Essential for team projects, but also very useful for individual projects
- Some well-known version control systems are CVS, Subversion, Mercurial, and Git
 - CVS and Subversion use a “central” repository; users “check out” files, work on them, and “check them in”
 - Mercurial and Git treat all repositories as equal
- Distributed systems like Mercurial and Git are newer and are gradually replacing centralized systems like CVS and Subversion



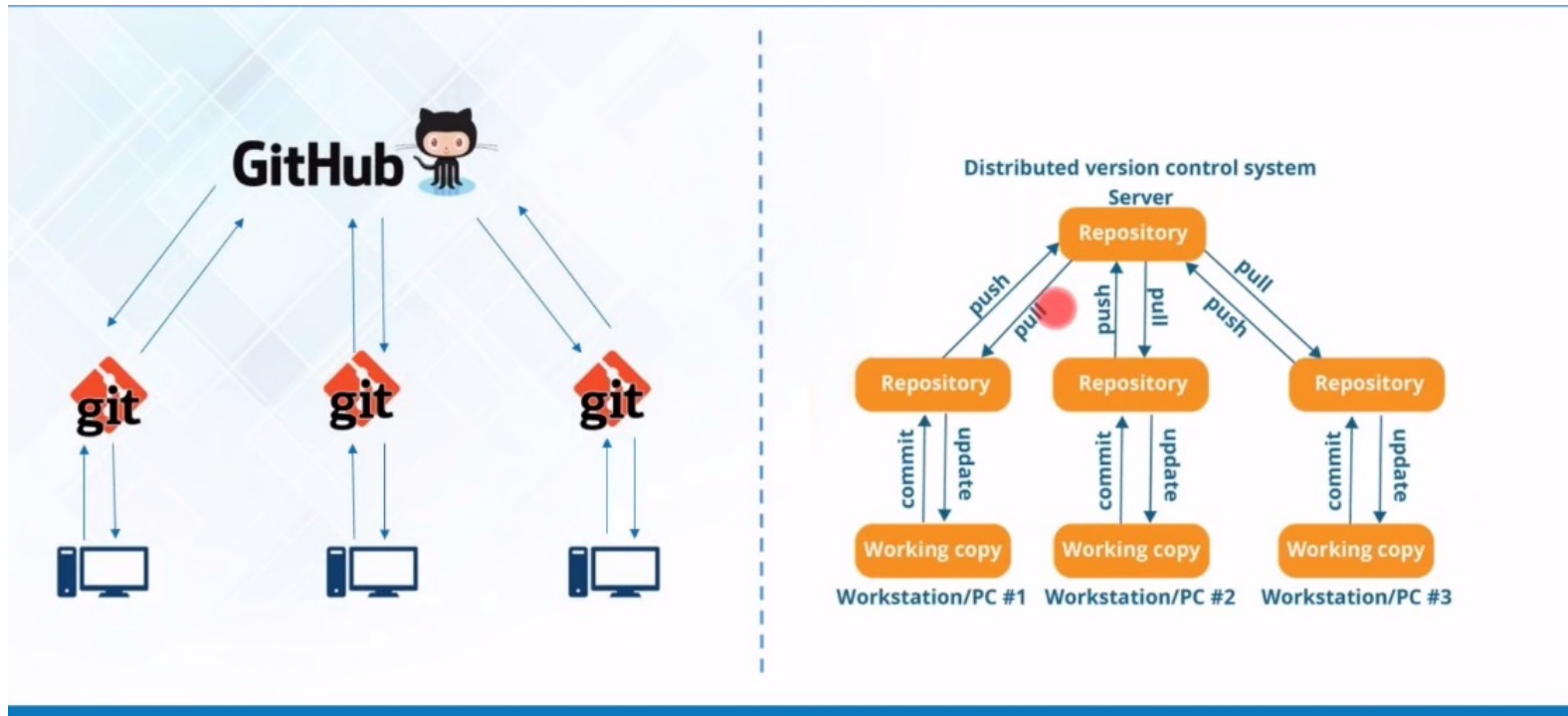
Why version control?

- For working by yourself:
 - Gives you a “time machine” for going back to earlier versions
 - Gives you great support for different versions (standalone, web app, etc.) of the same basic project
- For working with others:
 - Greatly simplifies concurrent work, merging changes.
- **Version control is important** for documents that undergo a lot of revision and redrafting and is particularly **important** for electronic documents because they can easily be changed by a number of different users. These changes may not be immediately apparent.

Version Control System Tool



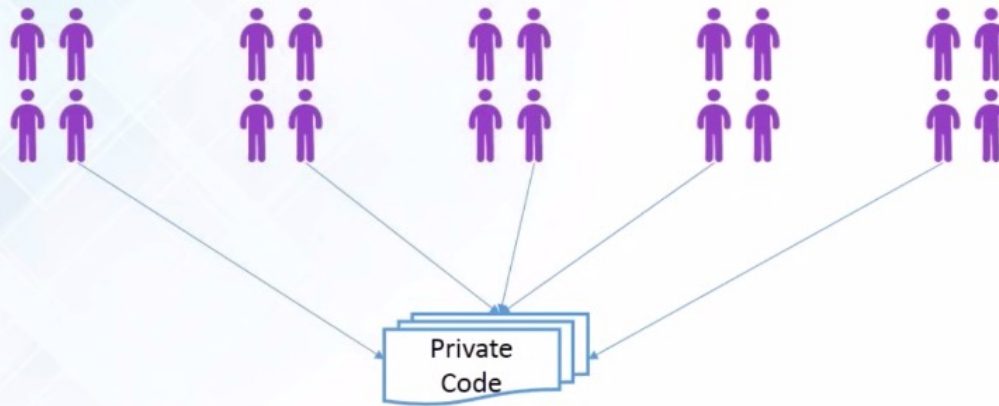
GitHub and Git



Dominion Enterprises case study

Problem Statement:

Each team has its own goals, projects, and budgets and they also have Unique needs and workflows

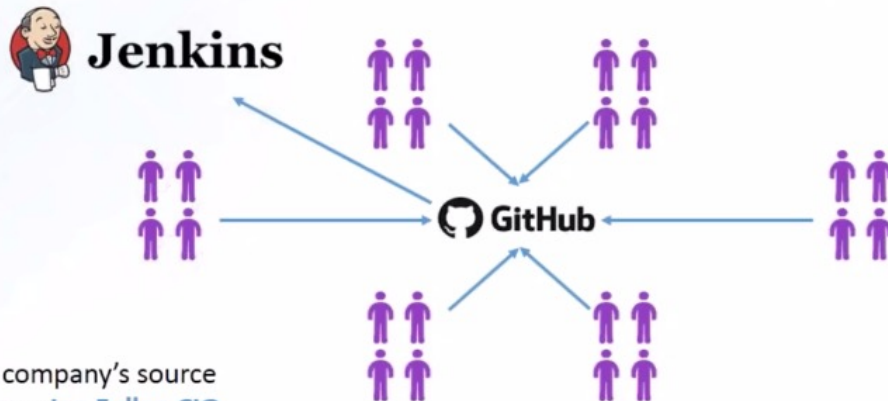


They wanted to make private code “publicly” to make their work more transparent across the company

Solution

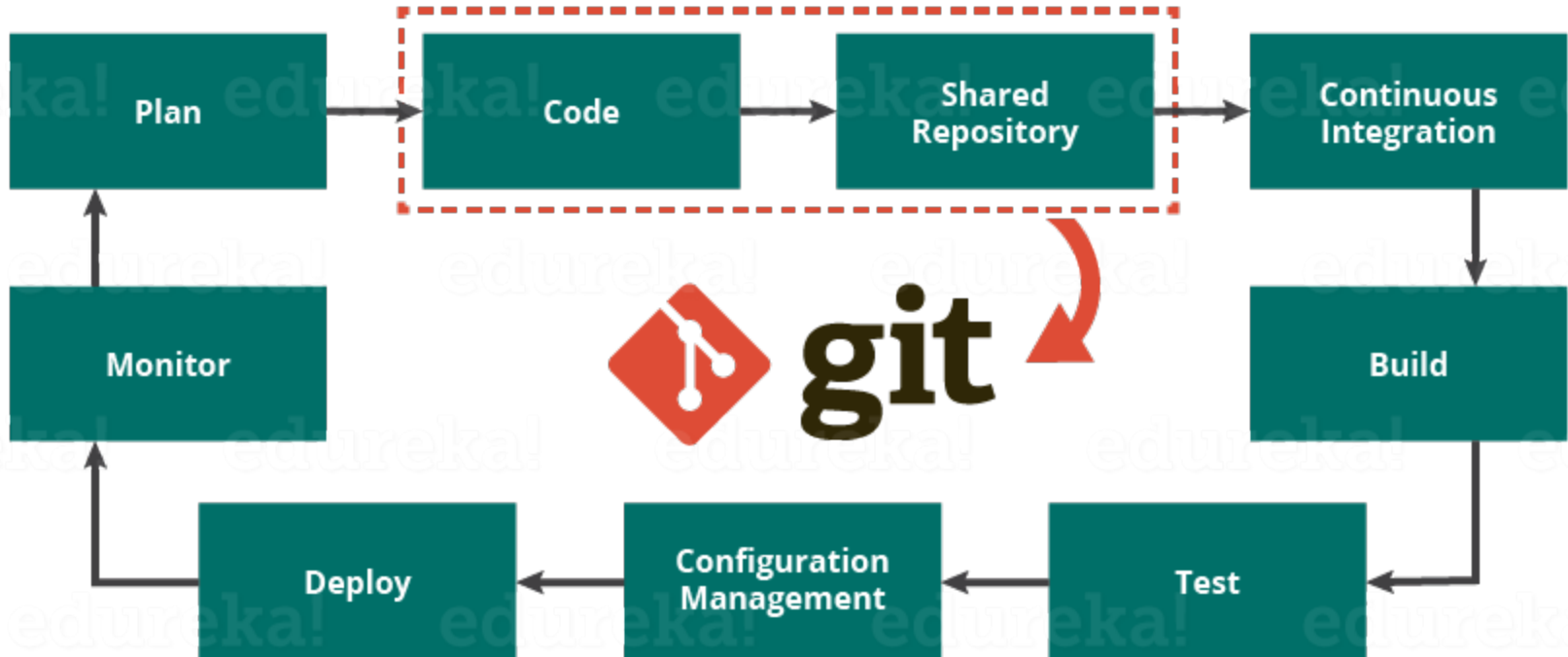
Reason for using
GitHub as the solution:

- ❑ They noticed that few of the teams were already using GitHub. Adopting a familiar platform has also made onboarding easier for new employees.
- ❑ Having all of their code in one place makes it easier for them to collaborate on projects.



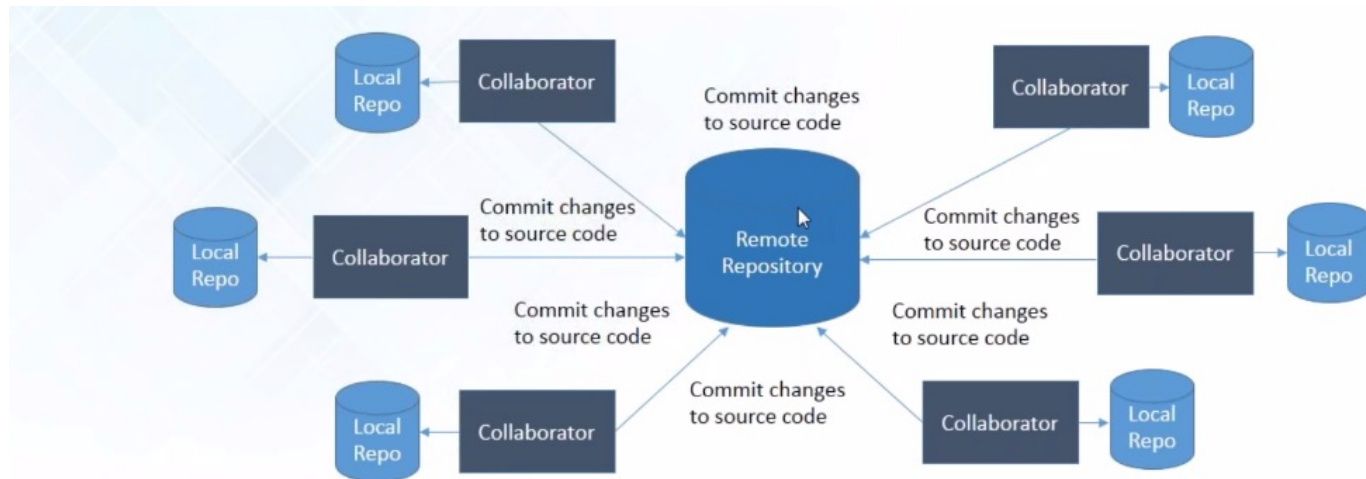
GitHub Enterprise has allowed us to store our company's source code in a central, corporately controlled system. - [Joe Fuller, CIO](#)

Git



What is Git

- Git is a distributed revision control and source code management system with an emphasis on speed.
- Git was initially designed and developed by Linus Torvalds for Linux kernel development.
- Git is a free software distributed under the terms of the GNU General Public License version 2.





Why Git?

- Git has many advantages over earlier systems such as CVS and Subversion
 - More efficient, better workflow, etc.
 - See the literature for an extensive list of reasons
 - Of course, there are always those who disagree
- Subversion offers a centralized model whereas Git offers a decentralized model.
- With git, everyone has their own copy of the entire repository. Basically, everyone is always working on their own branch and functional code can later be merged into the master. With SVN, everyone has a working copy and changes are committed to a central repository.

Features of Git



Distributed



Compatible



Non-linear



Branching



Lightweight



Speed



Open Source



Reliable



Secure



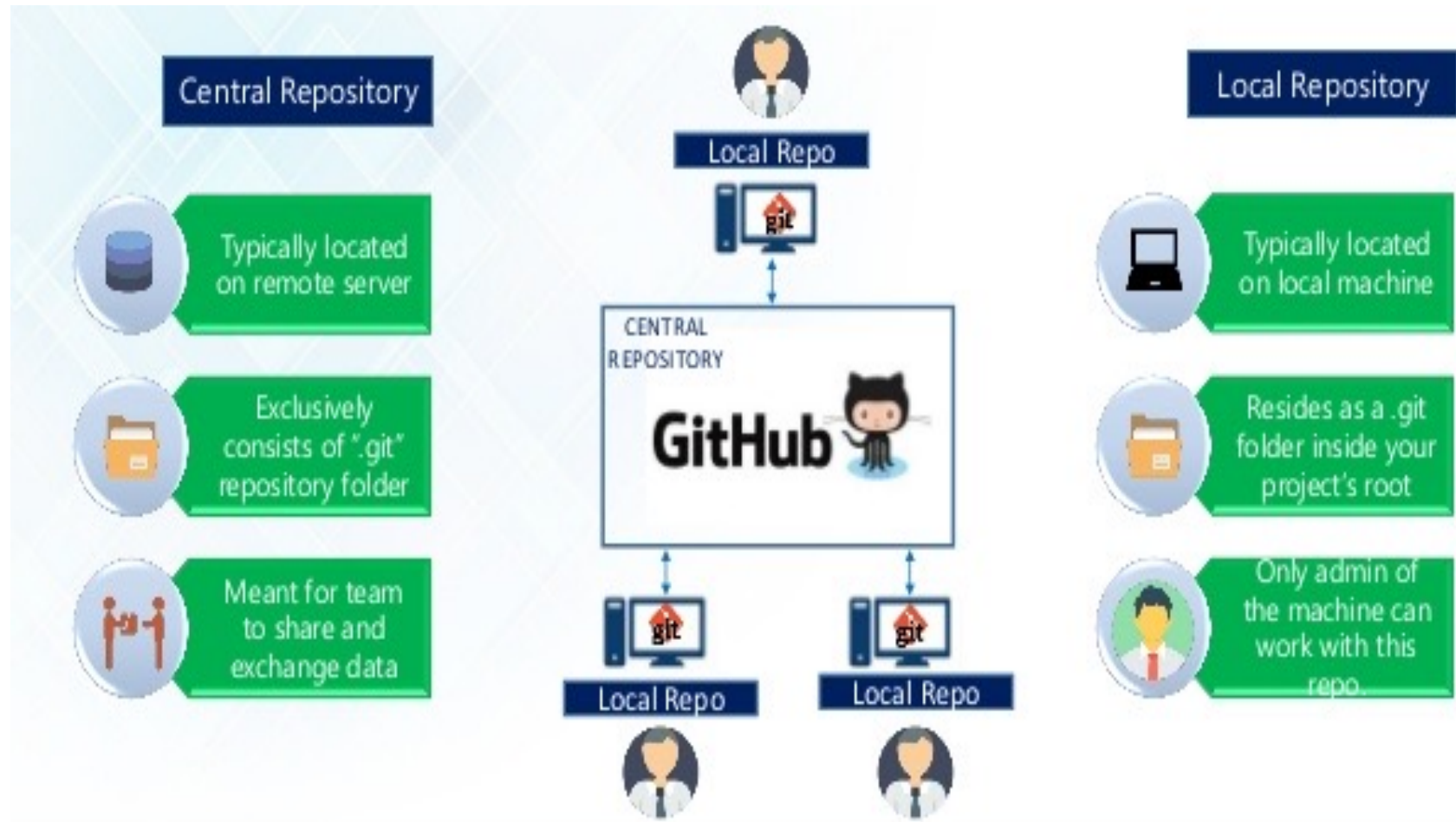
Economical



What is repository?

- a directory or storage space where your projects can live. It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, image files, you name it, inside a repository.
- There are 2 types of repository
 - Central
 - Local

Central Repository vs Local Repository



Git Operations and Commands

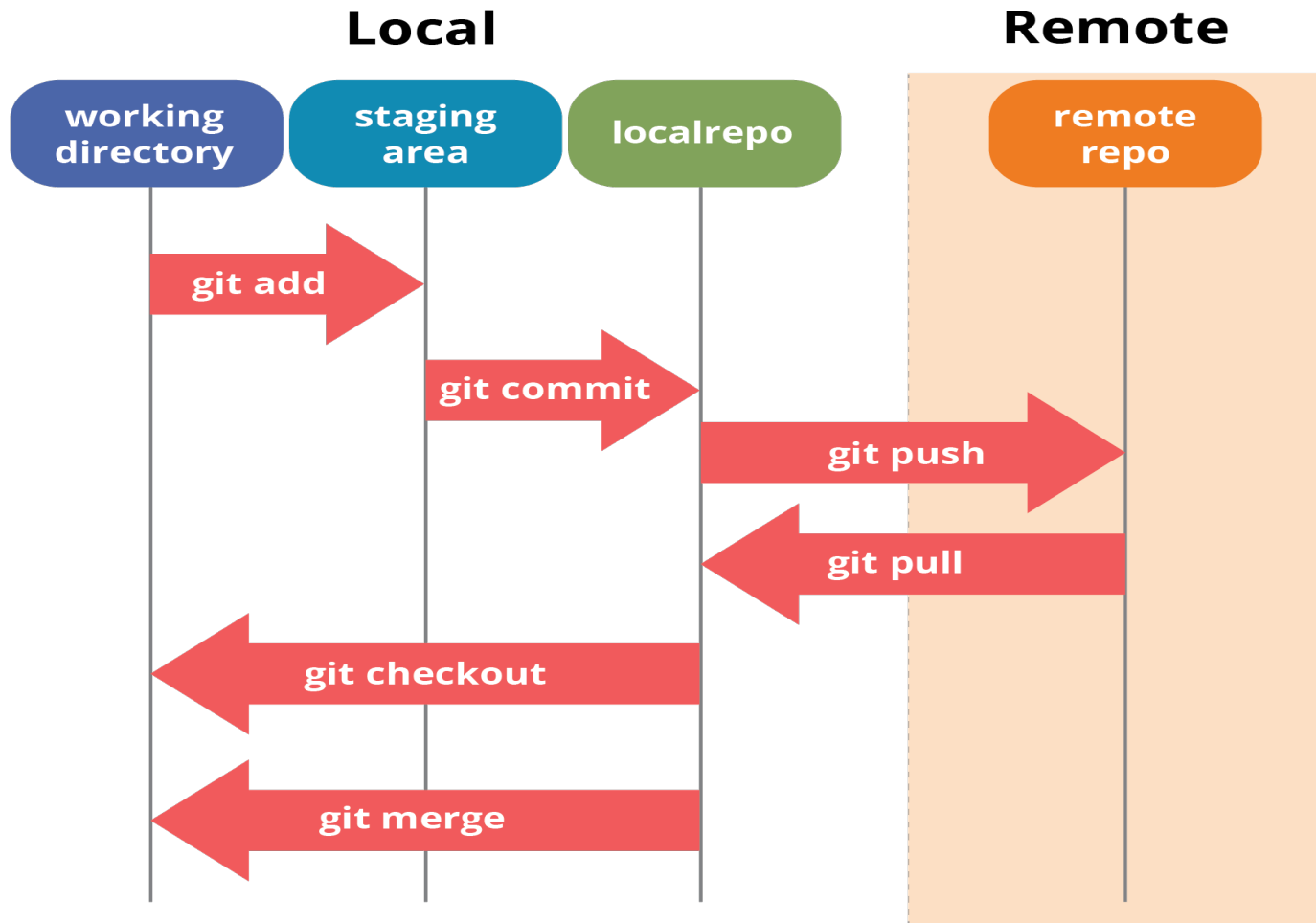




Basic operations of Git

- Some of the basic operations in Git are:
 - Initialize
 - Add
 - Commit
 - Pull
 - Push
- Some advanced Git operations are:
 - Branching
 - Merging
 - Rebasing

Operation Flow





Continue..

- Git has three main states that your files can reside in: *committed*, *modified*, and *staged*
- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- This leads us to the three main sections of a Git project: the Git directory, the working tree, and the staging area.



Continue..

- The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.
- The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.



Basic Git workflow

- The basic Git workflow goes something like this:
 - You modify files in your working tree.
 - You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
 - You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.
 - If a particular version of a file is in the Git directory, it's considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified



Directories

- each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
- Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.



Git Installation

- Install Git from <https://gitforwindows.org/>
- After installation:
 - Create folder in Windows
 - Open it
 - Right click and select Git bash.
 - On prompt, write '**git init**' to initialize git
 - '**Git add origin <link>**' to add remote repository to local machine
 - Link you will get from clone button in github



init and the .git repository

- When you said `git init` in your project directory, or when you cloned an existing project, you created a repository
 - The repository is a subdirectory named `.git` containing various files
 - The dot indicates a “hidden” directory
 - You do *not* work directly with the contents of that directory; various git commands do that for you
 - You *do* need a basic understanding of what is in the repository

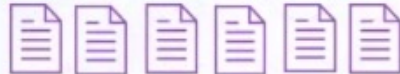


Syncing Repository

- Pull files with ‘git pull origin master’
- Push your own changes into central repo with ‘git push’

Making Changes

git status



- Tells you which files are added to index and are ready to commit.

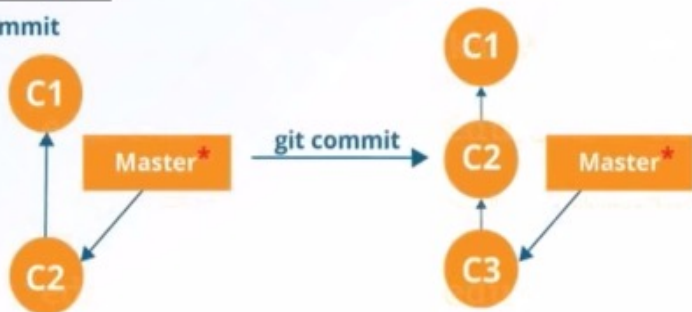
git add



- Lets you add files to your index.

git commit

Commit



- It refers to recording snapshots of the repository at a given time.
- Committed snapshots will never change unless done explicitly.



Typical workflow

- `git pull remote_repository`
 - Get changes from a remote repository and merge them into your own repository
- `git status`
 - See what Git thinks is going on
 - Use this frequently!
- Work on your files (remember to **add** any new ones)
- `git commit -m "What I did"`
- `Git Commit -a -m "message"`
 - makes Git automatically stage every file that is already tracked before doing the commit.
- `git push`



Git Log

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

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

```
    removed unnecessary test
```

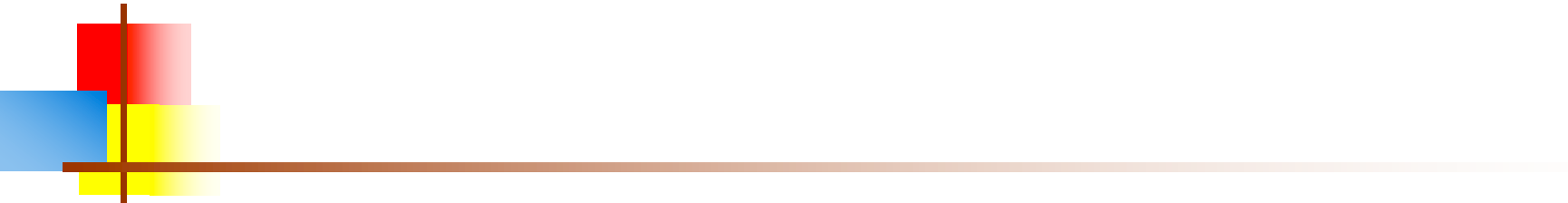
```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

```
    first commit
```



Remove File

- To remove a file from Git, you have to remove it from your tracked files and then commit.
 - `git rm <filename>`
- It removes the file from your working directory so you don't see it as an untracked file the next time around.
- you may want to keep the file on your hard drive but not have Git track it anymore.
 - `git rm --cached <filename>`

- 
- 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
 - `git commit --amend`
 - let's say you've changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. Then you can unstage one of the two by using command:
 - `git reset HEAD <filename>`

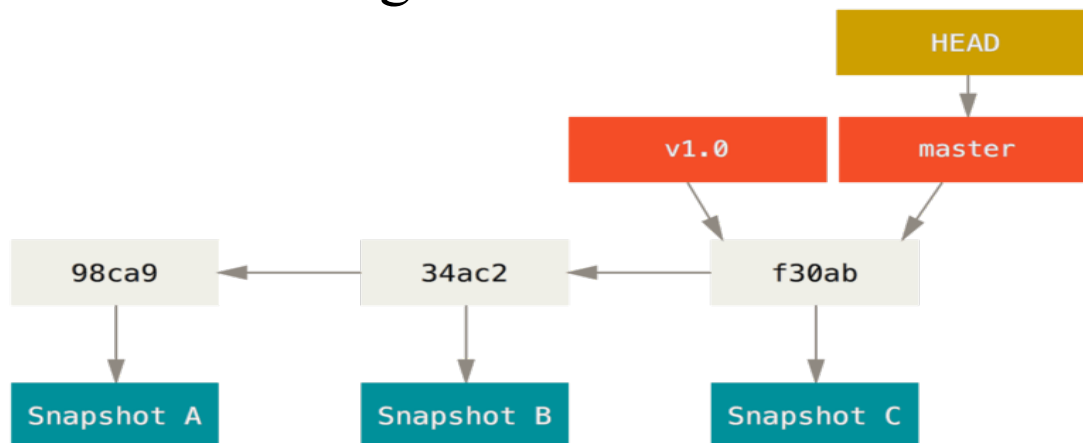


Working with remotes

- To see which remote servers you have configured
 - `Git remote`
- shows you the URLs that Git has stored for the shortname to be used
 - `Git remote -v`
- To add a new remote Git repository as a shortname you can reference easily, run `git remote add <shortname> <url>`
 - `git remote add pb https://github.com/paulboone/ticgit`
- to get data from your remote projects
 - `git fetch <remote>`
- If you want to remove a remote
 - `git remote remove <url>`

Parallel Development-Branching

- Branches are pointers to a specific commit.
- The default branch name is Master. Master points to current snapshot.
- Branches are of 2 types:
 - Local Branches
 - Remote-tracking branches





Branching

- To create new branch
 - `Git branch <branchname>`
 - Example-> `git branch firstbranch`
 - New branch contains all files of master branch
- To change branch
 - `Git checkout <branchname>`
- Create a new branch and switch to it
 - `git checkout -b <branchname>`
- Delete the feature branch
 - `git branch -d <branchname>`



Remote Branching

- To push data on remote branch
 - `git push <remote> <branch>`
- To activate remote branch
 - `git checkout -b <branch> <remote>/<branch>.`
 - `git push origin --delete <remote>/<branch>`



Merging

- It is a way to combine the work of different branches together.
- Allows to branch off, develop a new feature & combine it back in.
- Let assume you want to add changes from firstbranch to master branch:
 - `Git checkout master`
 - `Git merge firstbranch`
 - `Ls`



Rebasing

- Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features.
- consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch.
 - `Git rebase <branch>`



Continue..

- Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further.
 - `git rebase --onto master server client`
- `git rebase -- d` means during playback the commit will be discarded from the final combined commit block.
- `git rebase -- p` leaves the commit as is. It will not modify the commit's message or content and will still be an individual commit in the branches history.
- `git rebase -- x` during playback executes a command line shell script on each marked commit. A useful example would be to run your codebase's test suite on specific commits, which may help identify regressions during a rebase.



Protocol

- Git can use four distinct protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git.

Local Protocol

- The most basic is the *Local protocol*, in which the remote repository is in another directory on the same host. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer.
- To clone a repository like this, or to add one as a remote to an existing project, use the path to the repository as the URL.



Continue..

- `git clone <remote project url>`
- To add a local repository to an existing Git project, you can run something like this:
`git remote add local_proj /srv/git/project.git`

Smart HTTP

- Smart HTTP operates very similarly to the SSH or Git protocols but runs over standard HTTPS ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.



Continue..

- It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the `git://` protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both.
 - `git clone https://example.com/gitproject.git`

The SSH Protocol

- A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places — and if it isn't, it's easy to do. SSH is also an authenticated network protocol and, because it's ubiquitous, it's generally easy to set up and use
 - `git clone ssh://[user@]server/project.git`



Continue..

- In order to initially set up any Git server, you have to export an existing repository into a new bare repository — a repository that doesn't contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone command with the `--bare` option. By convention, bare repository directory names end with the suffix `.git`, like so:
 - `git clone --bare my_project my_project.git`
 - Cloning into bare repository 'my_project.git' ...



Putting the Bare Repository on a Server

- Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let's say you've set up a server called `git.example.com` to which you have SSH access, and you want to store all your Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, you can set up your new repository by copying your bare repository over:
 - `scp -r my_project.git user@git.example.com:/srv/git`
- At this point, other users who have SSH-based read access to the `/srv/git` directory on that server can clone your repository by running
 - `git clone user@git.example.com:/srv/git/my_project.git`



Generating Your SSH Public Key

- The .pub file is your public key, and the other file is your private key. If you don't have these files (or you don't even have a .ssh directory), you can create them by running a program called ssh-keygen,
 - `ssh-keygen -o`



Stashing and Cleaning

- Suppose you are implementing a new feature for your product. Your code is in progress and suddenly a customer escalation comes. Because of this, you have to keep aside your new feature work for a few hours. You cannot commit your partial code and also cannot throw away your changes. So you need some temporary space, where you can store your partial changes and later on commit it.
- In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.



Continue

- To push a new stash onto your stack, run the `git stash` command.
- We can view a list of stashed changes by using the `git stash list` command.
- You can reapply previously stashed changes with `git stash pop`
- you can reapply the changes to your working copy and keep them in your stash with `git stash apply`



Continue..

- You can also choose to stash just a single file, a collection of files, or individual changes from within files.
 - `Git stash -p` or `–patch`.
- you can use `git stash branch` to create a new branch to apply your stashed changes to
 - `git stash branch <new-branch name> stash@{1}`
- If you decide you no longer need a particular stash, you can delete it with `git stash drop`
 - `git stash drop stash@{1}`

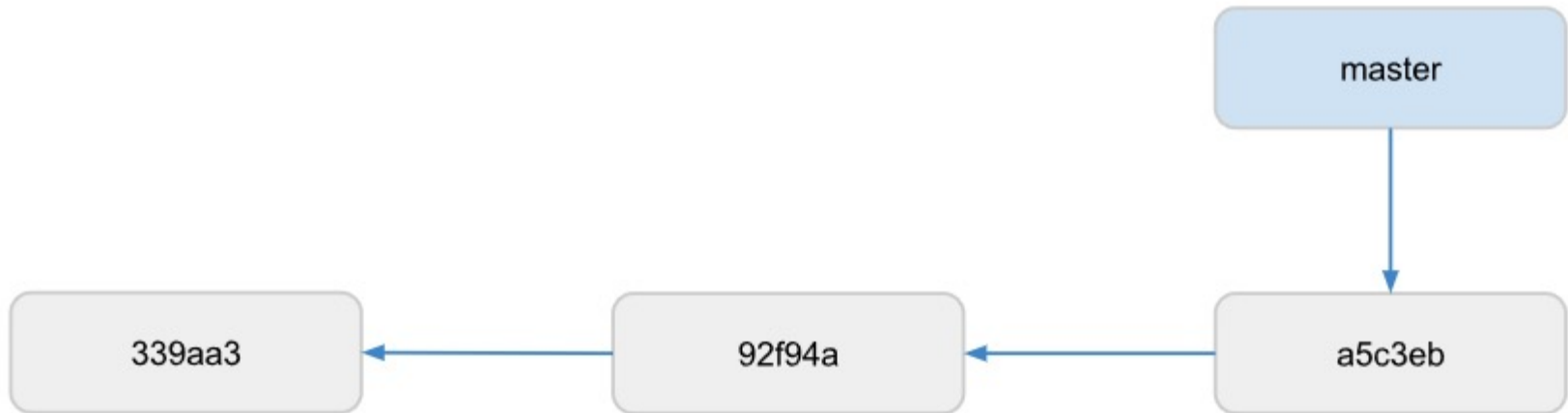



Cleaning

- you do want to remove cruft files or clean your working directory, you can do so with `git clean`.
- To remove all the untracked files in your working directory, you can run `git clean -f -d`

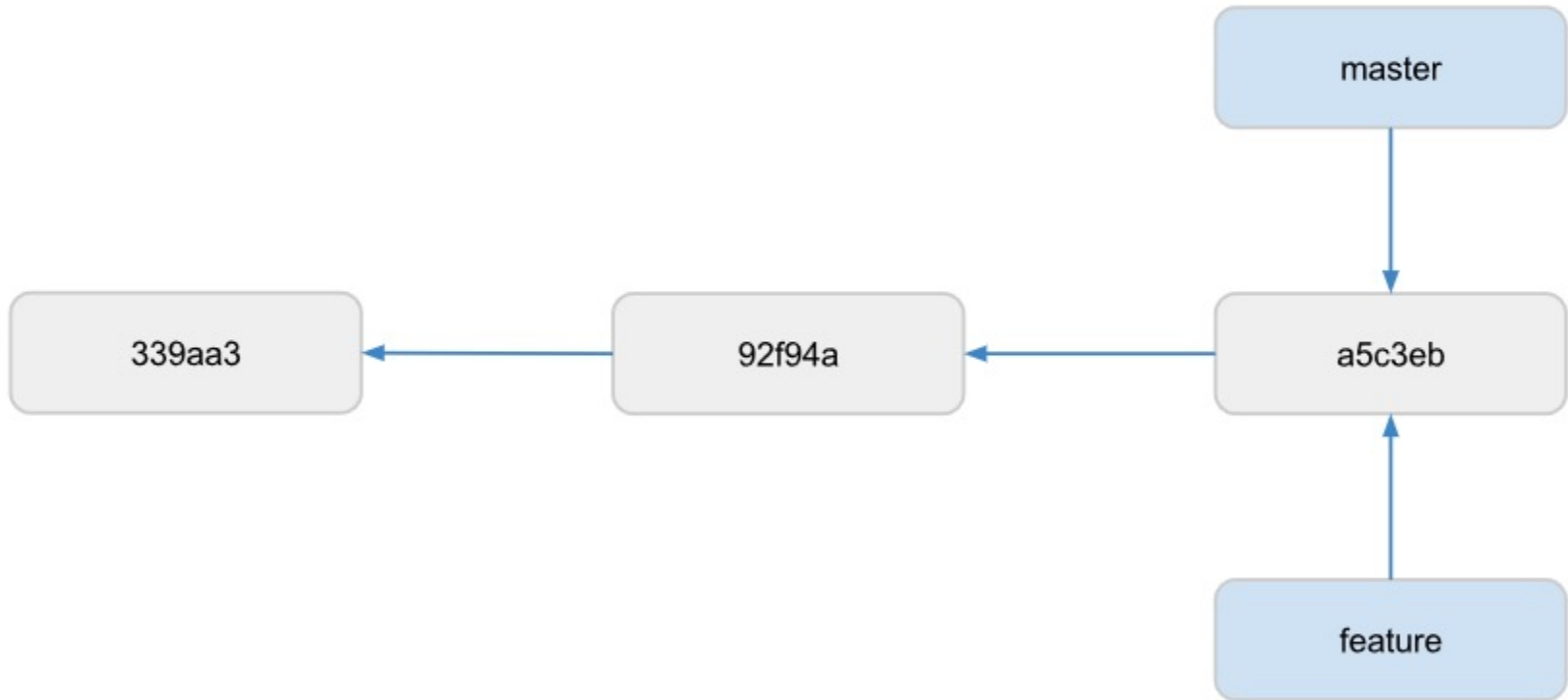
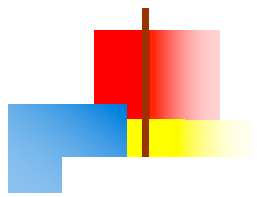
Branches in Git

Let's start by looking at a simple commit history.



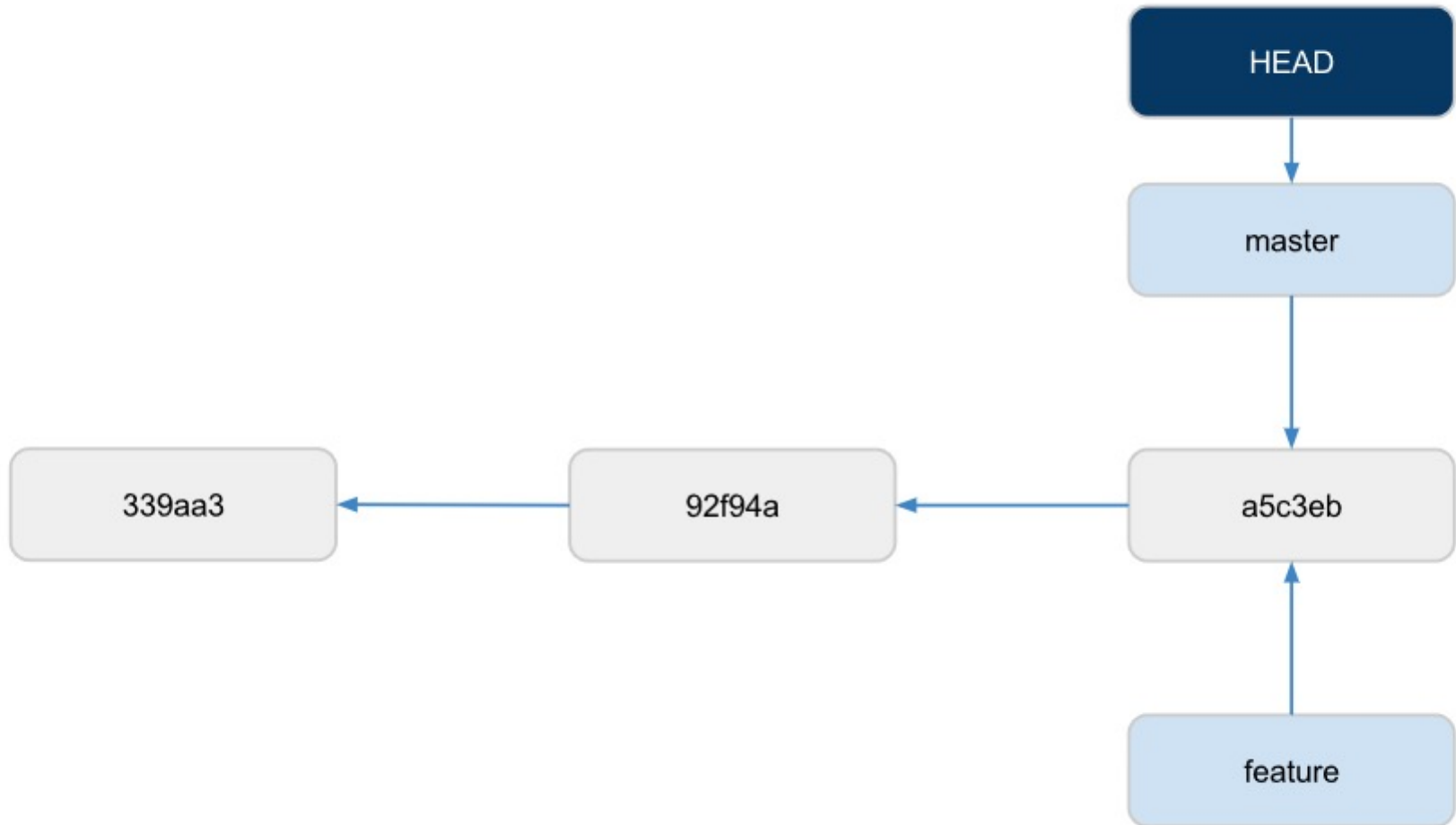
- 
- Let's assume our entire repository history is made up only from those three commits. There's a pointer called master pointing at commit a5c3eb. That's all what branches are: they are movable pointers. Let's create a branch called feature.

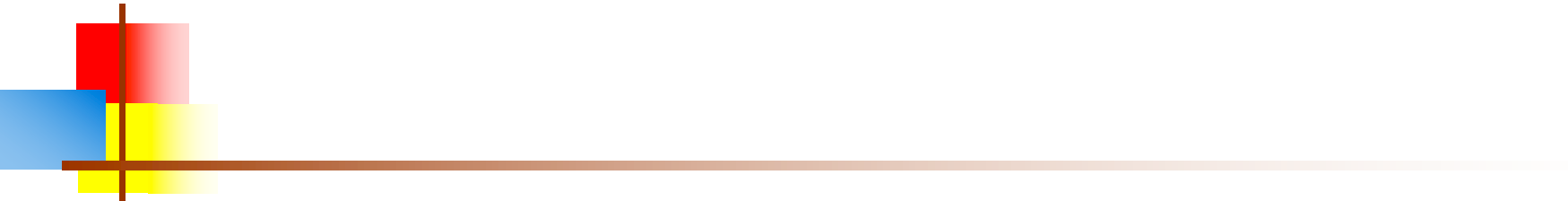
git branch feature



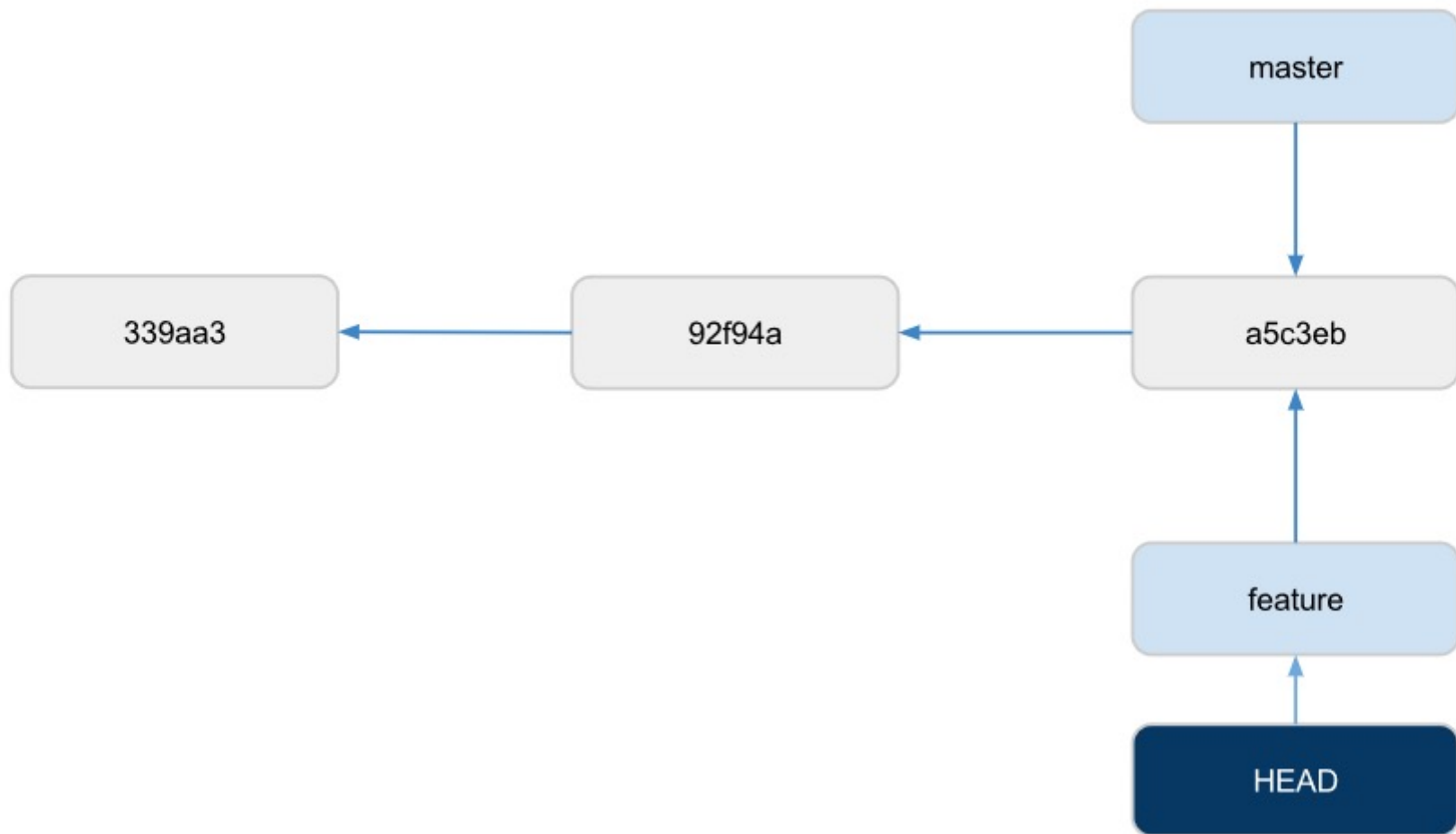
We just created another pointer called feature pointing at the exact same commit. Basically, all Git does it creates a file called feature with the contents being a 40 char string, the SHA-1 revision name of the commit.

- But wait! Now that we have two different branches pointing to the same commit. How does Git know which branch is currently checked out? This is where the HEAD pointer comes into play!



- 
- The HEAD is a special pointer that simply points to the currently checked out branch or commit. And again, it's a simple file inside the .git folder called HEAD which in our case currently contains the string master.
 - Ok then, what happens if we switch to our feature branch?

git checkout feature



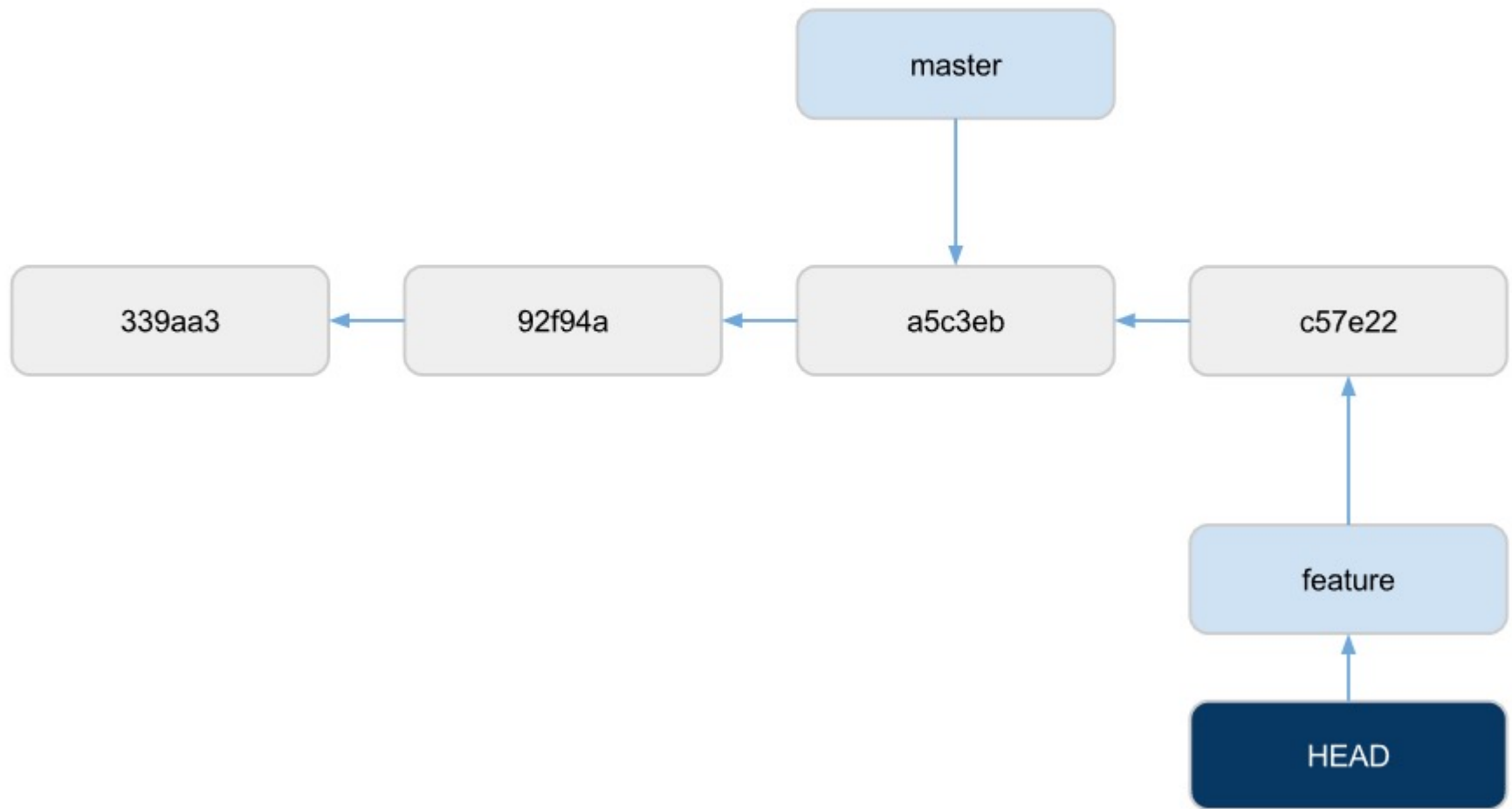
Exactly, all what happened is that HEAD is now pointing to feature instead of master. Switching between master and feature at this point boils down to Git replacing the string master with feature in the HEAD file. Super cheap!

- 
- But what happens if we now create or modify some files and make another commit? Let's find out.

```
vim file.txt
```

```
git add file.txt
```

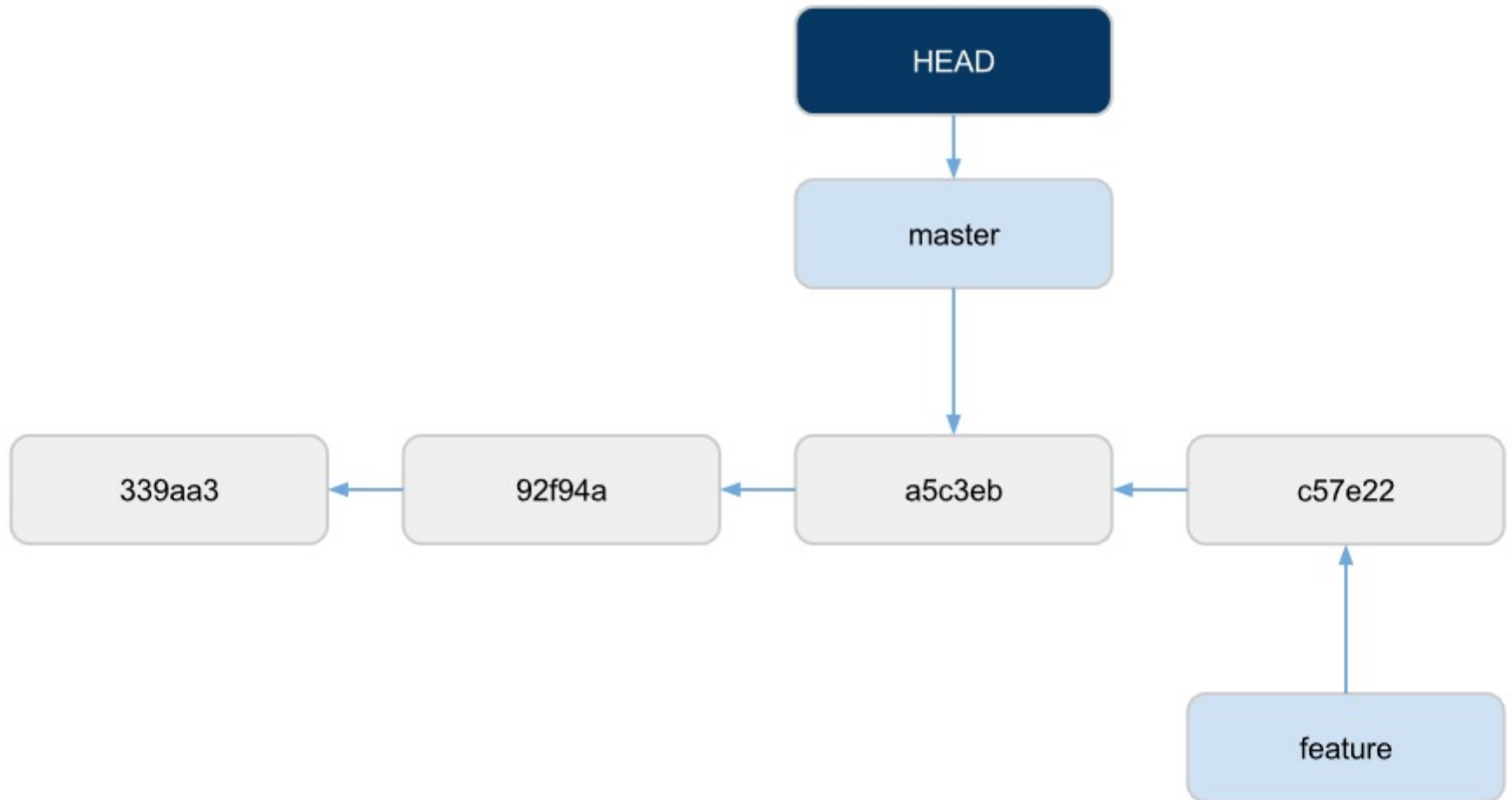
```
git commit -m "yay, that's fun!"
```

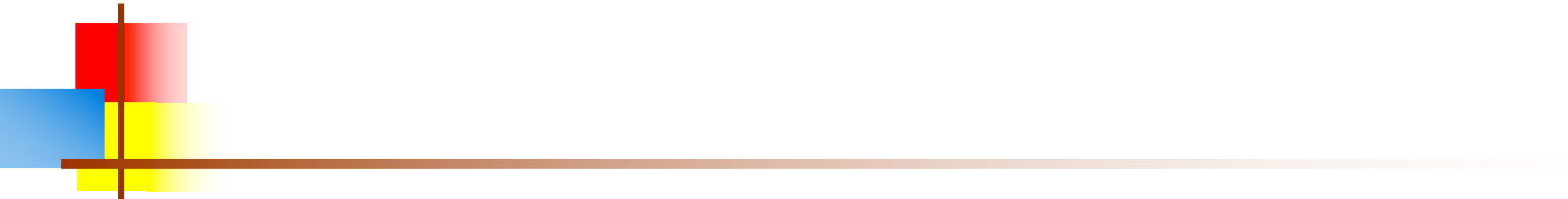


We created a new commit c57e22 and the feature pointer moved on to it. That's because branches are simply movable pointers to commits. But why did feature move and master did not? Because feature is the currently checked out branch. And how does Git know? Because HEAD points to feature!

- Let's switch back to master.

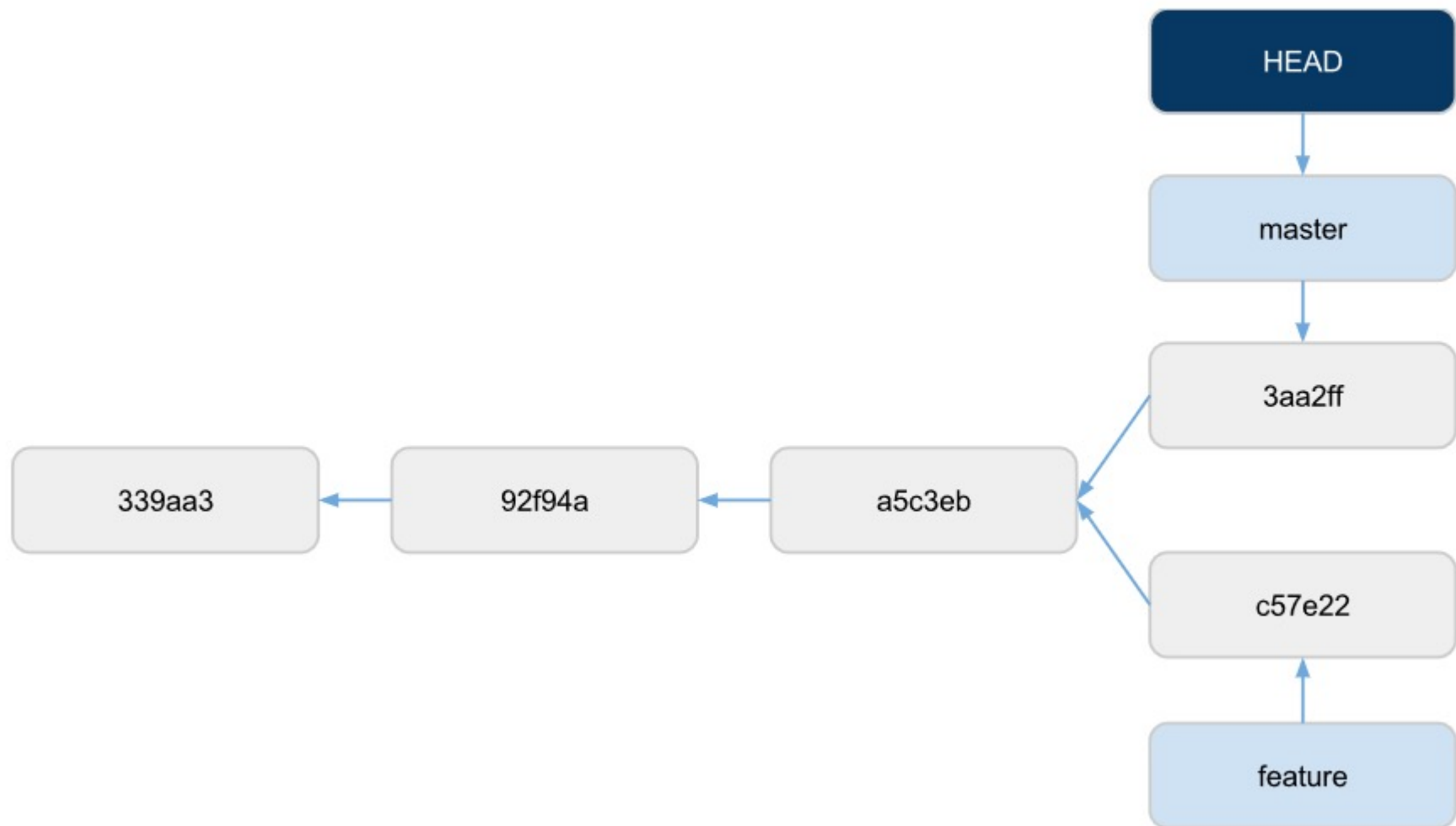
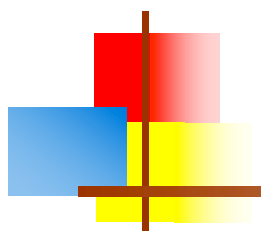
git checkout master

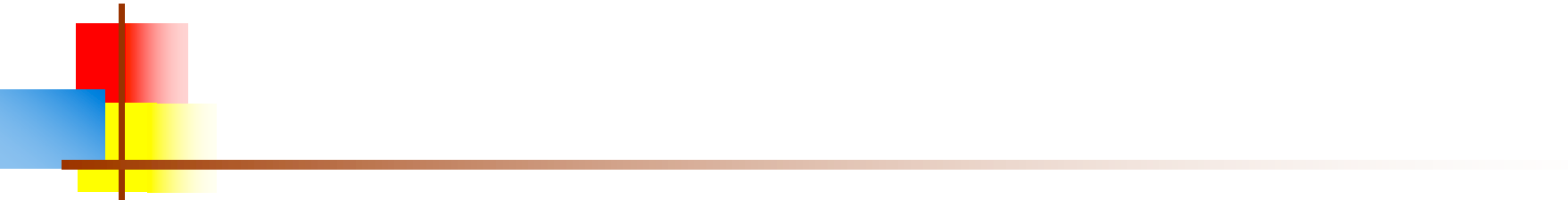


- 
- What happened now is that the HEAD pointer changed to master and the state of the entire working directory was changed to what it was at a5c3eb.

Let's modify a file and create another commit.

```
vim anotherFile.txt  
git add anotherFile.txt  
git commit -m "yay, more fun!"
```



- 
- A new commit 3aa2ff appeared and master moved on to it. At this point our history has diverged. And this is the whole point of branches. They enable us to have parallel evolution of a code base.



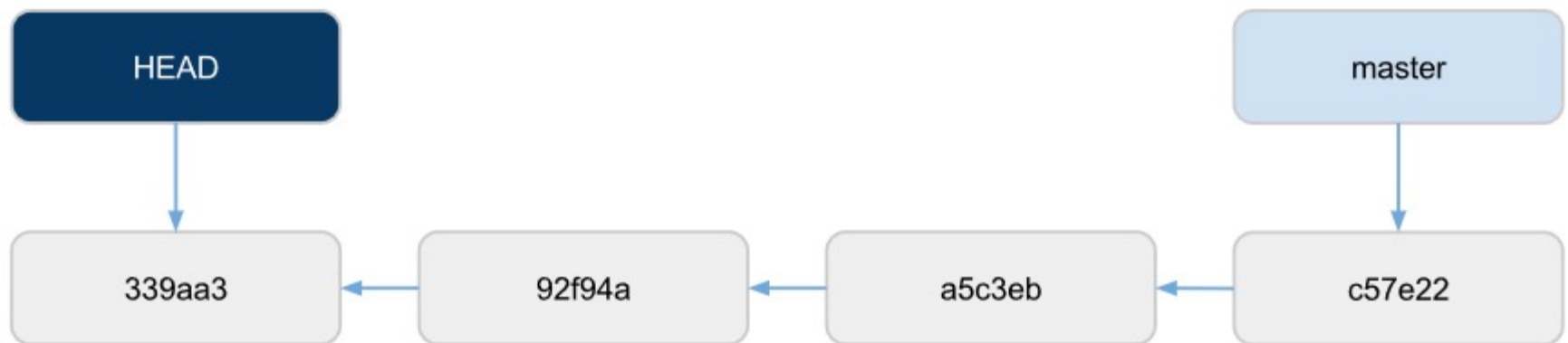
Detached HEAD

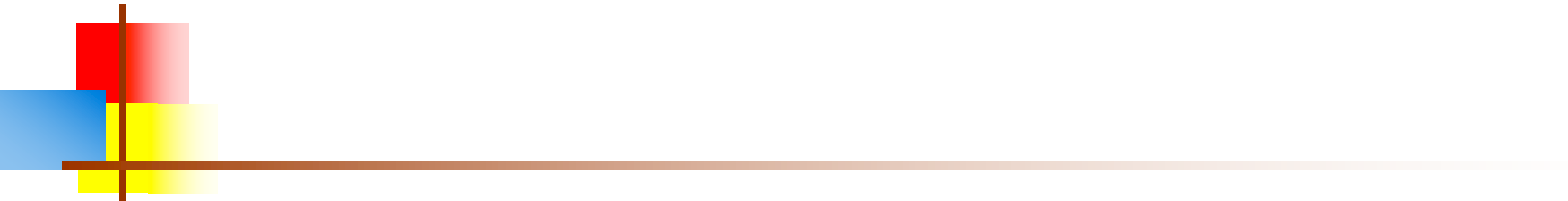
- Ever came across a detached HEAD? Now that we know how Git handles branches, it's the perfect time to demystify the detached HEAD.

The HEAD is a special pointer that simply points to the currently checked out branch or commit

- 
- Not only can we check out branches, we can also checkout any commit revision explicitly.

`git checkout 339aa3`



- 
- What happened now is that HEAD points to 339aa3 explicitly and the entire working directory was changed to what it looked like at that commit.

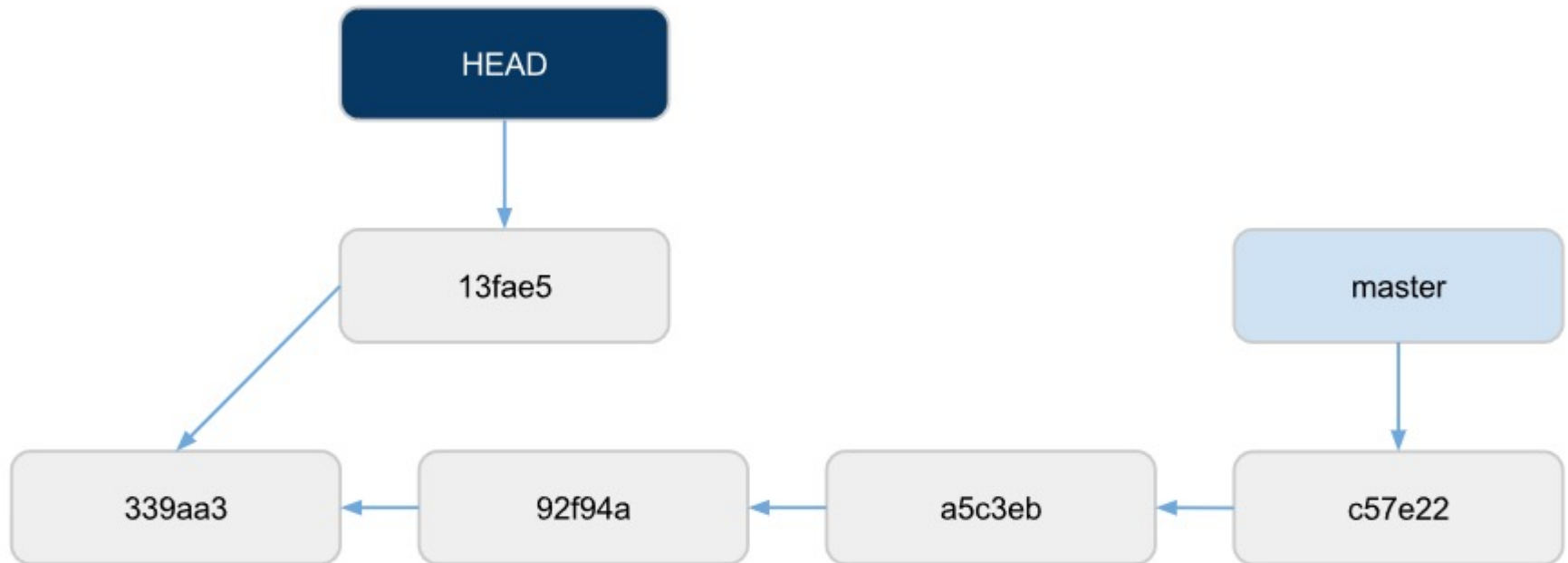
When HEAD points to a commit hash explicitly (rather than to a branch) it's in a detached state.

- What happens if we create new commits from here?
Let's find out!

```
vim someFile.txt
```

```
git add someFile.txt
```

```
git commit -m "detached HEAD fun"
```

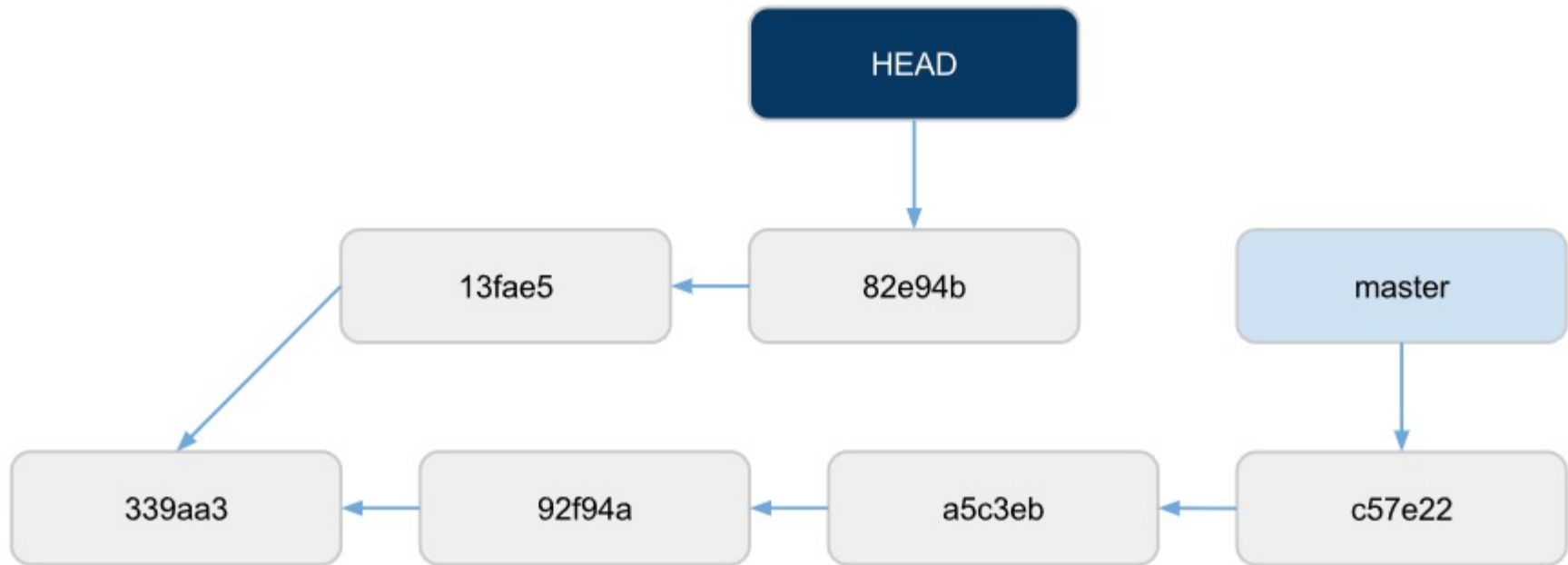


- Let's add one more!

```
vim someFile.txt
```

```
git add someFile.txt
```

```
git commit -m "more detached HEAD fun"
```



- We can go on like that. Does that mean, we don't actually need branches? Yes and no. What we just did is kinda like having a branch without a branch name. It works but there are two problems:
- How do you remember the SHA-1 key without a simple branch name?
- Commits that are not reachable by any branch or tag will be garbage collected and removed from the repository after 30 days.

- We can simply create a new branch right here!

`git checkout -b bugfix`

