Week 4: Source Control: Git

git,github,source control

---

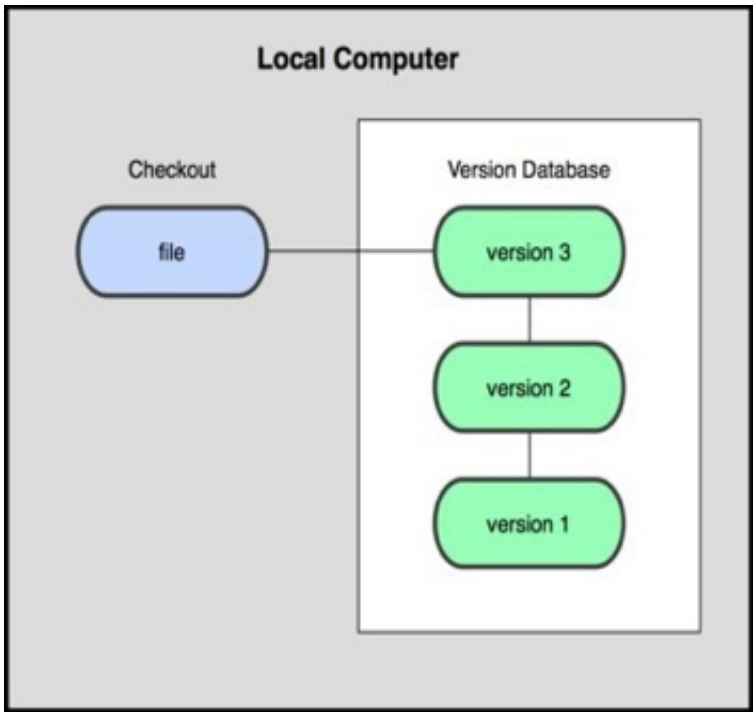
## Software Development Process

- involves making a lot of changes to code
    - new features added
    - bugs fixed
    - performance enhancements
- software team has many people working on the same/different parts of code
- many versions of software released
    - Ubuntu 10, Ubuntu 12, etc.
    - need to be able to fix bugs for Ubuntu 10 for customers using it
        - even though you've shipped Ubuntu 12

## Source/Version Control

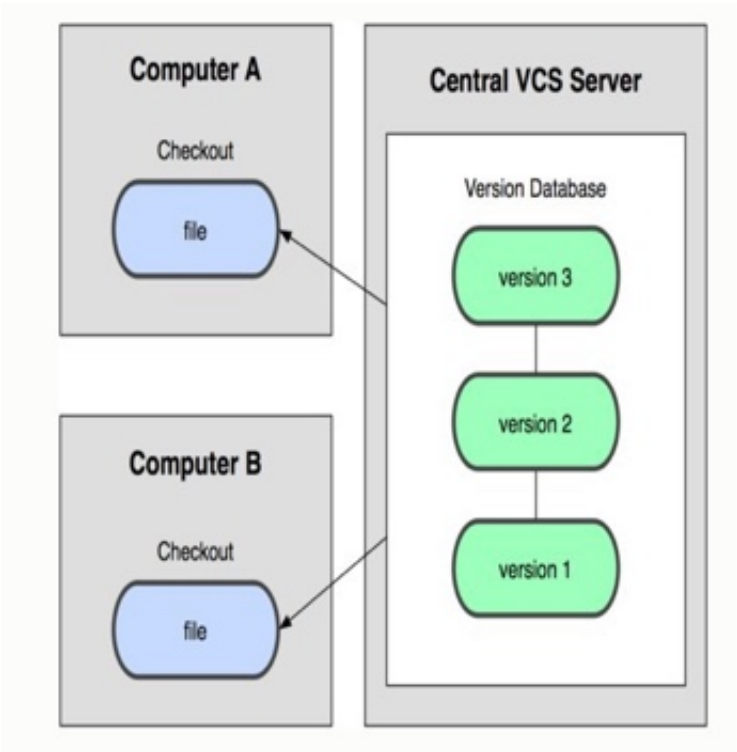
- track changes to code and other files related to the software
    - What new files were added?
    - What changes made to files?
    - Which version had what changes?
    - Which user made the changes?
- track entire history of the software
- version control software
    - GIT, Subversion, Perforce

## Local VCS




- organize different versions as folders on the local machine
- no server involved
- other users should copy it via disk/network

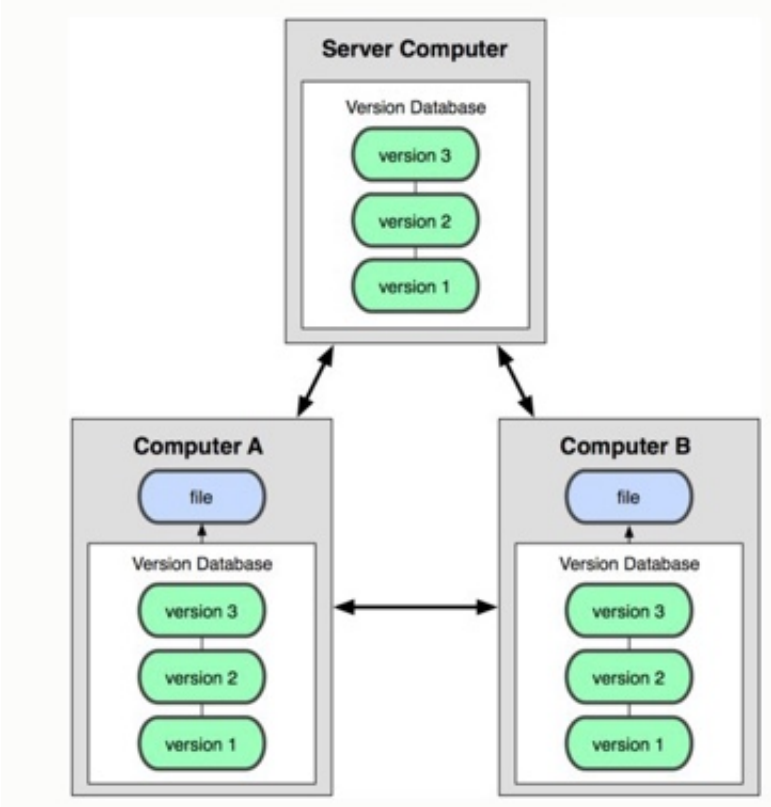
## Centralized VCS




- version history sits on a central server
- users will get a working copy of the files

- changes have to be committed to the server
- all users can get the changes

## Distributed VCS




- version history is replicated at **every user's machine**
- users have version control all the time
- changes can be communicated between users
- git is distributed

## Terminology

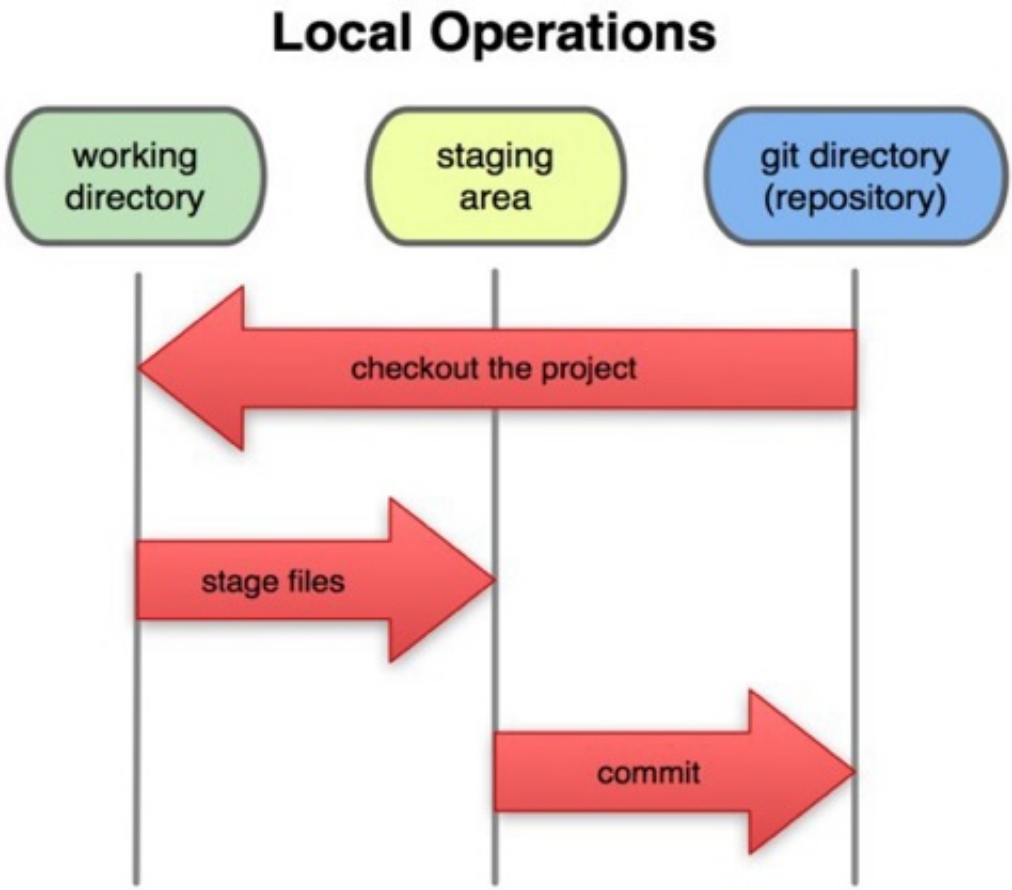
- **repository**
  - files and folders related to the software code
  - full history of the software
- **working copy**
  - copy of software's files in the repository at a specified version
- **check-out**
  - to create a working copy of the repository
- **check-in/commit**
  - write the changes made in the working copy to the repository
  - commits are recorded by the VCS

# Git Source Control

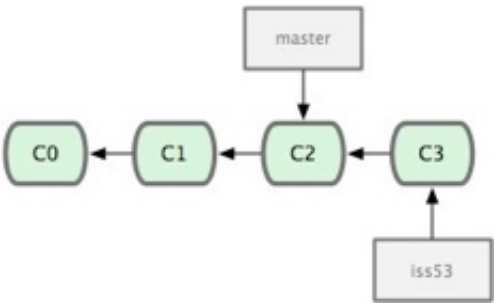
## Git Repository Objects

- objects used by GIT to implement source control
    - **blobs**
        - sequence of bytes
    - **trees**
        - groups blobs/trees together
    - **commit**
        - a particular version of the software
        - refers to a particular “git commit”
        - contains all information about the commit
    - **tags**
        - just a named commit object for convenience
            - ex: versions of the software
- objects uniquely identified with **hashes**

## Git States



## More Terminology




- **Head**
  - refers to a commit object
  - there can be many heads in a repository
- **HEAD**
  - refers to the currently active head
- **detached HEAD**
  - if a commit is not pointed to by a branch
  - this is okay if you want to just take a look at the code and if you don't commit any new changes
  - if new commits have to be preserved, a new branch has to be created
    - `git checkout v3.0 -b BranchVersion3.1`
- **branch**
  - refers to a head and its entire set of ancestor commits
- **master**
  - default branch

## Basic Git Commands

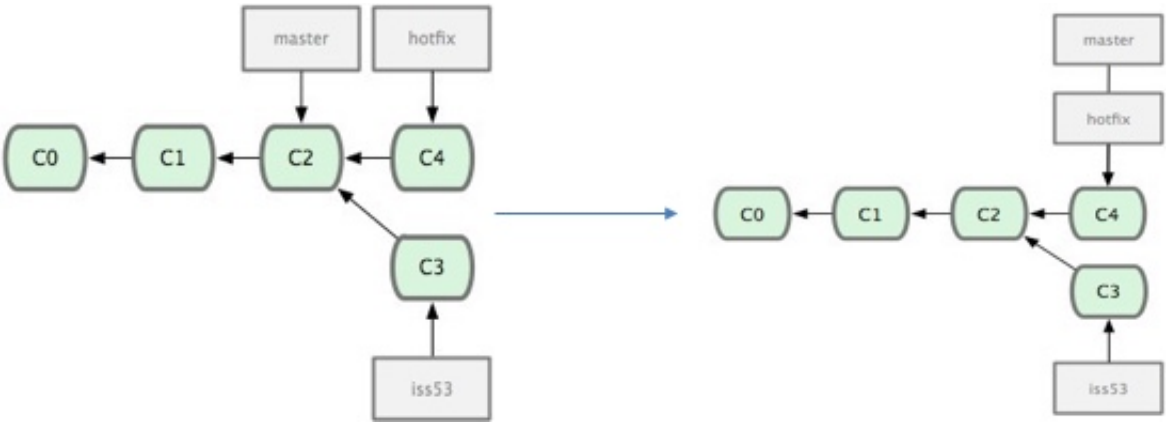
| Command | Action |
|---|---|
| `$ git init` | create a new repository (.git directory) |
| `$ git clone` | create a copy of an existing repository |
| `$ git checkout <tag/commit> -b <new_branch_name>` | creates a new branch |
| `$ git add` | stage modified/new files |
| `$ git commit` | check-in the changes to the repository |
| `$ git status` | shows modified files, new files, etc. |
| `$ git diff` | compares working copy with staged files |
| `$ git log` | shows history of commits |
| `$ git show` | show a certain object in the repository |

| | |
|---|---|
| `$ git help` | shows help for git actions |

## Git Example

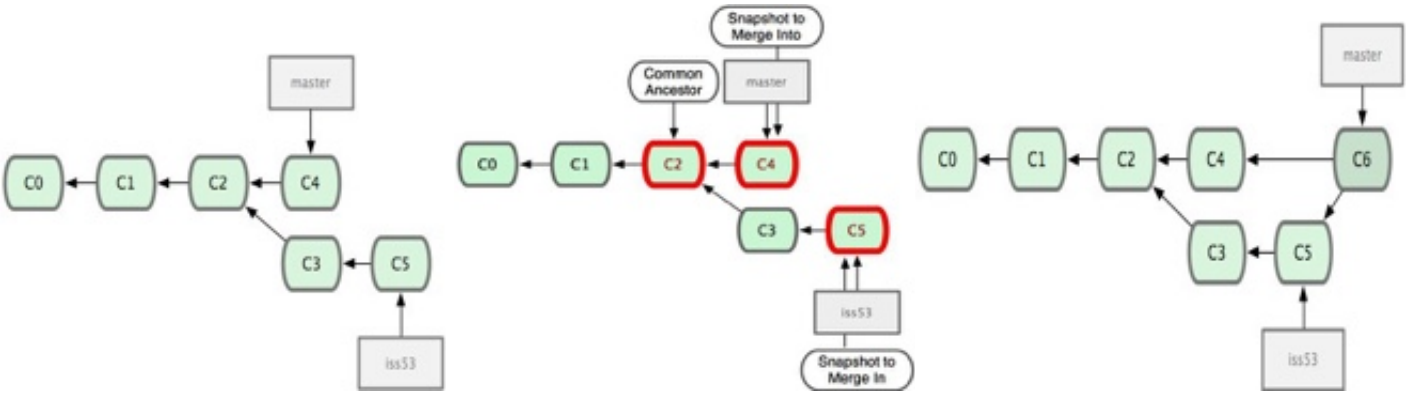
```
$ mkdir gitroot

$ cd gitroot

$ git init
# creates an empty git repo (.git directory with all necessary subdirectories)

$ echo “Hello World” > hello.txt

$ git add .
# Adds content to the index
# Must be run prior to a commit

$ git commit –m ‘Check in number one’

$ echo “I love Git” >> hello.txt

$ git status
# Shows list of modified files
# hello.txt

$ git diff
# Shows changes we made compared to index

$ git add hello.txt

$ git diff
# No changes shown as diff compares to the index

$ git diff HEAD
# Now we can see changes in working version

$ git commit –m “Second commit”
```

## Merging




- merging hotfix into master
  - `$ git checkout master`
  - `$ git merge hotfix`
- Git tries to merge automatically
  - simple if its a forward merge
    - **foward merge**: moves forward on the arrows
  - otherwise, you have to manually resolve conflicts

## Merging (complex)



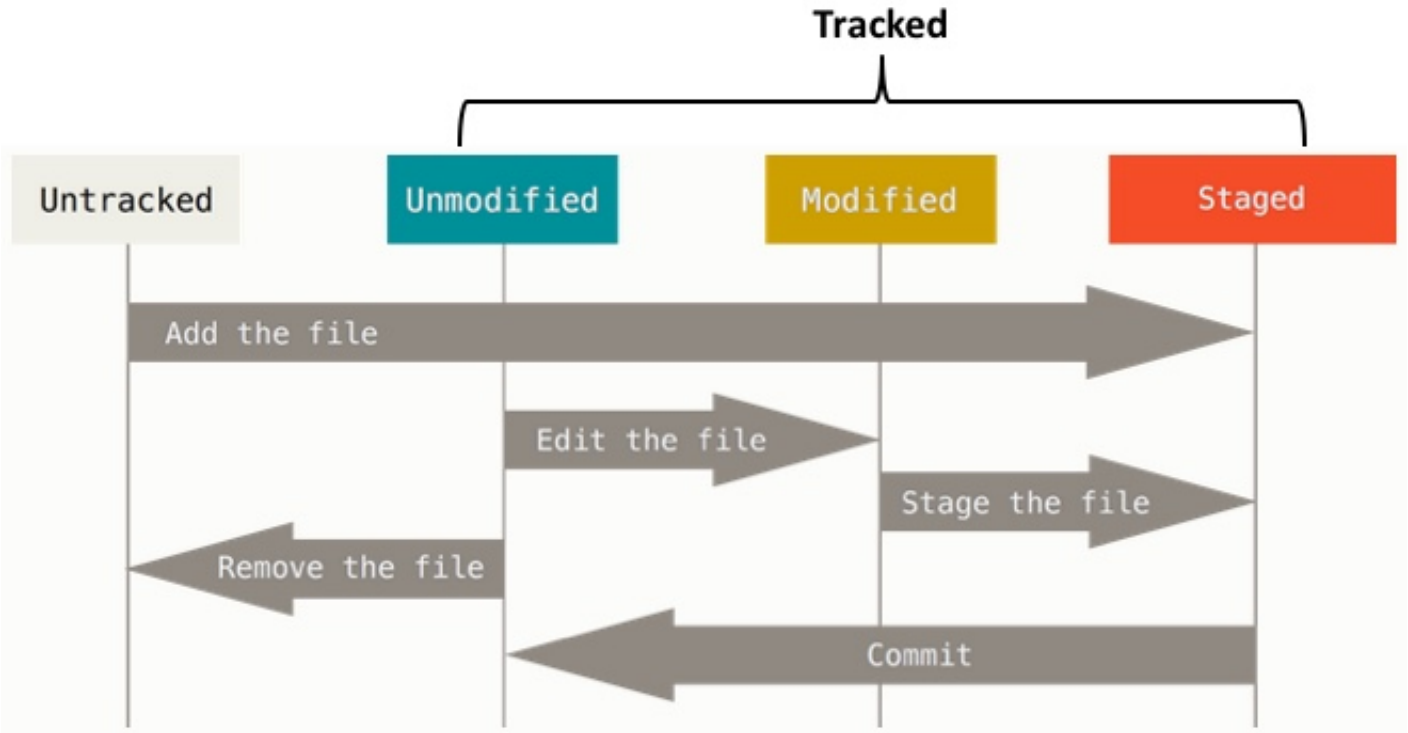

- merge iss53 into master
- git tries to merge automatically by looking at the changes since the common ancestor
- manually merge using 3-way merge or 2-way merge
  - merge conflicts — same part of the file was changed differently
- refer to mutliple parts
  - `$ git show hash`
  - `$ git show hash^2`
    - shows second parent
  - HEAD^^  HEAD~2

## More Git Commands

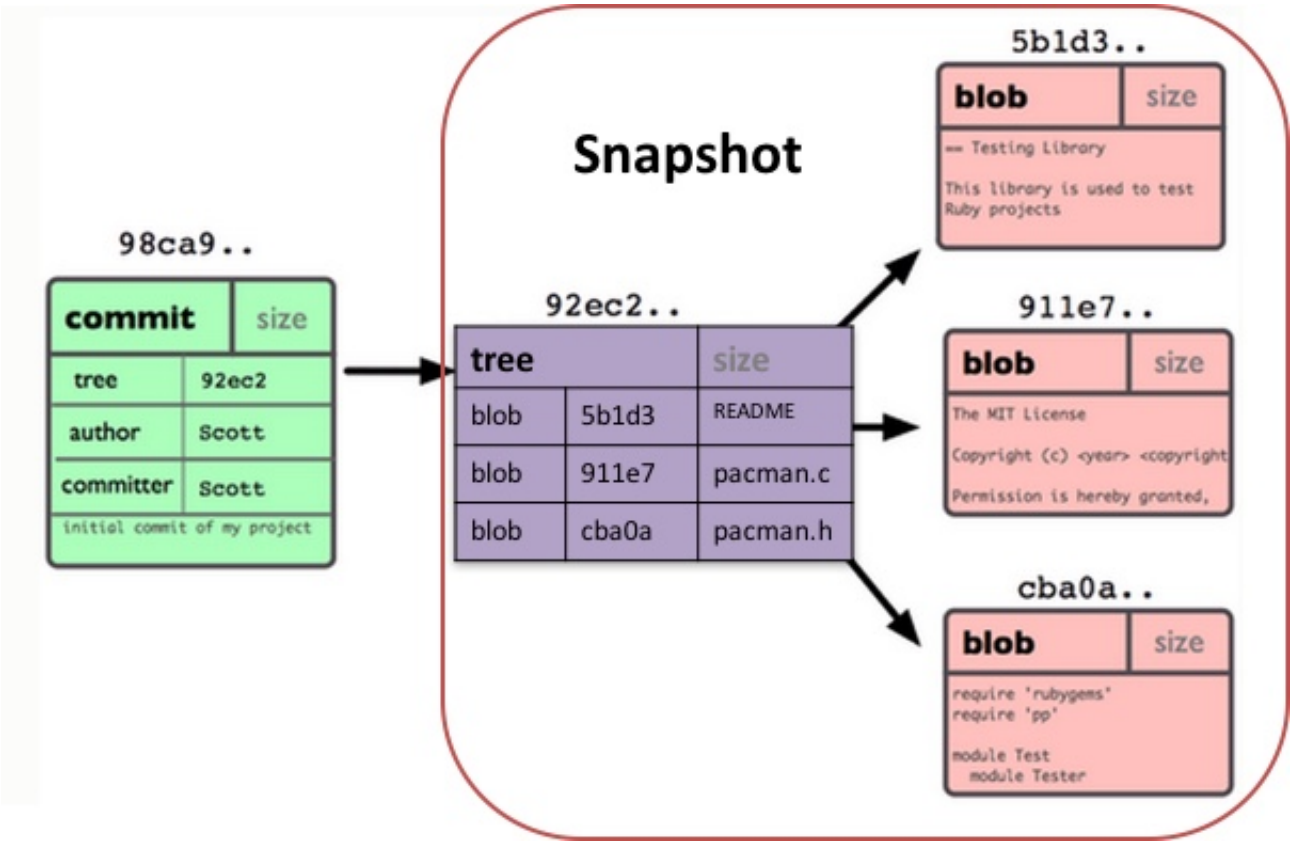
| | |
|---|---|
| | |

| Command | Action |
| --- | --- |
| `$ git checkout HEAD main.cpp` | gets the HEAD revision for the working copy |
| `$ git checkout – main.cpp` | reverts changes in the working directory |
| `$ git revert` | reverts commits (this creates new commits) |
| `$ git clean` | cleans up untracked files |
| `$ git tag –a v1.0 –m 'Version 1.0'` | names the HEAD commit as v1.0 |

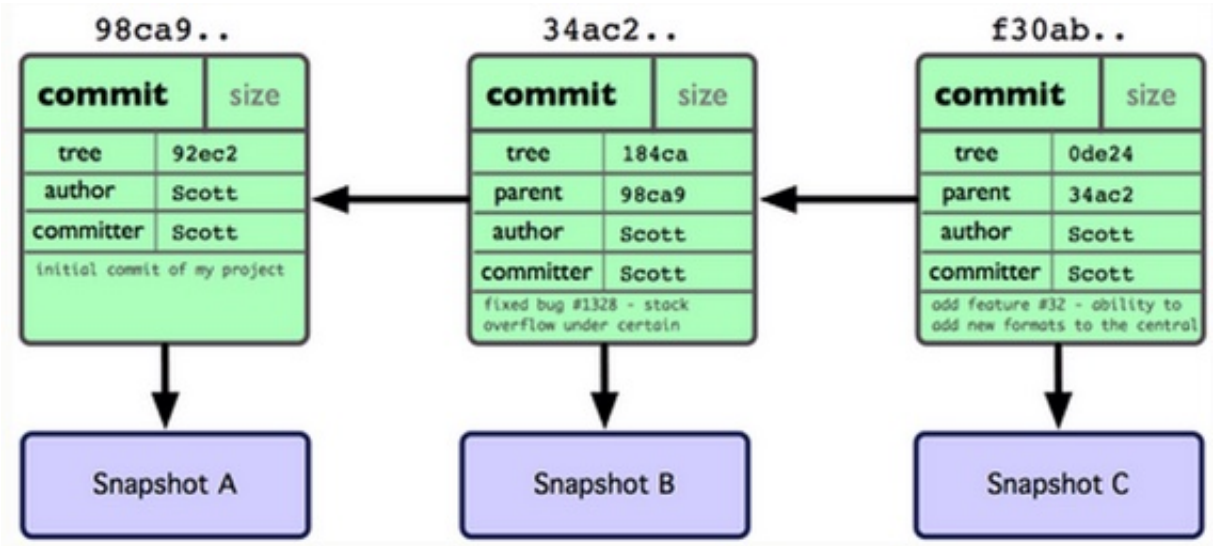
## Git File Status Lifecycle




## Git Repo Structure



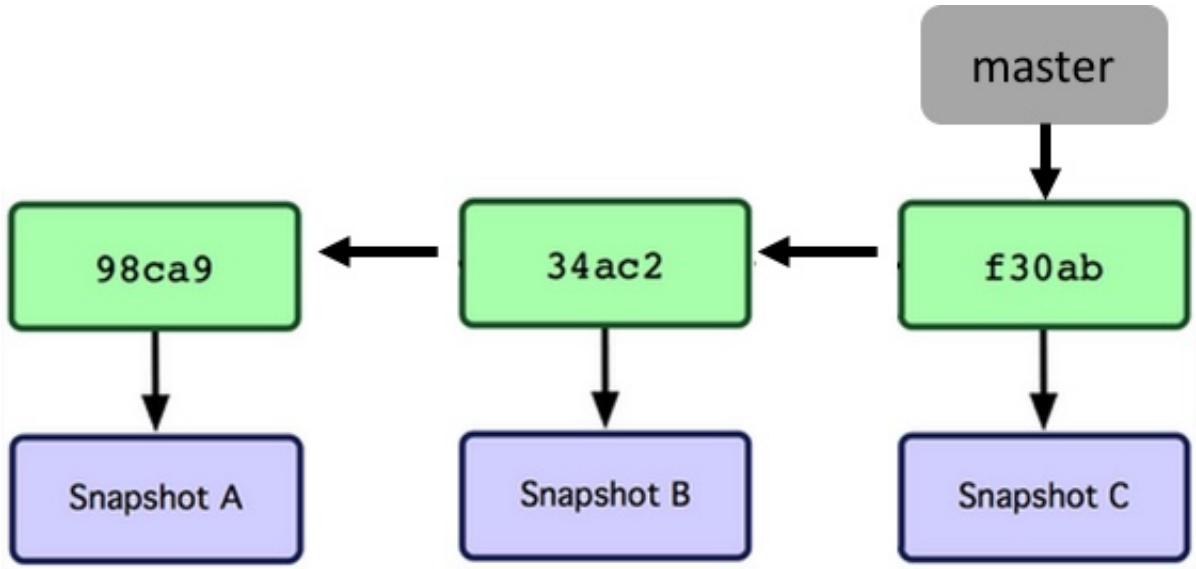

**After two more commits…**




## What is a Branch?

- a pointer to one of the commits in the repo (head) + all ancestor commits
- when you first create a repo, are there any branches?
    - default branch named “master”
- the default master branch
    - points to last commit made
    - moves forward automatically, every time you commit

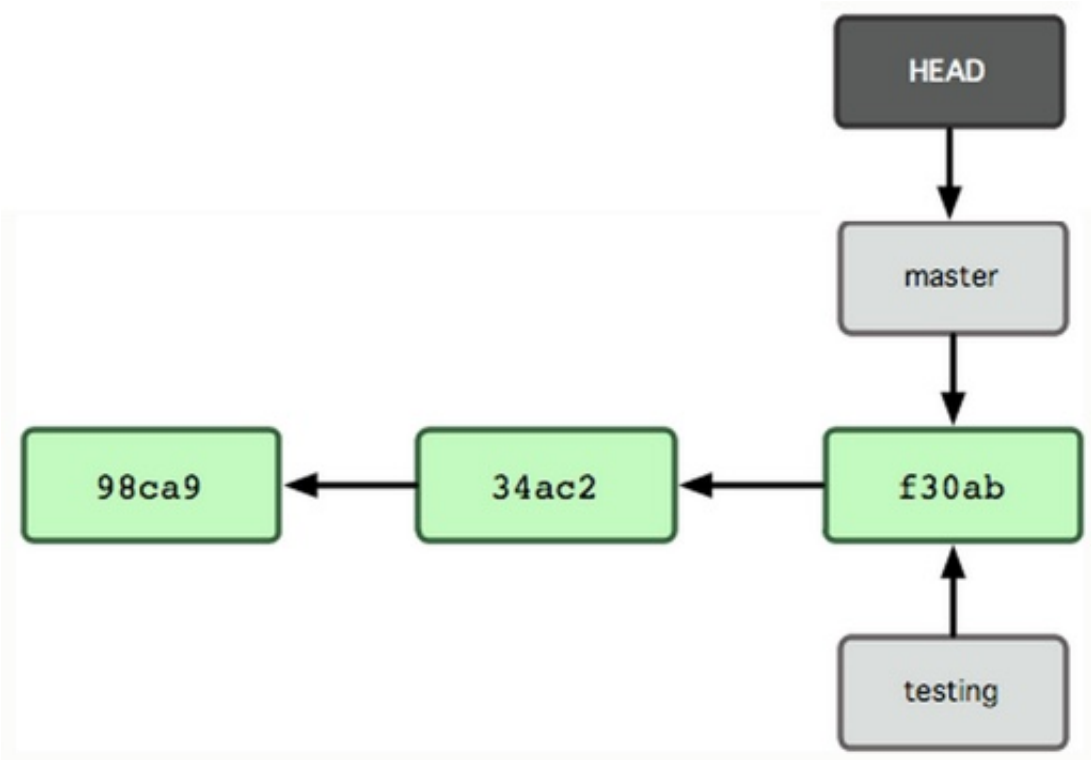
## Where is Master?




- as new commits to master are added, the master pointer follows along
  - 98ca9 ➤ 34ac2 ➤ f30ab

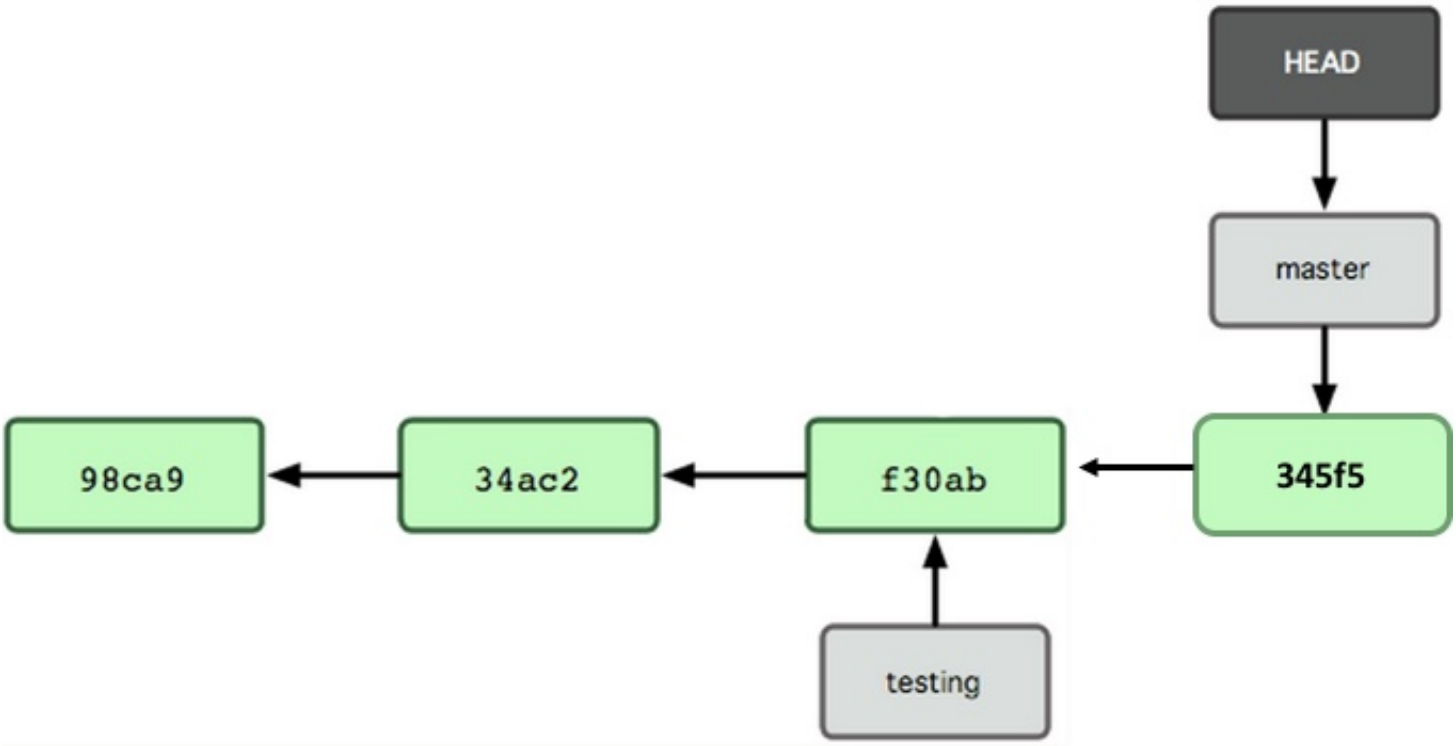
## New Branch

- creating a new branch **creates a new pointer**
  - `$git branch testing`
  - new branch, *testing*, created pointing to same commit that the branch at **HEAD** is pointing to



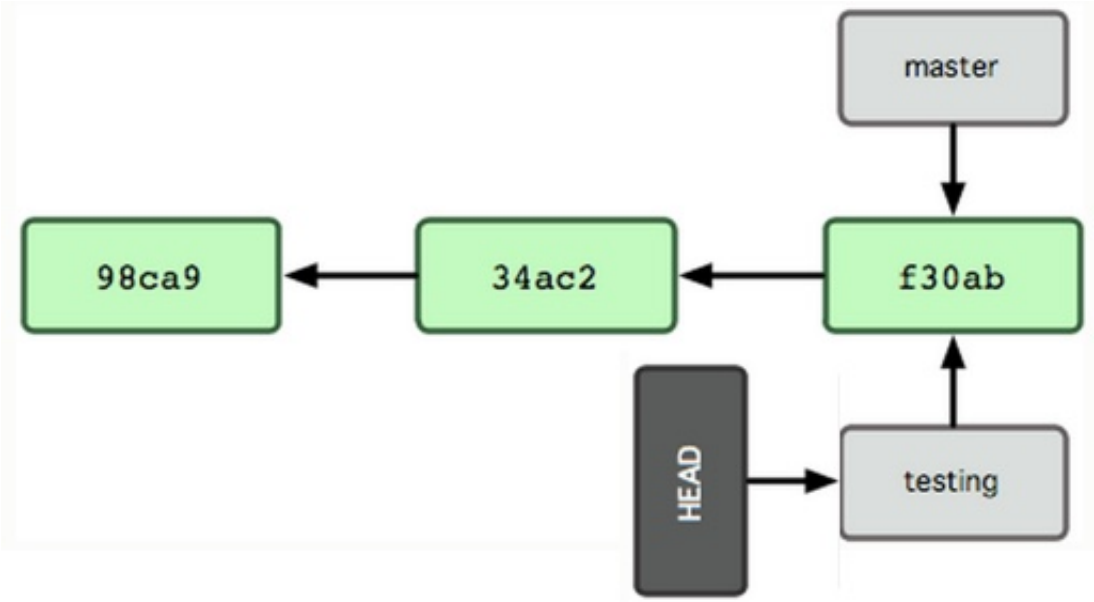
## New Commit

- What happens when we make another commit to **master**?
  - master will move to `345f5` while “testing” branch stays put
  - only master moves because that is the current branch (HEAD)

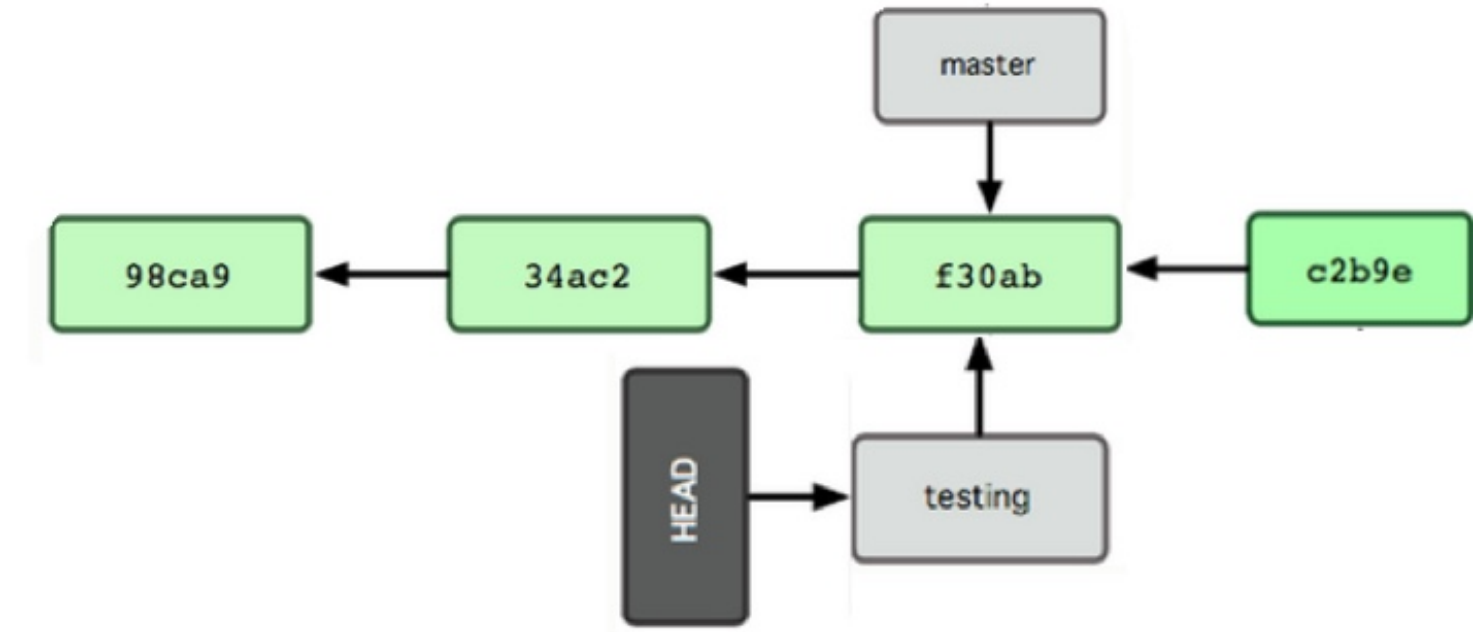



## Switching to New Branch

- check out new branch
  - `$ git checkout <branch_name>`
  - `$ git checkout testing`
- moves **HEAD** to point to *testing* branch




## Commit After Switch




## Why is Branching So Useful?

- experiment with code without affecting main branch
- separate projects that once had a common code base
- 2 versions of the project
- branching is very cheap