



# The Git Parable

Johan Herland  
@jherland  
<[jherland@cisco.com](mailto:jherland@cisco.com)>  
<[johan@herland.net](mailto:johan@herland.net)>

Bio:

My name is Johan Herland

Software developer at Cisco Systems Norway since 2011.  
Working on Collaboration and Videoconferencing products.  
Before: Worked at Opera Software for 7 years.  
Meddled with Git and DVCS since 2006.  
At Opera, I headed up the migration from CVS to Git.  
Was involved in migration from SVN to Git at Cisco.  
I have occasionally contributed code to Git itself.

Can everybody hear me?

Can everybody see the presentation?

Stop me any time if I'm going too fast, or there's something that doesn't make sense.  
This is just a starting point. If you got questions and we start wandering off, that's OK too.



# The Git Parable

- Shamelessly stolen from Tom Preston-Werner
  - <http://tom.preston-werner.com/2009/05/19/the-git-parable.html>
- I'm lazy...
- Also: Best introduction to Git I've found so far
  
- Fair warning:
  - Google Docs supports animated slide transitions
  - I made these slide *very* late in the evening...

The following story is shamelessly stolen from Tom Preston-Werner.

He has written this story, and my only contribution is adapting it to this presentation format.

I'm doing this because I'm lazy.

But also because this is the best introduction to Git I have found so far.

# Git – simple & powerful

Git - simple and powerful system



Often, people try to teach Git by demonstrating a few dozen commands, and then yelling...



TADAAA!





I don't believe this is the best way to teach Git.

Sure, it lets you use Git to perform simple tasks

But the Git commands will still feel like magical incantations

Doing anything out of the ordinary will be terrifying.


Until you understand basic concepts, you'll feel like a stranger in a foreign land.



So instead, I will tell you a story...



# parable

/ˈpərəb(ə)/ 

**noun**

noun: **parable**; plural noun: **parables**

a simple story used to illustrate a moral or spiritual lesson, as told by Jesus in the Gospels.

"the parable of the blind men and the elephant"

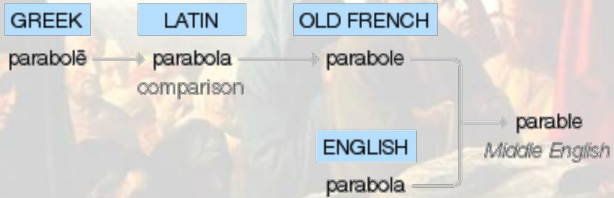
*synonyms*: allegory, moral story, moral tale, fable, lesson, exemplum; More

Haggadah;

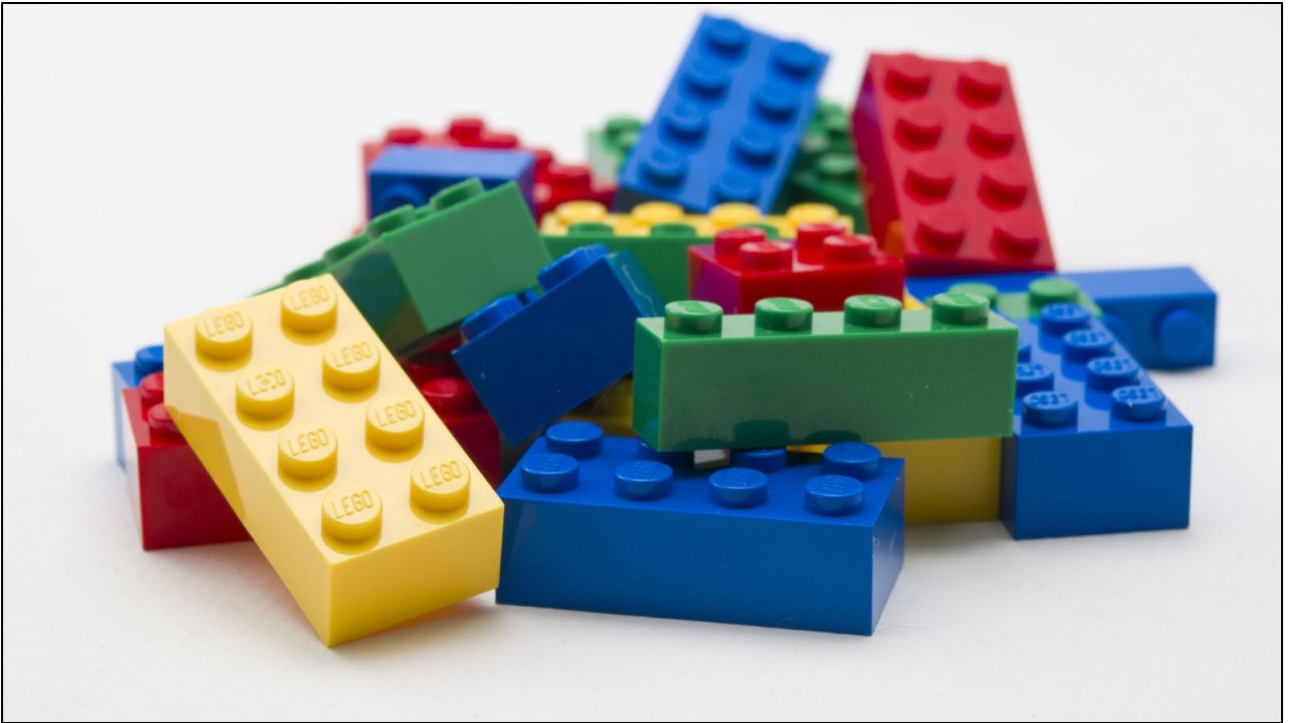
*rare* apologue

"the parable of the prodigal son"

Origin



In fact, I'm going to tell you a parable.



Journey through the creation of Git-like system

From the ground up



Understanding concepts,  
\*most\* valuable thing to fully grok Git

Concepts are simple,  
\*but\* allow for a lot of amazing functionality

After this parable, have everything you need to

- easily master the various Git commands
- become a Git power user



# The Parable

- A simple computer
  - A text editor
  - A few filesystem commands



Imagine:

simple computer with absolutely nothing but:

- a text editor
- some simple filesystem commands

# The Parable

- Write a large software program



Imagine:

Decided to write a large software program on this system

# The Parable

- Write a large software program
- Invent some method to keep track of versions
  - retrieve code that you changed/deleted



Responsible software developer →

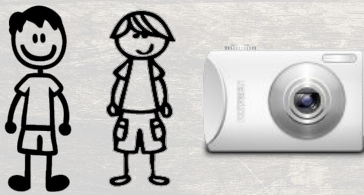
Must keep track of versions  
so that you can retrieve code that you previously changed or deleted.

What follows: A story about how you might design this version control system, and the reasoning behind those design choices.



# Snapshots

- Alfred, the photographer



But, before we start the software project...

Alfred:

- friend of yours
- works as a photographer

# Snapshots

- Alfred, the photographer



All day long he takes awkward family photos.

# Snapshots

- Alfred, the photographer
- Hazel and her daughter

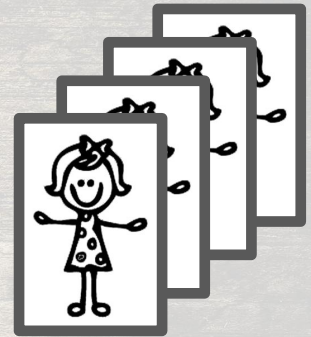


Alfred tells you a story about a woman named Hazel, who brings her daughter in for a portrait \*every\* year on the \*same\* day.



# Snapshots

- Alfred, the photographer
- Hazel and her daughter
  - Remember what the daughter was like at each different stage



“She likes to remember what her daughter was like at each different stage, as if the snapshots really let her move back and forth in time to those saved memories.”

You suddenly see the ideal solution to your version control dilemma:

## Snapshots

- like save point in a video game
- what you care about in your VCS.

What if you could take snapshots of your codebase at any time, and resurrect that code on demand?

You go back to your computer and start working...

# Snapshots



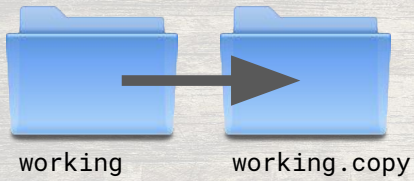
working



Start project in directory named “working”.

Try to write one feature at a time.

## Snapshots

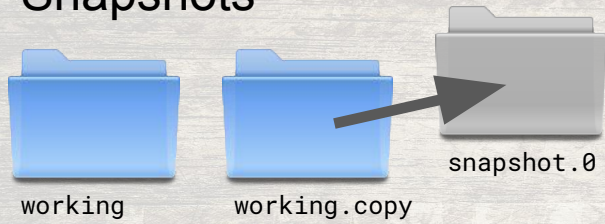


Complete a self-contained portion of a feature →

Duplicate the entire working directory, to *\*archive\** the current state.



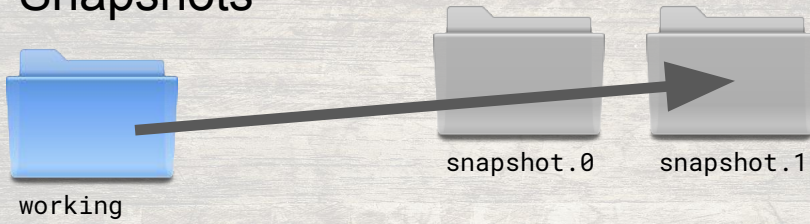
## Snapshots



Going to make *\*more\** copies of the working directory, so you rename the copy to “snapshot.0”.

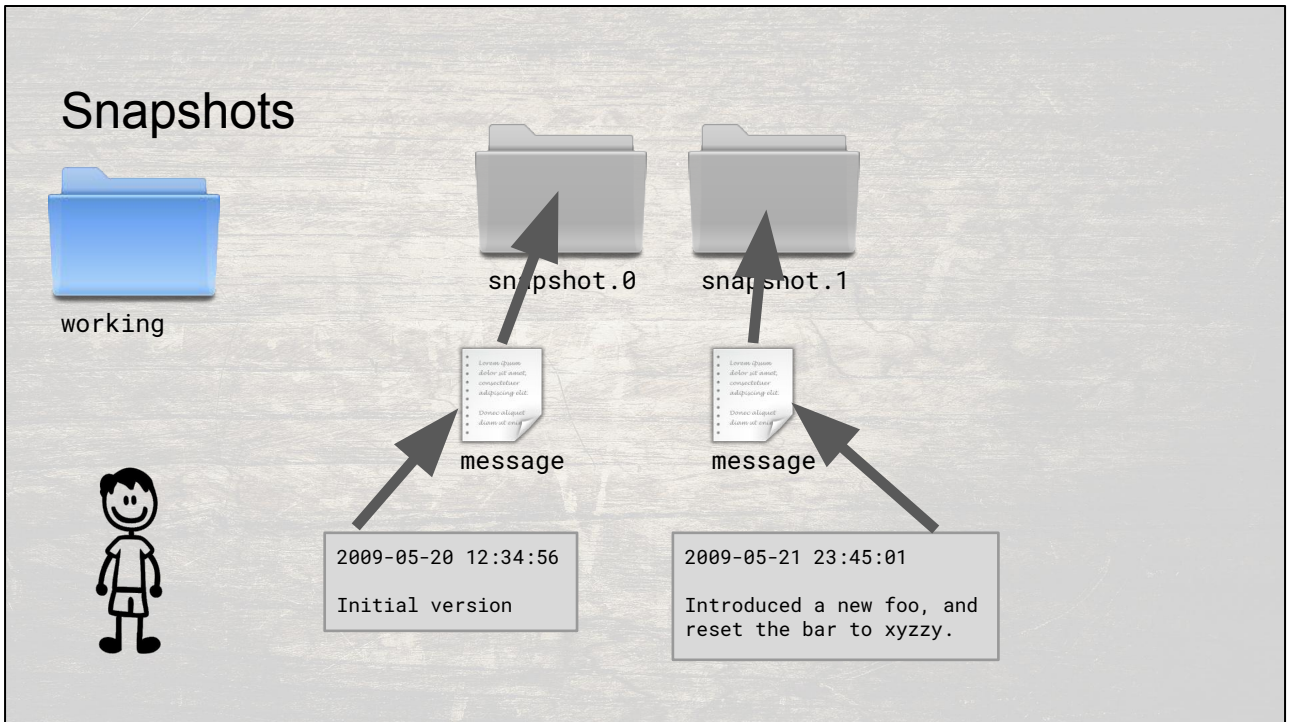
After: *\*never\** again change code in “snapshot.0”

# Snapshots



After the \*next\* chunk of work:

Another copy → name “snapshot.1”, and so on.



Remember the changes made in each snapshot → add a special file named “message” to each snapshot directory.

Contains:

- Summary of the work that you did
- The date you did it

Find a specific change →  
search the message files in previous snapshots

## Branches



working



snapshot.0



snapshot.1

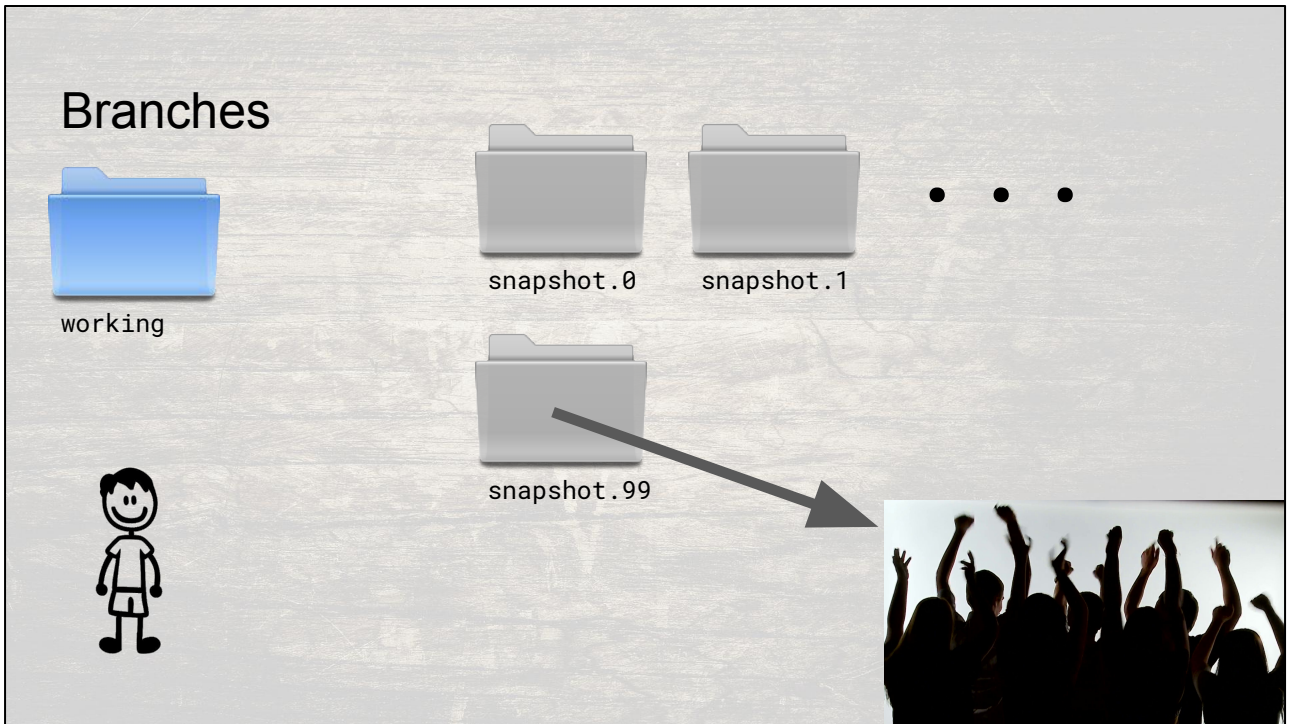


snapshot.99

After some time:

- release candidate starts to form
- decide to release "snapshot.99" as Version 1.0





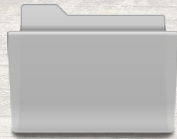
Package “snapshot.99” and distribute as Version 1.0.

Your users love it, and you push forward, determined to make the next version an even bigger success.

# Branches



working

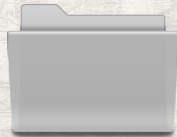


snapshot.0



snapshot.1

...



snapshot.99

...



snapshot.109

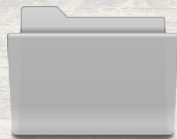
Keep adding new features

Make 10 new snapshots

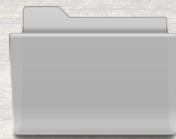
# Branches



working

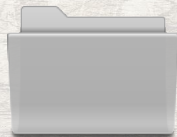


snapshot.0



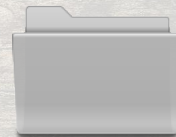
snapshot.1

...



snapshot.99

...



snapshot.109

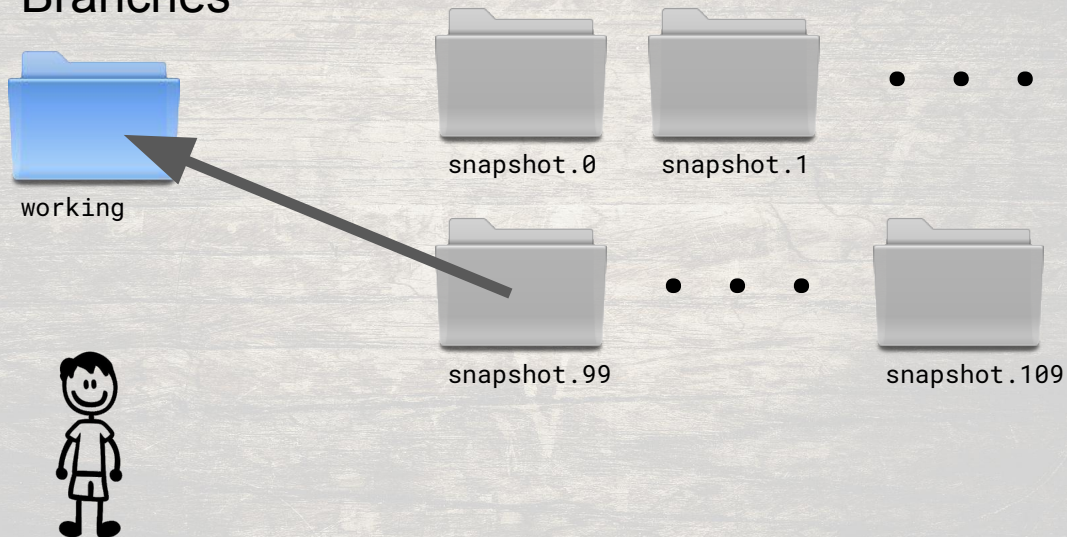


Your VCS: faithful companion.

Old versions are there when you need them & can be accessed with ease.

But not long after the release, bug reports from Version 1.0 start to come in.

## Branches



In order to fix the bugs in Version 1.0:  
copy "snapshot.99" to "working" so that your working directory is at exactly the point  
where Version 1.0 was released.

You then make the required fixes.

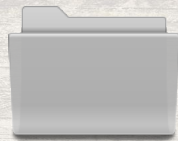
It is here that a problem becomes apparent:



## Branches



working



snapshot.0



snapshot.1

...



snapshot.99

...



snapshot.109



snapshot.110

The VCS deals very well with linear development.

\*but\* for the first time, you need to create a new snapshot that is not a direct descendant of the preceding snapshot.

“snapshot.110” does not follow “snapshot.109”

→ breaks the linear flow

→ cannot determine the ancestry of any given snapshot.

Clearly, you need something more powerful than a linear system.

Time to have a break, and you take a walk outside.

## Branches

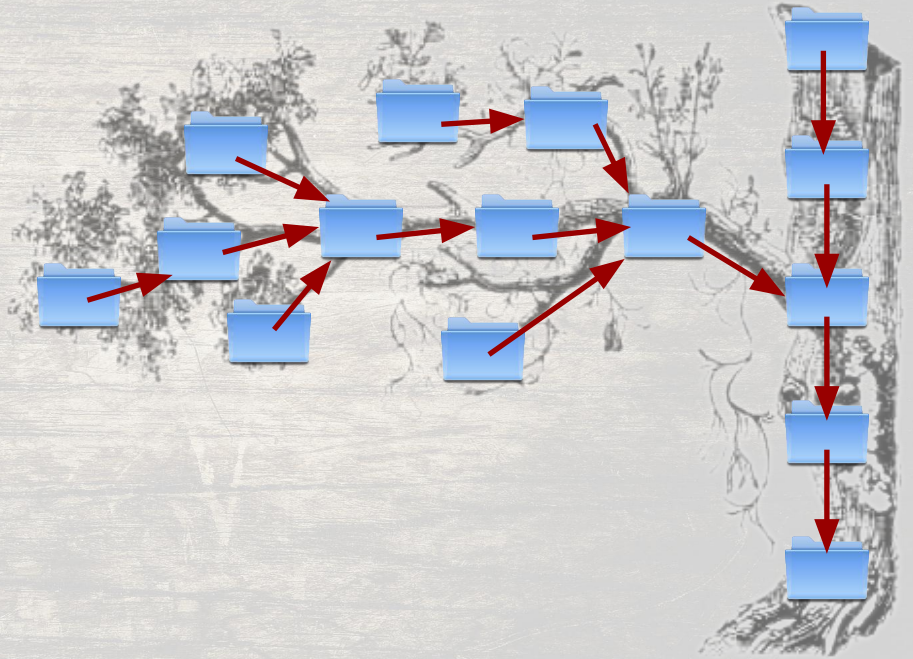
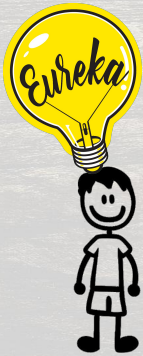


Walk past oak tree, look more closely

Start looking at branch tip, and trace it back to the trunk.

The tree is a very complex structure, but the rules for finding your way back to the trunk are so simple, and perfect for keeping track of multiple lines of development!

## Branches

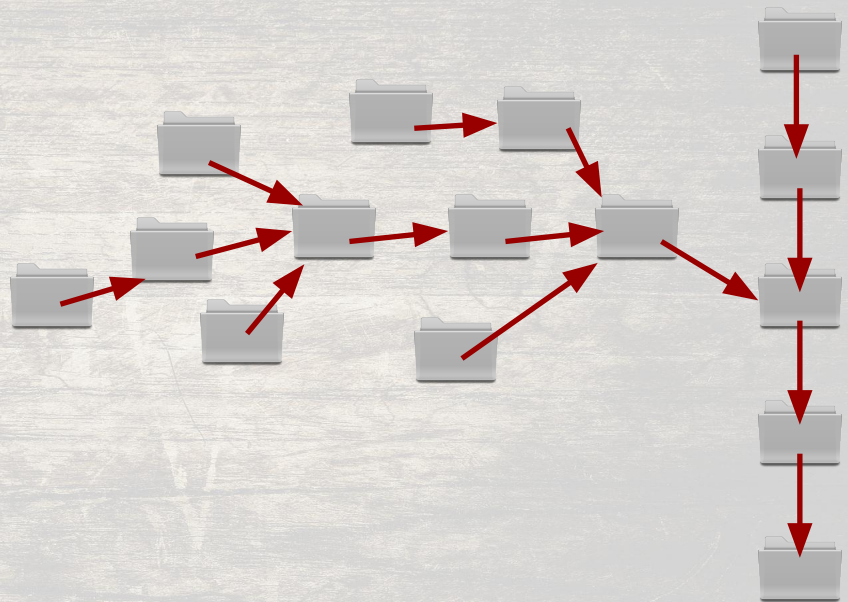


You have another epiphany.

By looking at your code history as a tree, solving the problem of ancestry becomes trivial.



## Branches

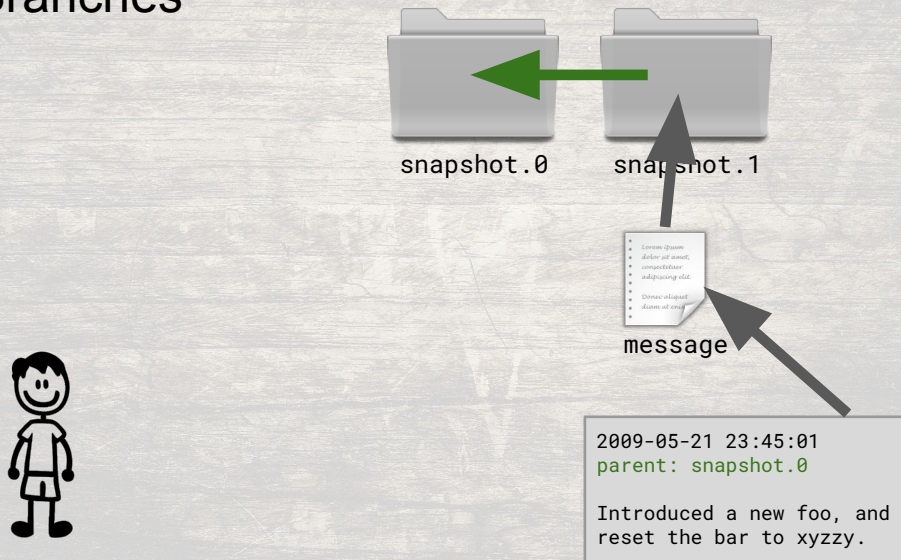


You need each snapshot to point to the previous snapshot, also known as the “parent” snapshot.

Each snapshot has one parent, except for the very first snapshot.



# Branches



So how do you store this pointer to the previous snapshot?

Include the name of the parent snapshot in the “message” file you write for each snapshot.

## Branch Names



working



snapshot.110



snapshot.109



snapshot.99



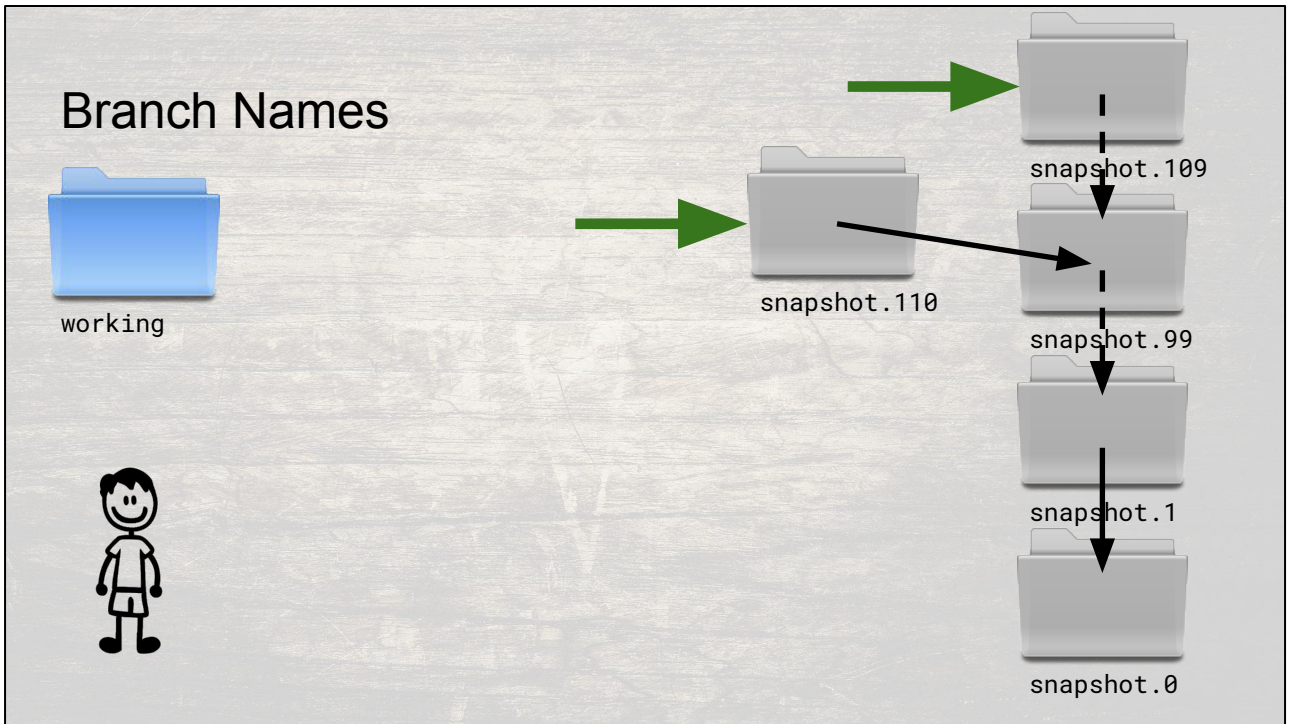
snapshot.1



snapshot.0

That pointer enables you to easily and accurately trace the history of any given snapshot all the way back to the root.

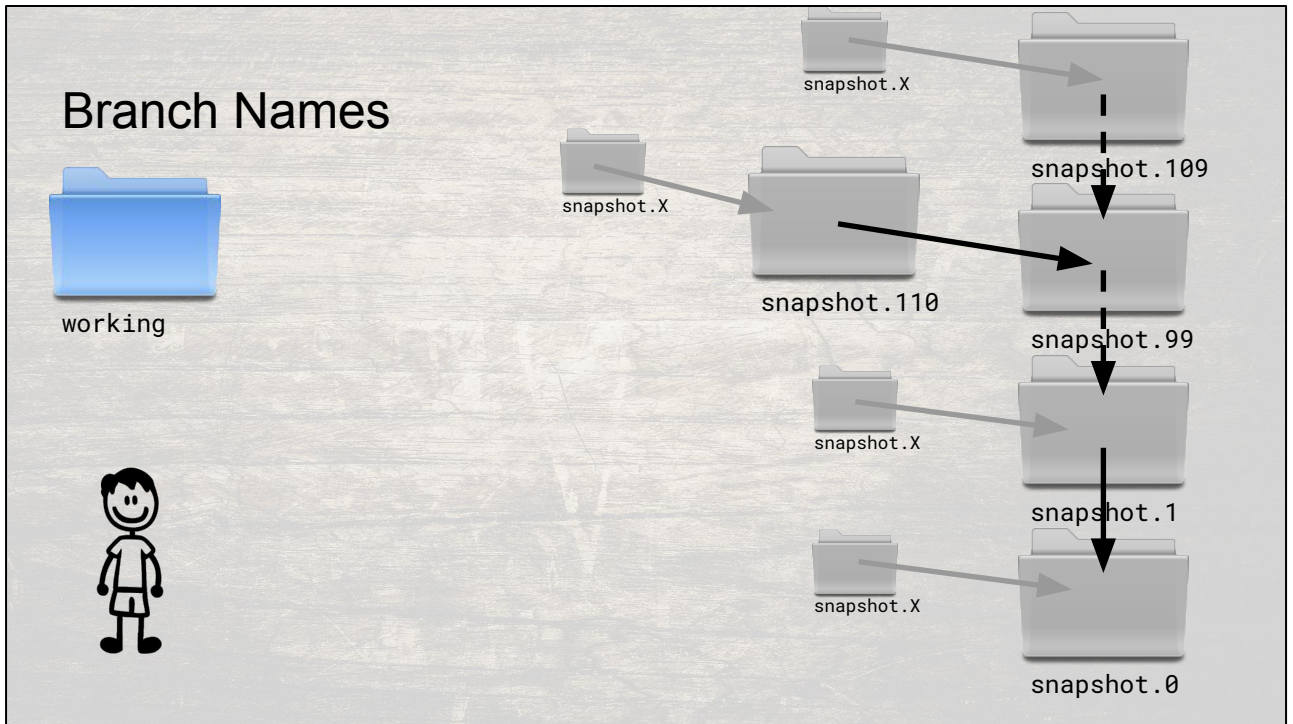
Your code history is now a tree.



Instead of having a single latest snapshot, you have *\*two\**: one for each branch.

non-linear history → sequential numbering system is less useful:

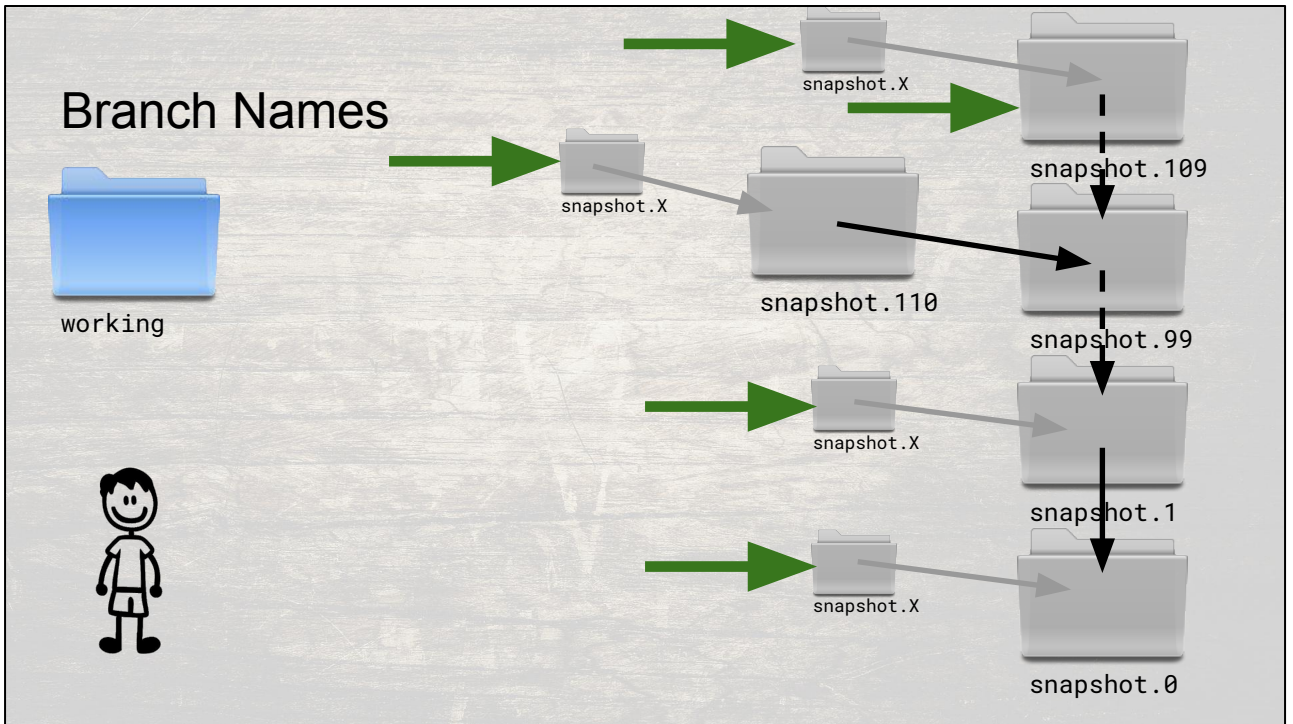
You cannot easily determine the latest snapshot, because you have *\*multiple\** latest snapshots, one for each branch.



However, Creating new development branches has become so simple that you'll want to take advantage of it all the time.

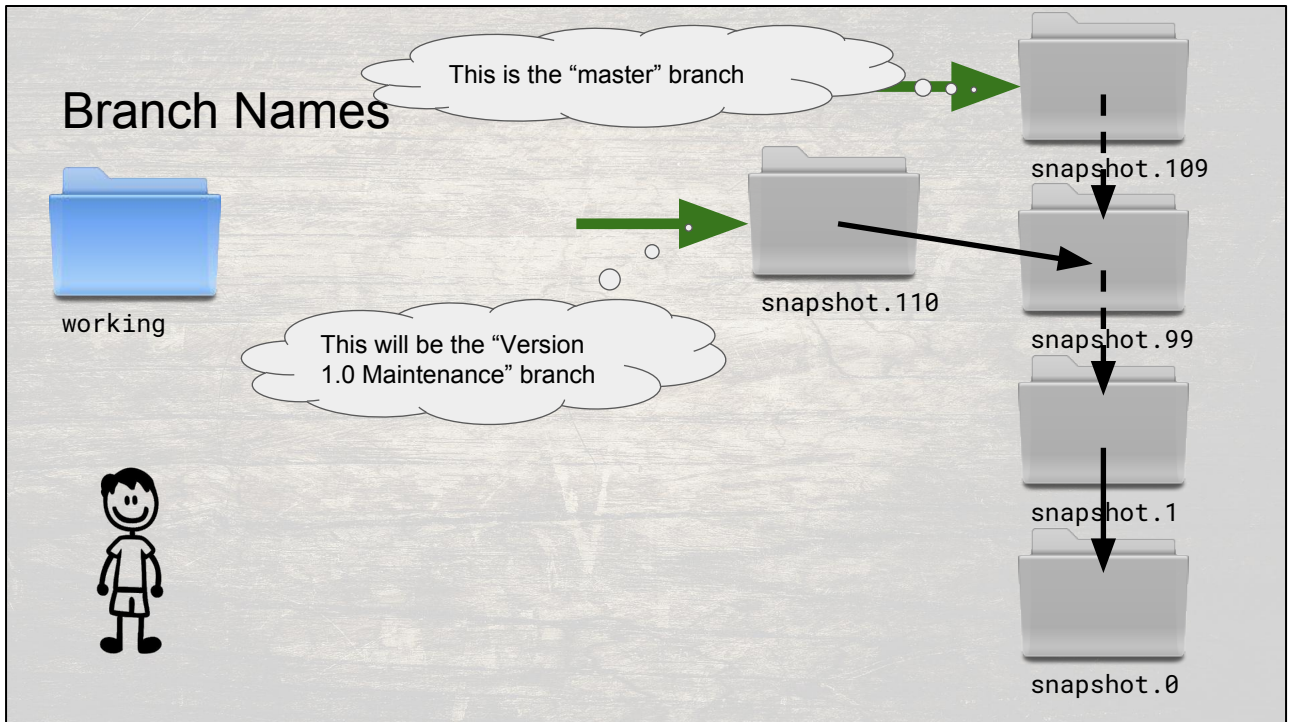
You'll be creating branches for fixes to old releases, for experiments that may not pan out; indeed it becomes possible to create a new branch for every feature you begin!





But like everything good in life, there is a price to be paid:

You need some way to keep track of these branches, and which snapshots belong on which branch.

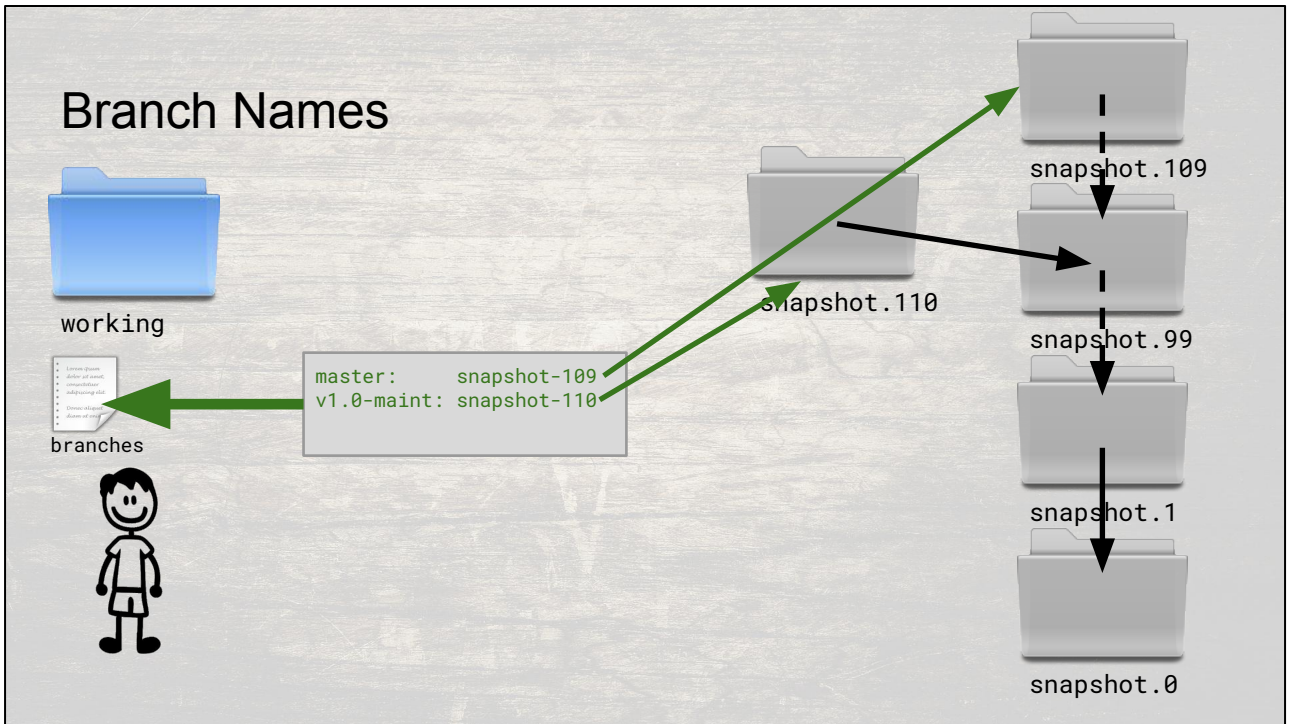


Every time you create a new branch you probably give it a name in your head.

(Identify the two branches)

Now, to find the snapshots on a branch, you only need to remember the *\*latest\** snapshot on that branch.

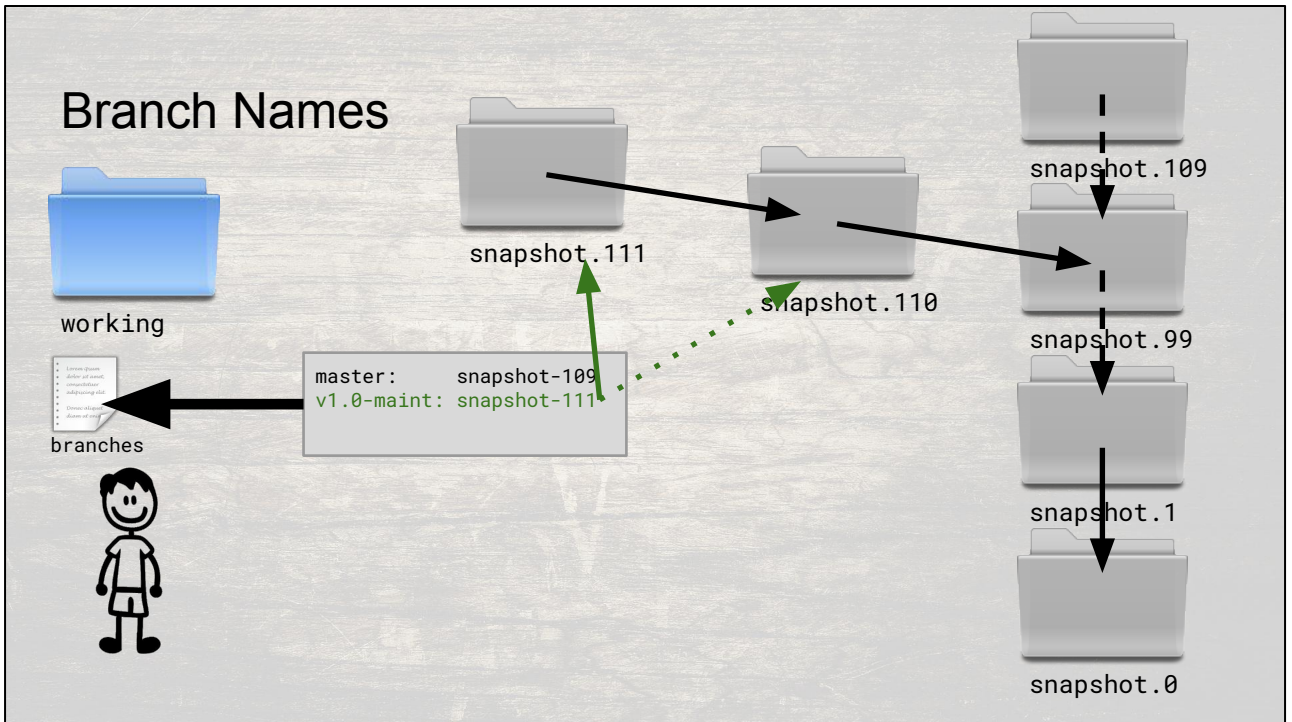
The earlier snapshots is found by following the parent pointer from each snapshot.



So, how do we store the link between branches and their latest snapshots?

Simple: In a file named “branches”, stored *outside* of any specific snapshot, you simply list the name/snapshot pairs that represent the tips of your branches.

To switch to a given branch you look up the snapshot for that branch in this file, and then copy that snapshot directory into the working directory.

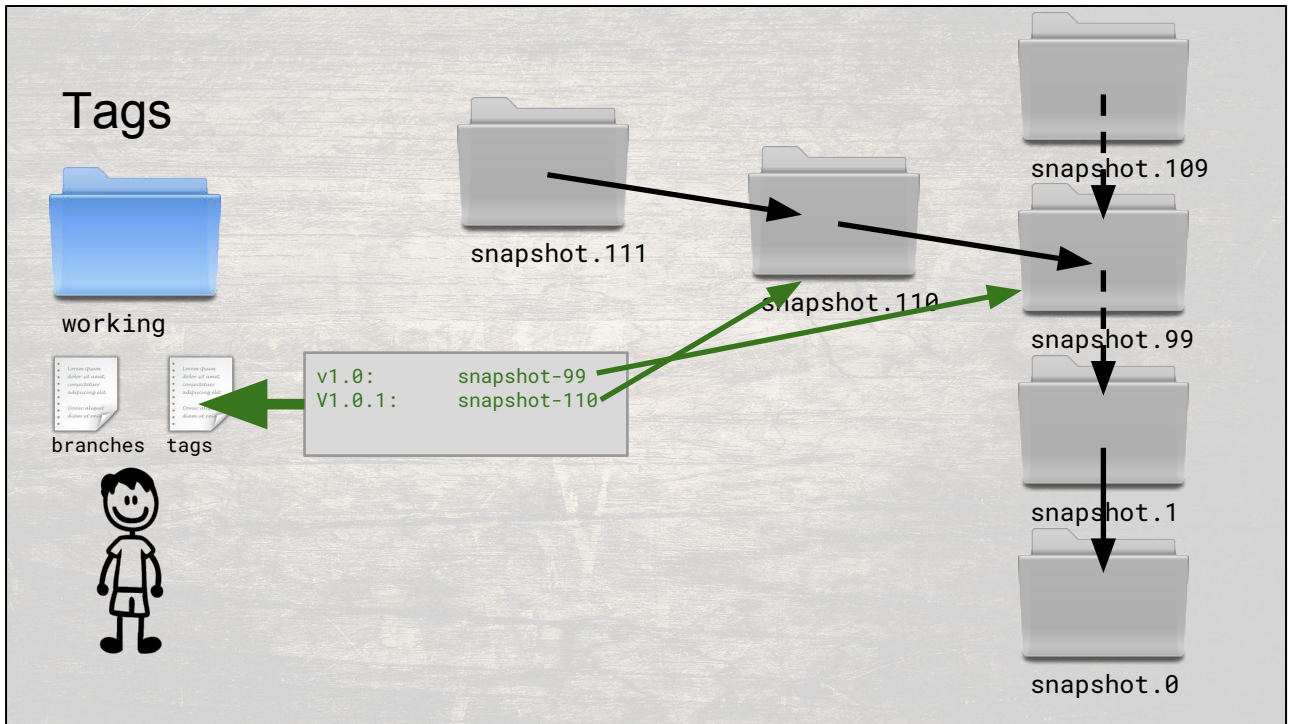


Because you're only storing the latest snapshot on each branch, creating a new snapshot now contains an additional step:

If the new snapshot is being created as part of a branch, the "branches" file must be updated so that the name of the branch becomes associated with the new snapshot.

A small price to pay for the benefit.





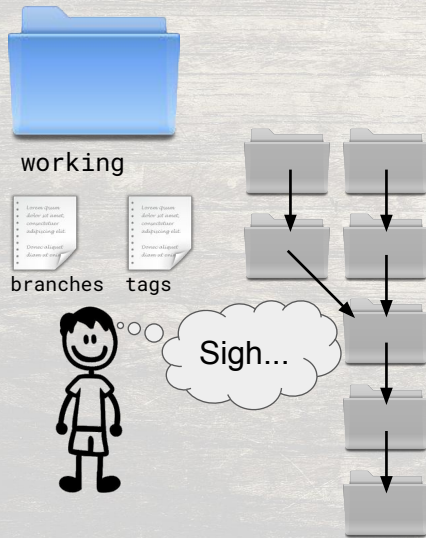
After using branches for a while you notice that you also need a *\*different\** kind of reference:

You want a reference that *\*always\** points to a certain snapshot, and *\*never\** moves.

You want to use these special references for labeling *\*certain important snapshots\**, like "Version 1.0" and "Version 1.0.1".

Since they don't behave like regular branches, you decide to call them "tags", and store them in a separate file, so that you don't accidentally treat them as a regular branch.

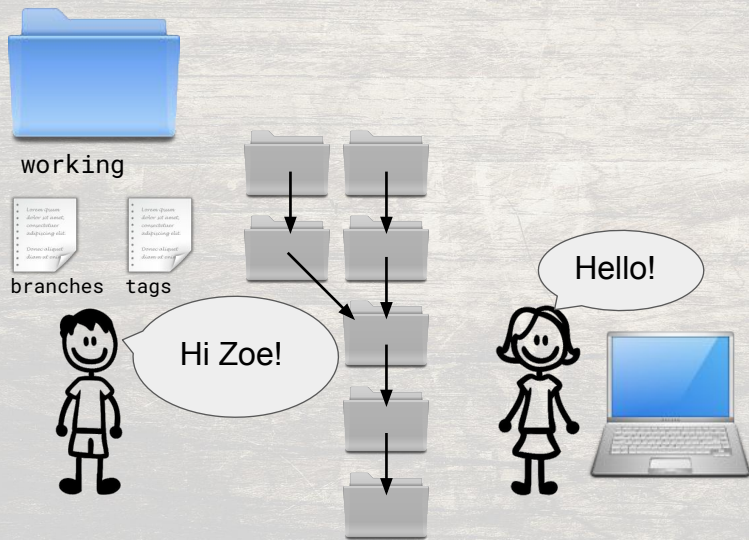
# Distributed



Working on your own gets pretty lonely.

Wouldn't it be nice if you could invite a friend to work on your project with you?

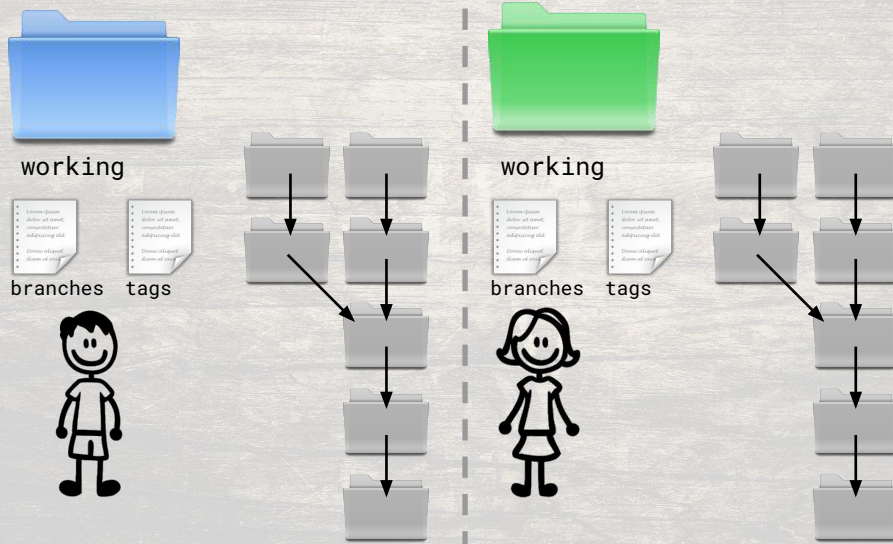
# Distributed



Well, you're lucky.

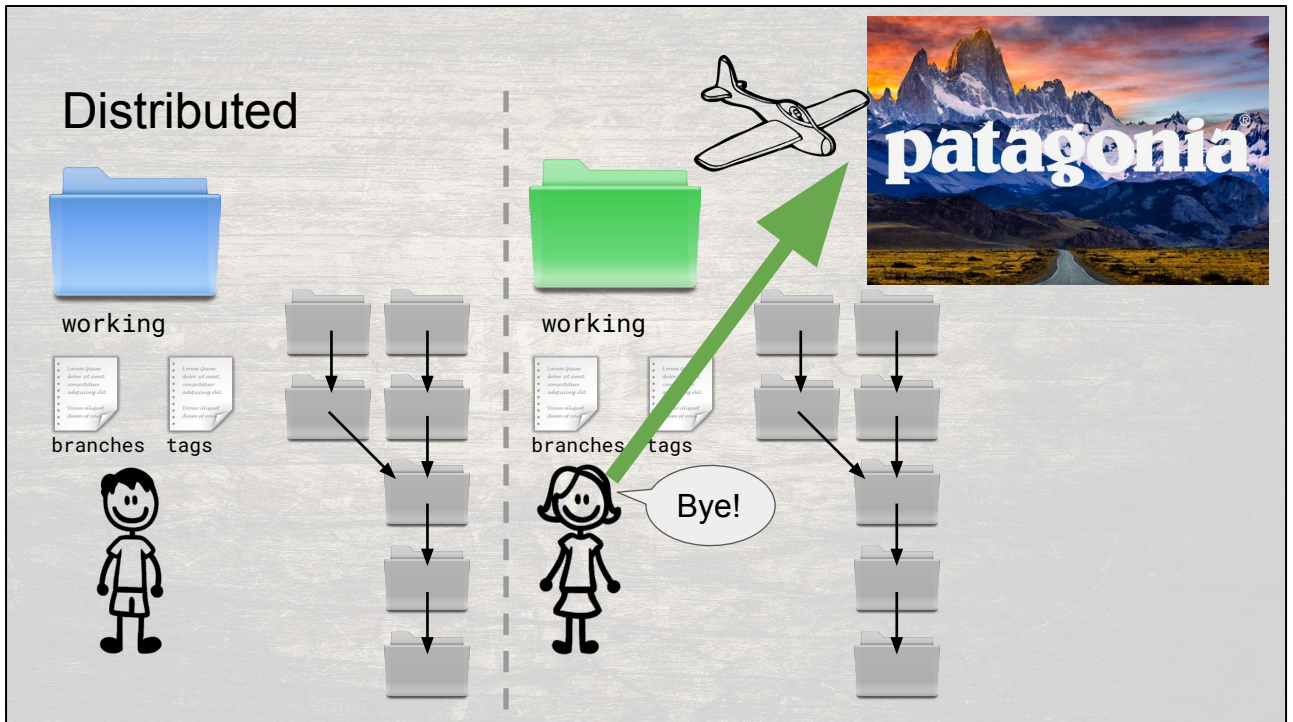
Your friend Zoe has a computer setup just like yours and wants to help with the project.

# Distributed



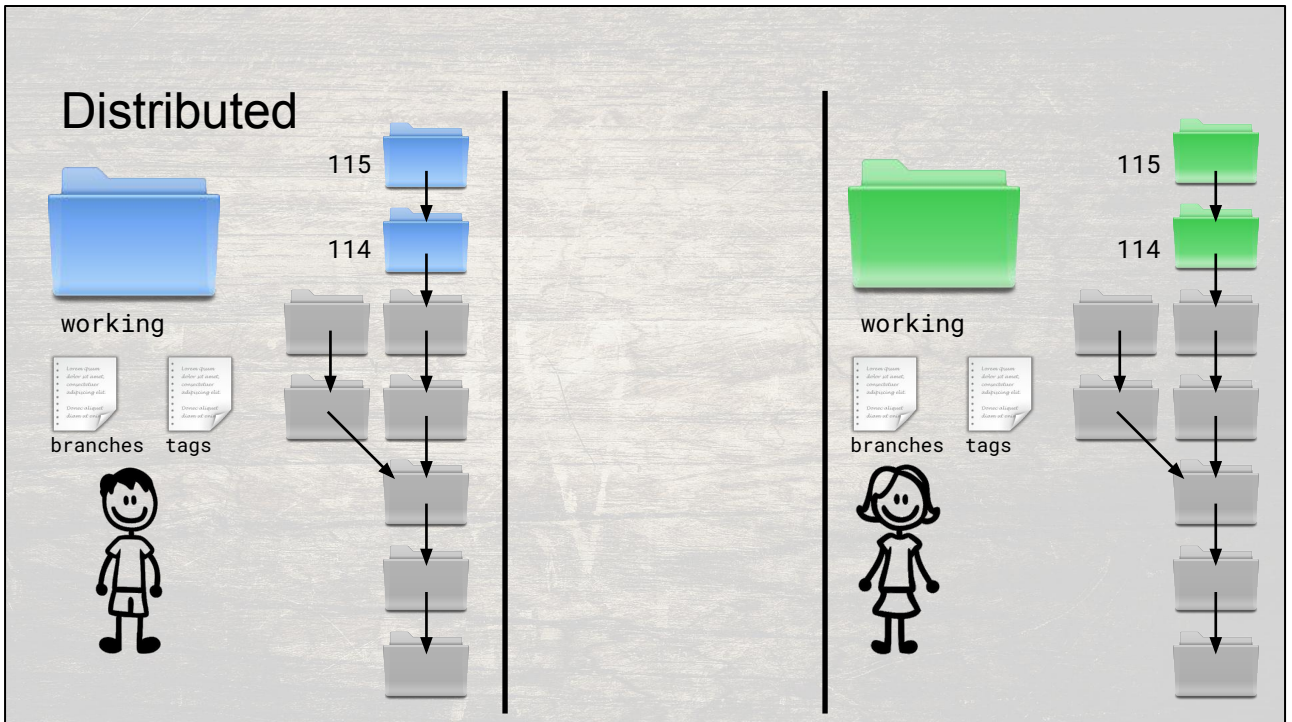
Because you've created such a great version control system, you tell her all about it and send her a copy of all your snapshots, branches, and tags so she can enjoy the same benefits of the code history.





It's great to have Zoe on the team but she has a habit of taking long trips to far away places without internet access.

As soon as she has the source code, she catches a flight to Patagonia and you don't hear from her for a week.



In the meantime you both do a lot of coding.

When she finally gets back, you discover a critical flaw in your VCS.

Because you've both been using the same numbering system, you each have directories named "snapshot.114", "snapshot.115", and so on, but with different contents!

To make matters worse, you don't even know who authored the changes in those new snapshots.

Together, you devise a plan for dealing with these problems.

## Distributed



snapshot.114



message

2009-05-22 12:12:12  
parent: snapshot.113  
author: Me <me@me.me>

Blarfle, a cool new feature;  
extends the existing blorg.



snapshot.114



message

2009-05-23 23:22:21  
parent: snapshot.113  
author: Zoe <zoe@z.oe>

Frotzed the blarglefoo  
and the garglezoat.

First, you change the snapshot messages to contain author name and email.

## Distributed



Second, we should not name snapshots with simple numbers.

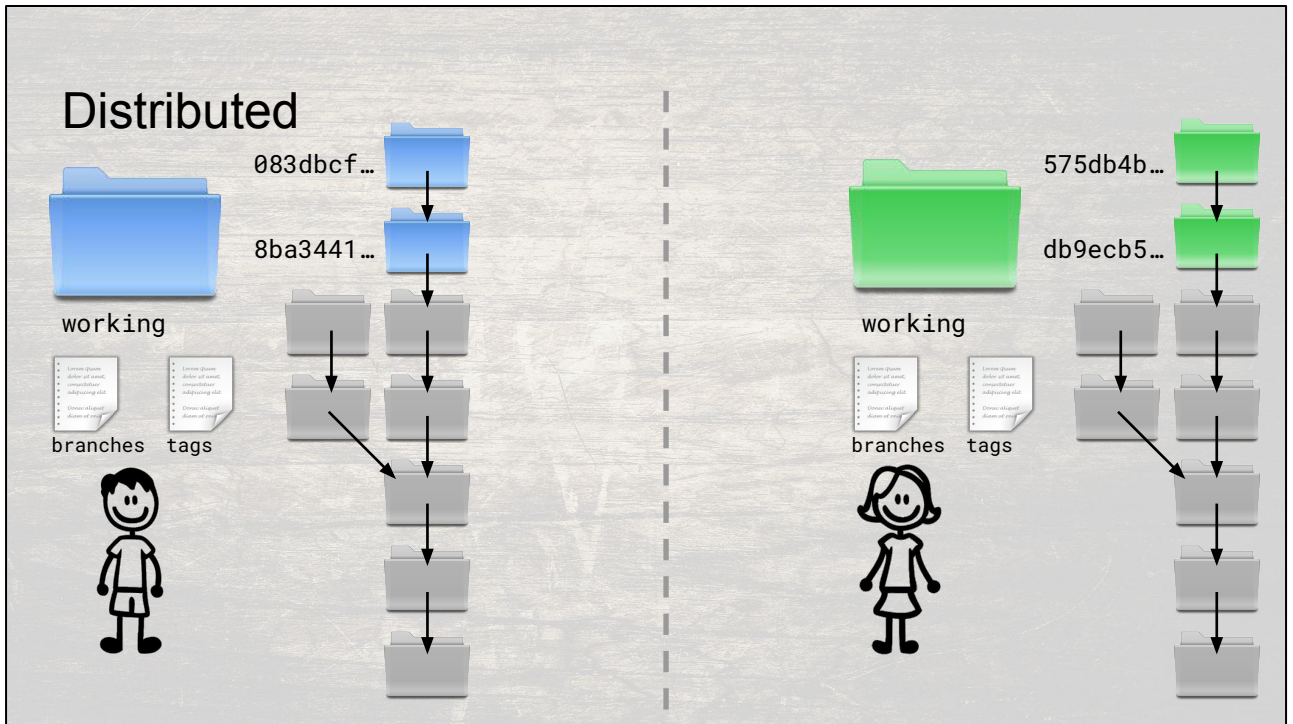
Instead, use contents of “message” file and the SHA1 algorithm to produce a hash.

Do you know SHA1? (Maybe talk about SHA1)

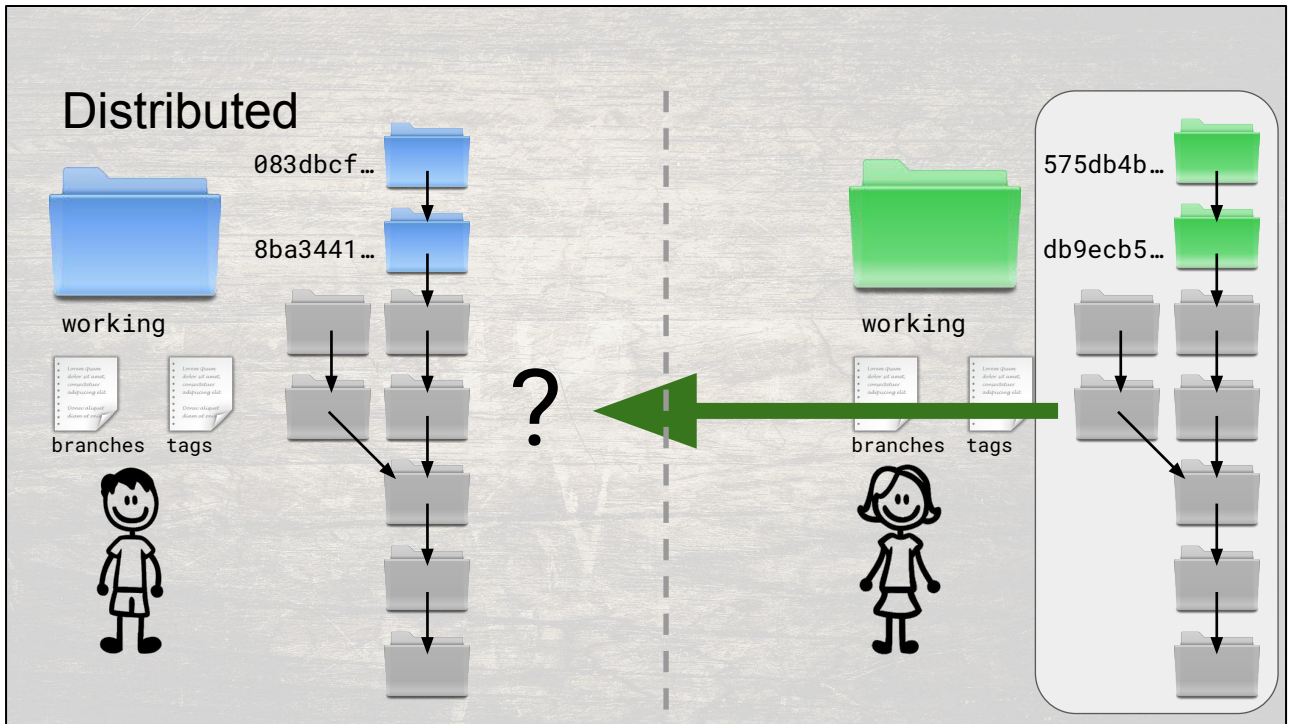
This hash will be unique to the snapshot since no two messages will ever have the same date, message, parent, and author.

You both update your histories with the new technique and instead of clashing “snapshot.114” folders, you now have distinct folders named “8ba3441...” and “db9ecb5...”.

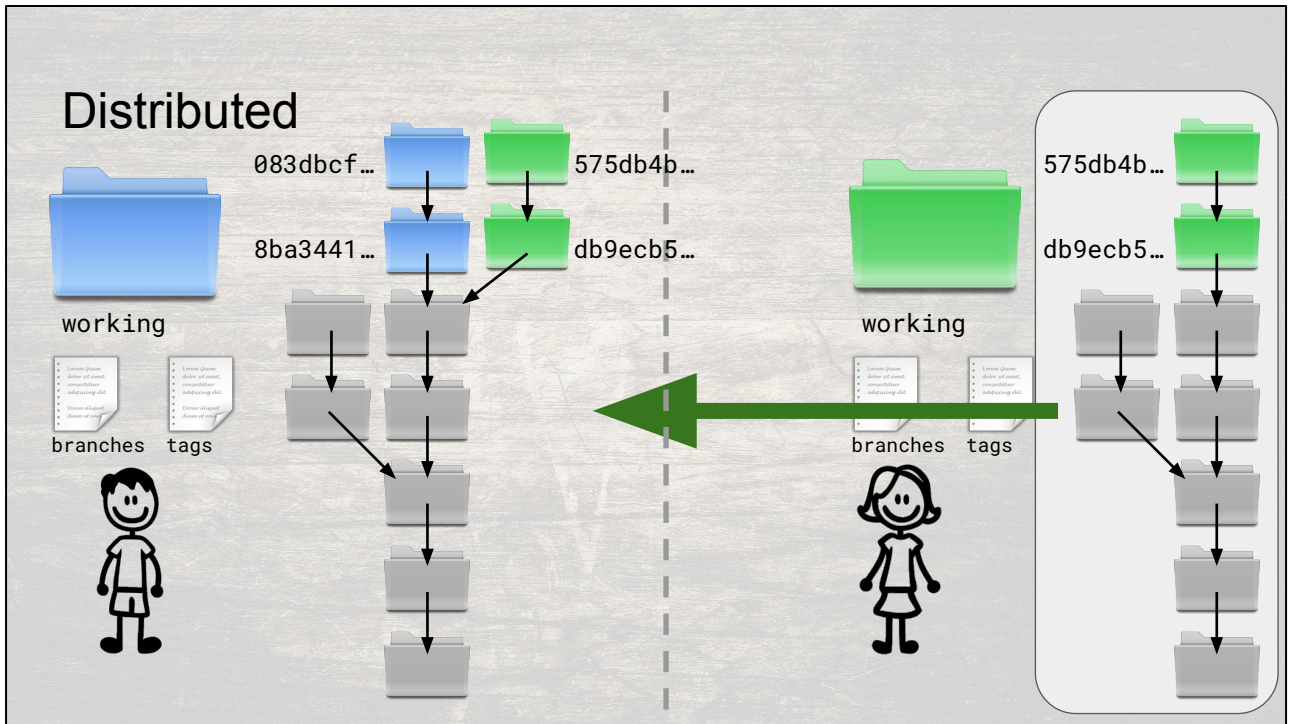




You do this process for all the other snapshots as well...

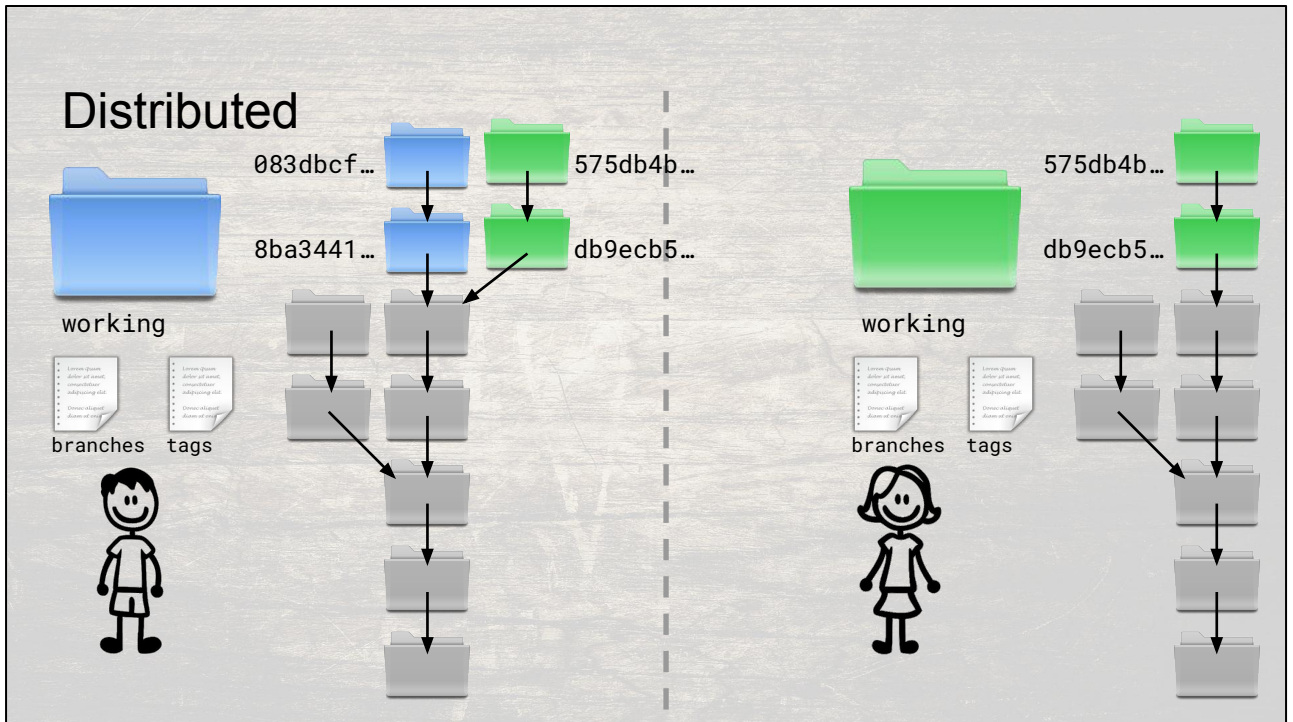


With the updated naming scheme, it becomes trivial for you to fetch all the new snapshots from Zoe's computer and place them next to your existing snapshots.



Because every snapshot specifies its parent, and identical messages (and therefore identical snapshots) have identical names no matter where they are created, the history of the codebase can still be drawn as a tree.

Only now, the tree is comprised of snapshots authored by *both* Zoe and you.



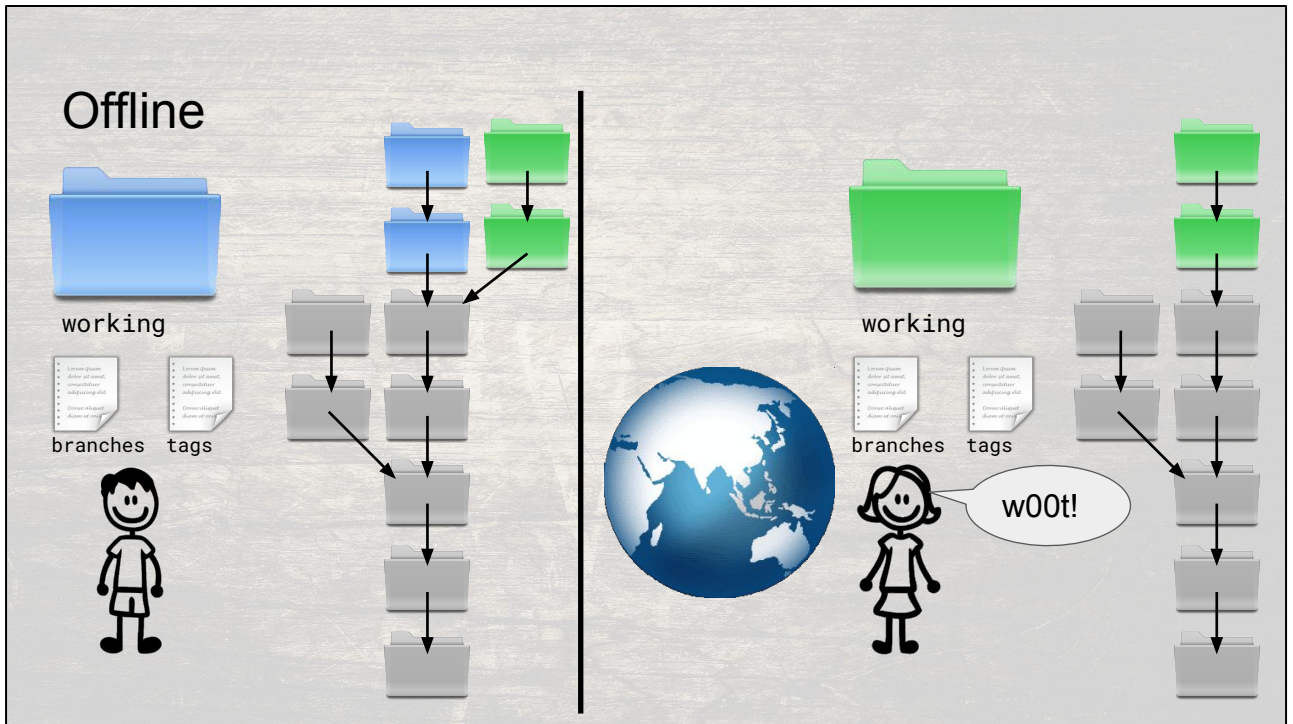
This point is *important*, so I'll repeat it:

A snapshot is identified by the SHA1 sum of the message file that uniquely identifies it (and its parent).

These snapshots can be created and moved around between computers without losing their identity or where they belong in the history tree of a project.

Finally, snapshots can be shared or kept private as you see fit. If you have some experimental snapshots that you want to keep to yourself, you can do so quite easily: Just don't make them available to Zoe!



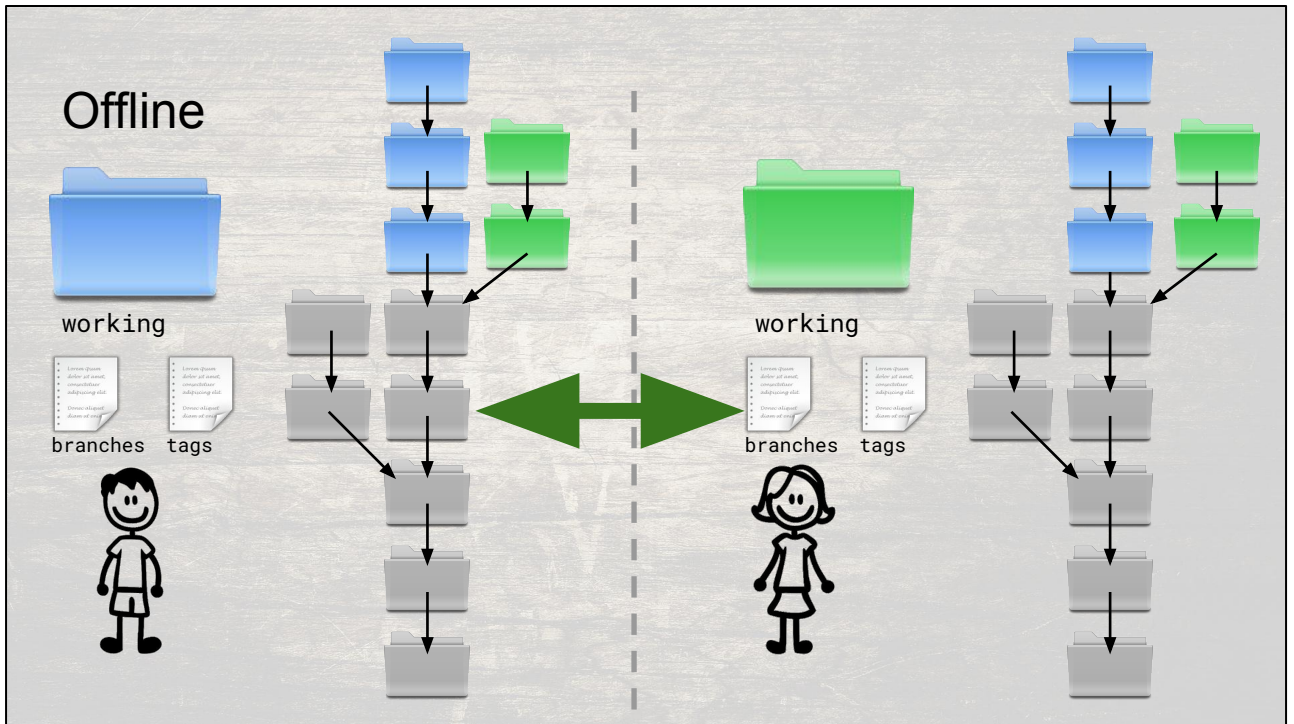


Zoe's travel habits cause her to spend countless hours on airplanes and boats. Most of the places she visits have no readily available internet access:

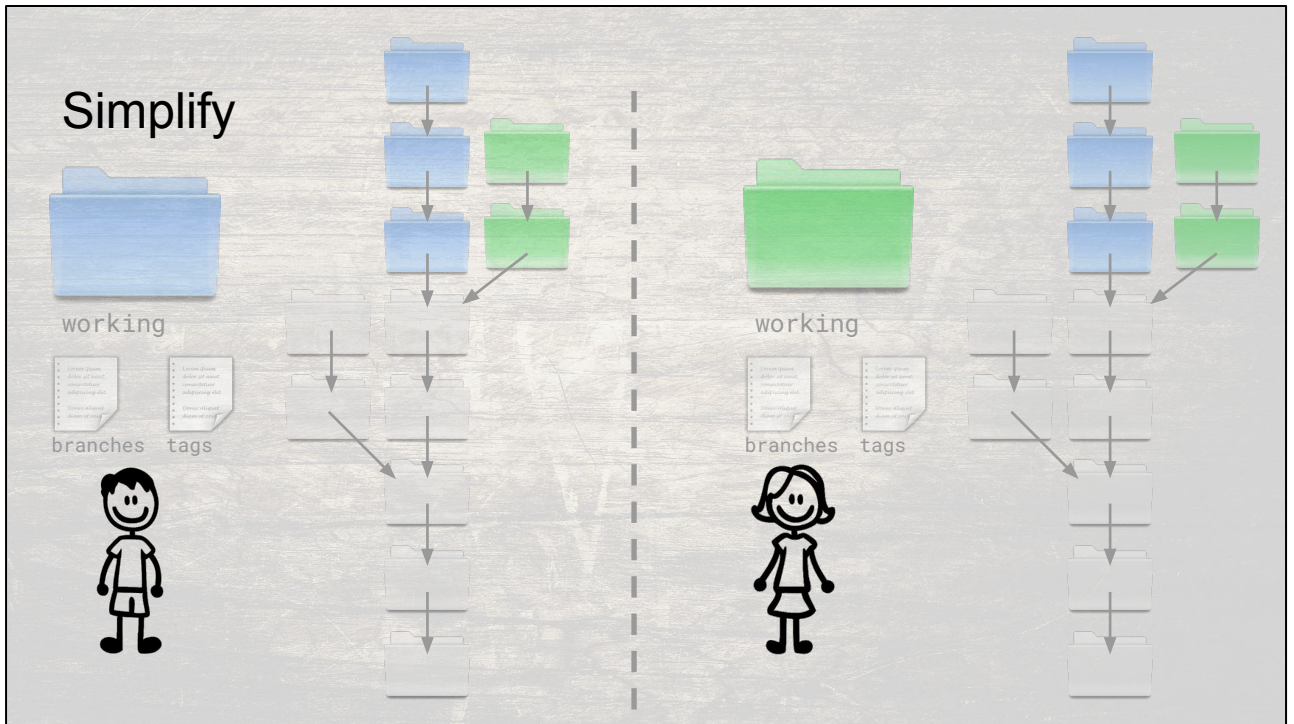
At the end of the day, she spends more time offline than online.

It's no surprise, then, that Zoe raves about your VCS:

All of the day to day operations that she needs to do can be done locally.

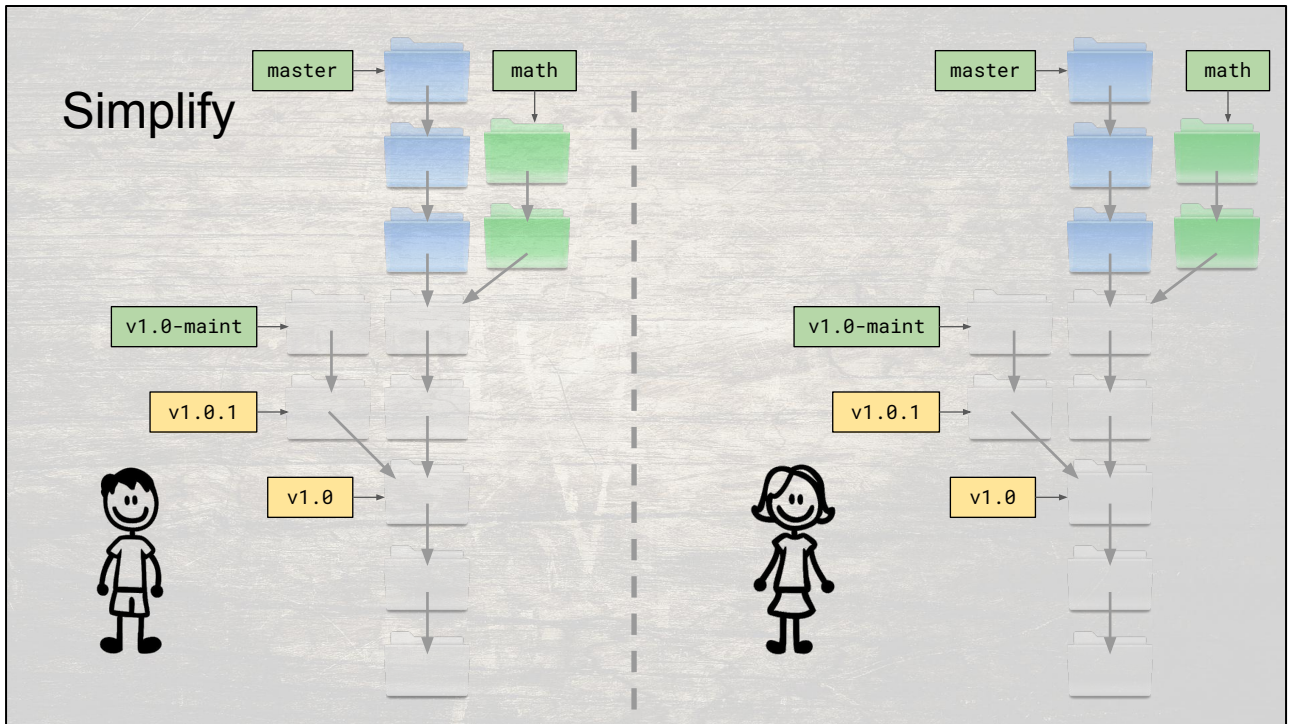


The only time she needs a network connection is when she's ready to share her snapshots with you.



Now, let's simplify our drawings a little, so that we can focus on the important issues:

We don't need to draw the "working" directory, and the "branches" and "tags" files. We know they're always there, anyway.



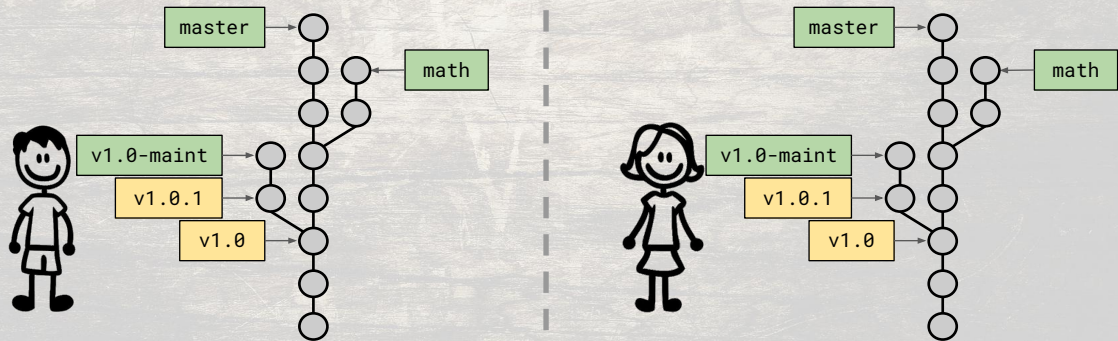
Instead we will add labels that point to the branches and tags that we are currently interested in.

Branches are green, and tags are yellow.

There's a “master” branch, a “v1.0-maint” branch, a “math” branch that we'll use shortly, and finally, two tags - “v1.0” and “v1.0.1”.



# Simplify

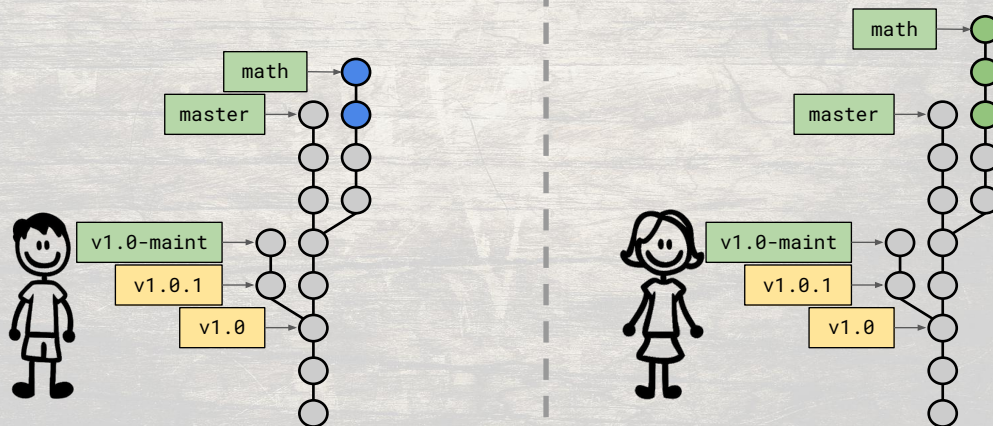


Finally, let's draw the snapshots as simple dots.

But remember, that they are still the same snapshot directories that we created from the start.

And since the pointers between snapshots always point downwards in this graph, we can use simple lines instead.

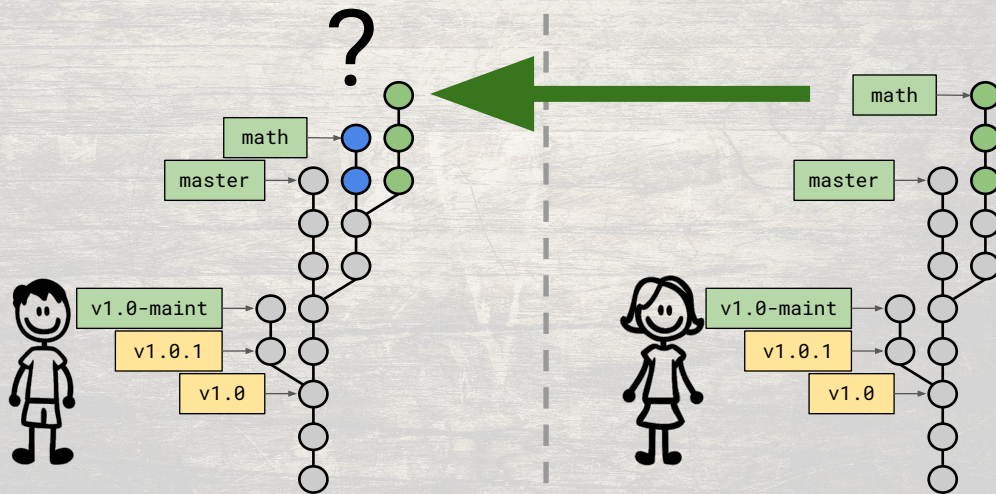
# Merges



Before Zoe left on her trip, you had asked her to start working off of the branch named 'math' and to implement a function that generated prime numbers (the 3 green snapshots).

Meanwhile, you were also developing off of the 'math' branch, only you were writing a function to generate, say, "magic numbers" (the 2 blue snapshots).

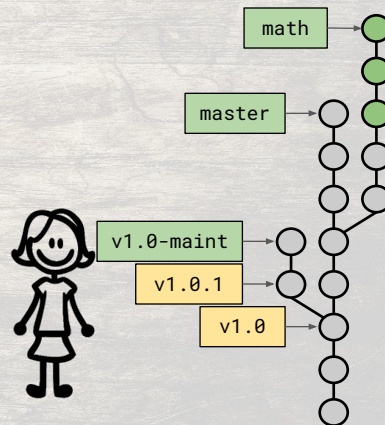
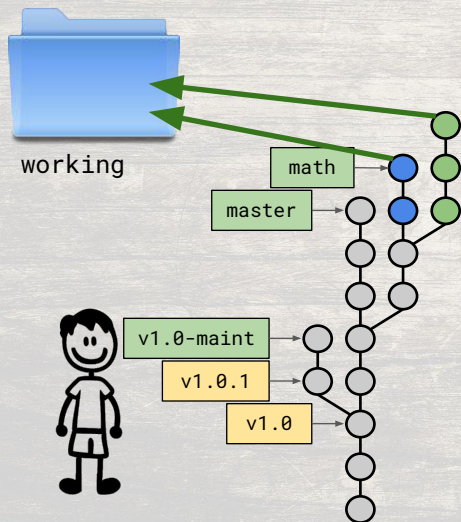
# Merges



Now that Zoe has returned, you are faced with the task of merging these two separate branches of development into a single snapshot.

But how?

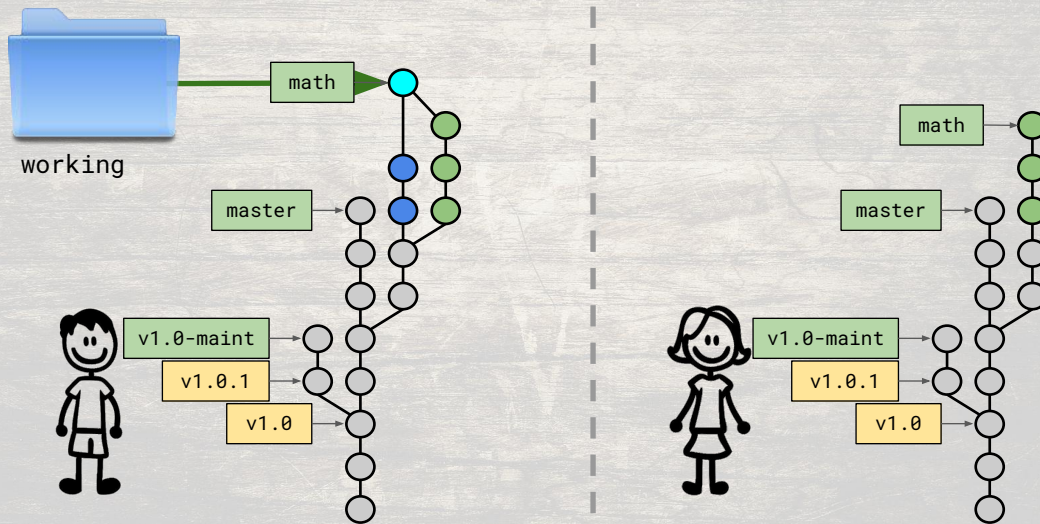
# Merges



In the working directory, you copy in the changes done on *both* branches, and reconcile their differences.



# Merges



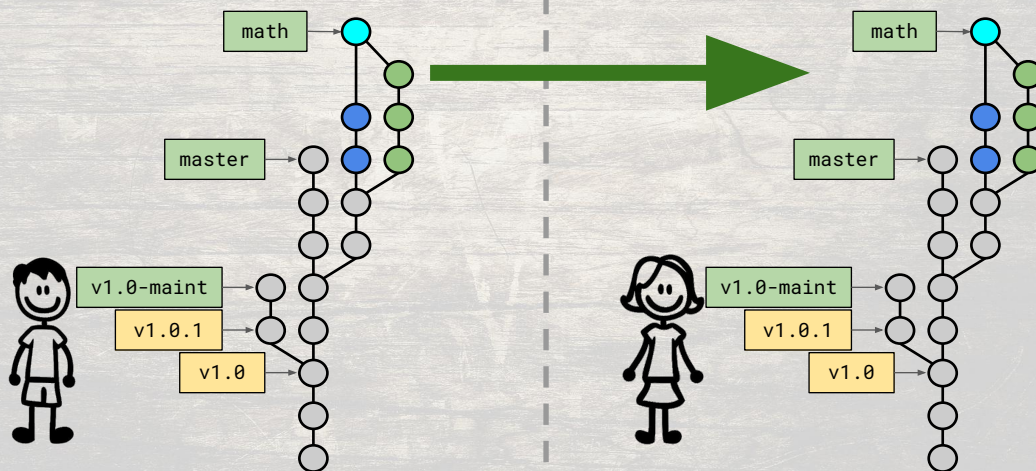
You create a new snapshot, a “merge” snapshot, from the working directory.

But there is something special about this snapshot:

Instead of just a single parent, this merge snapshot has *\*two\** parents!

The first parent is the latest on *\*your\** ‘math’ branch and the second parent is Zoe’s latest on *\*her\** ‘math’ branch.

# Merges

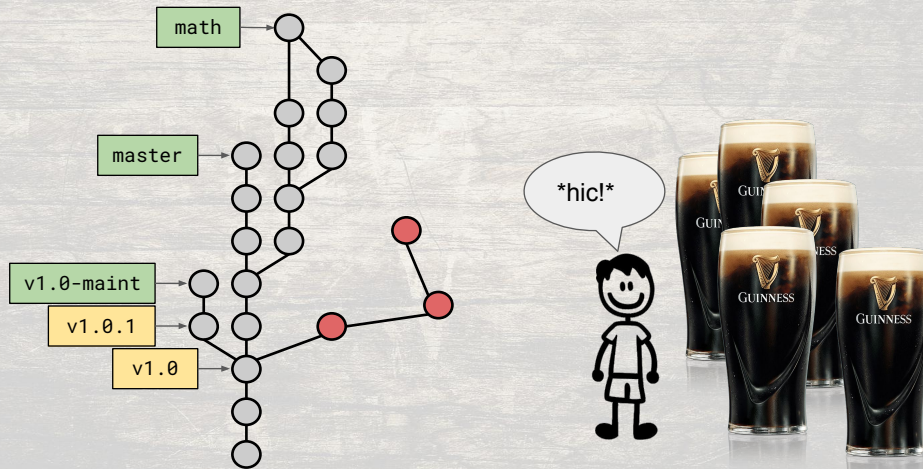


Once you complete the merge, Zoe fetches all the snapshots that you have that she does not:

- Your development on the 'math' branch (blue)
- The merge snapshot.

Once she does this, both of your histories match exactly!

# Rewriting History



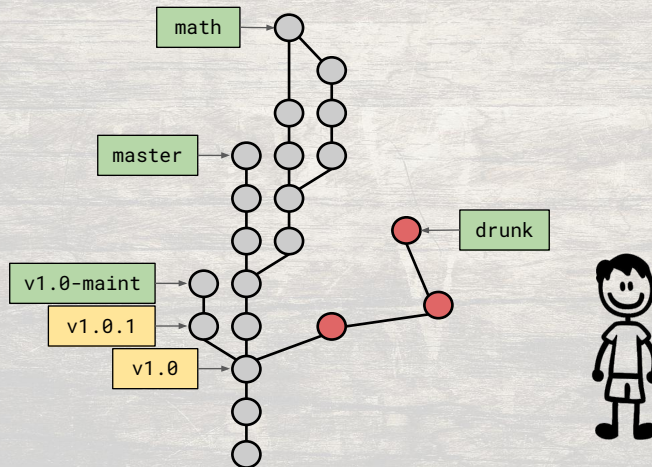
Like many software developers you have a compulsion to keep your code clean and very well organized. This carries over into a desire to keep your code history well groomed.

However, last night you came home after having a few too many pints of Guinness at the local pub and started coding, producing a handful of snapshots along the way.

This morning, a review of the code you wrote last night makes you cringe a little bit.

The code is good overall, but you made a lot of mistakes early on that you corrected in later snapshots.

# Rewriting History

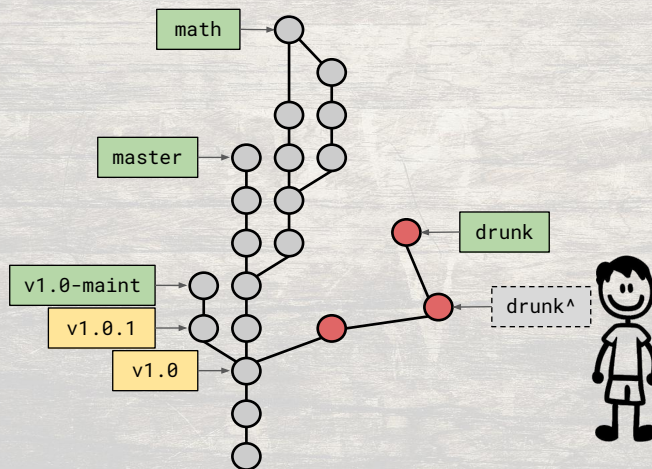


Let's say the branch on which you did your drunken development is called 'drunk' and you made three snapshots after you got home from the bar.

If the name 'drunk' points at the latest snapshot on that branch, then we can \*invent\* a useful notation to refer to the parent of that snapshot.

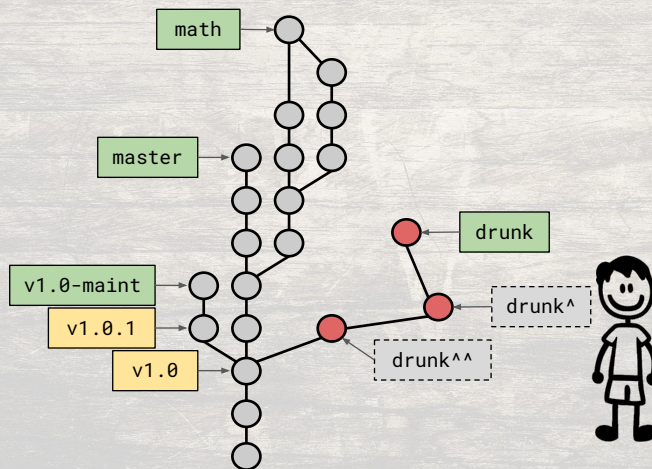


# Rewriting History



The notation 'drunk^' (caret/hat) means the parent of the snapshot pointed to by the branch name 'drunk'.

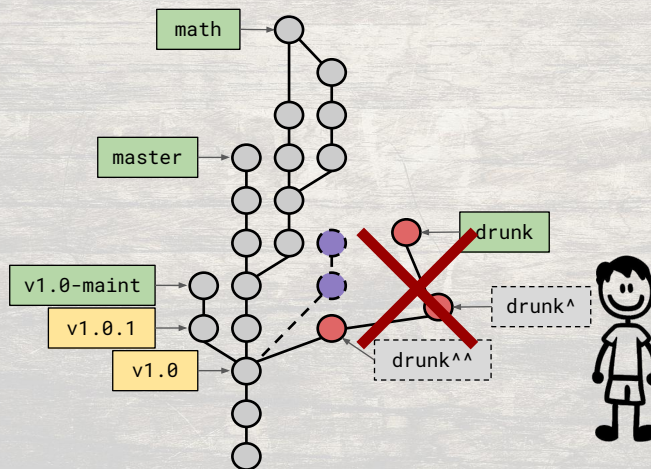
# Rewriting History



Similarly 'drunk^^' means the grandparent of the 'drunk' snapshot.

So the three snapshots in chronological order are 'drunk^^', 'drunk^', and 'drunk'.

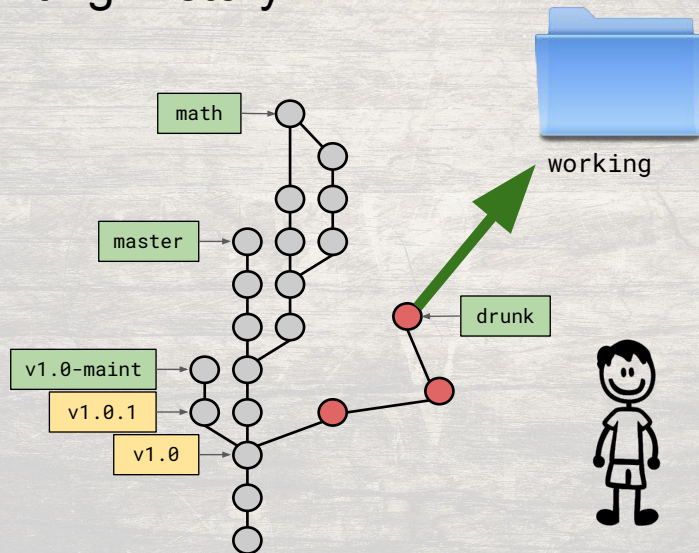
# Rewriting History



Say that the end result from the three messy snapshots on the “drunk” branch, was changing an existing function, “foo”, and adding a new file, “bar.cpp”.

You'd really like those three lousy snapshots to be two clean snapshots. One that changes the existing function, "foo", and one that adds the new file, "bar.cpp".

# Rewriting History

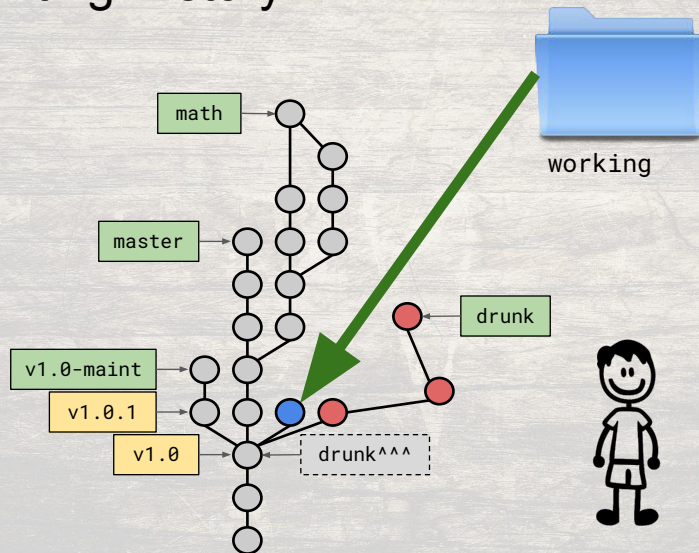


To accomplish this revision of history you copy 'drunk' to 'working' and delete the new file "bar.cpp".

Now 'working' contains the first change only: the correct modifications to the existing function "foo".



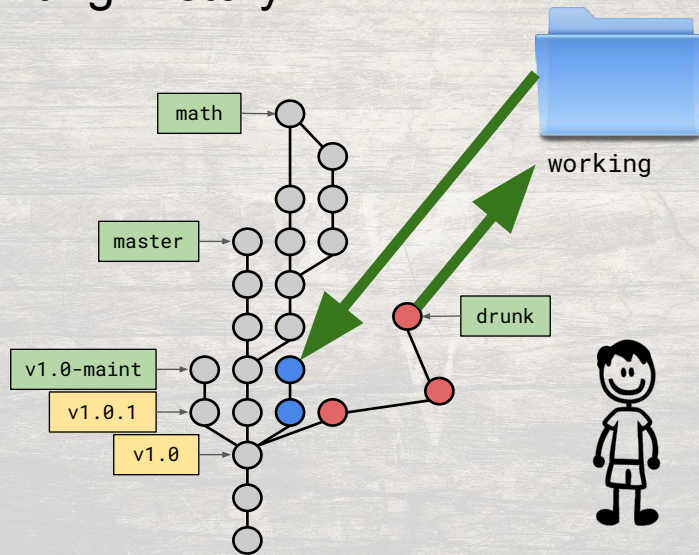
# Rewriting History



You create a new snapshot from 'working' and write the message to be appropriate to the changes.

For the parent you specify the SHA1 of the 'drunk^^^' snapshot, essentially creating a new branch off of the same snapshot as last night.

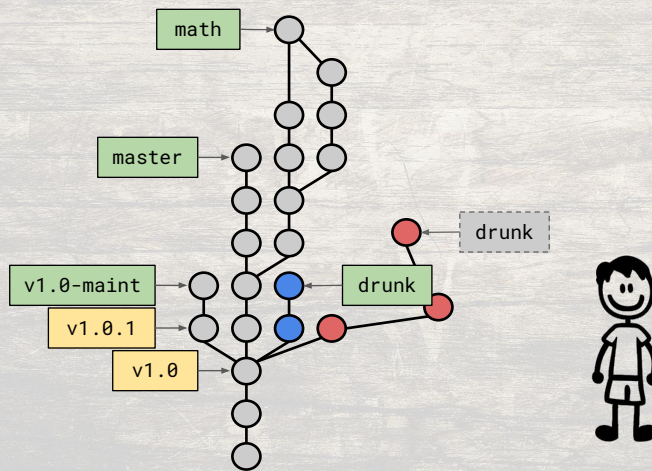
# Rewriting History



Now you can copy 'drunk' to 'working' and roll a snapshot that adds the new file "bar.cpp".

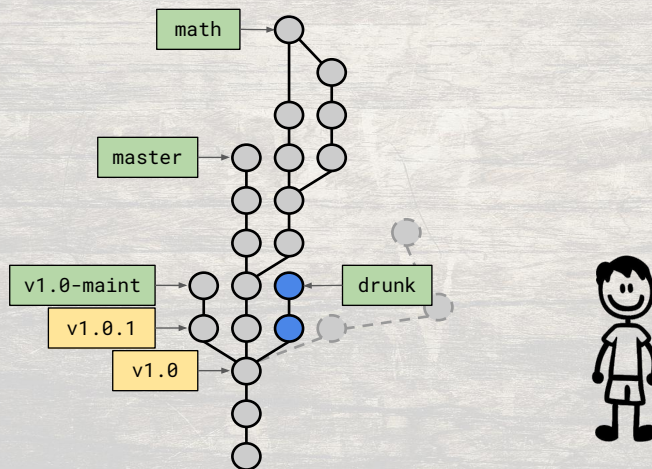
As the parent you specify the snapshot you created just before this one.

# Rewriting History



As the last step, you change the branch name 'drunk' to point to the last snapshot you just made.

# Rewriting History



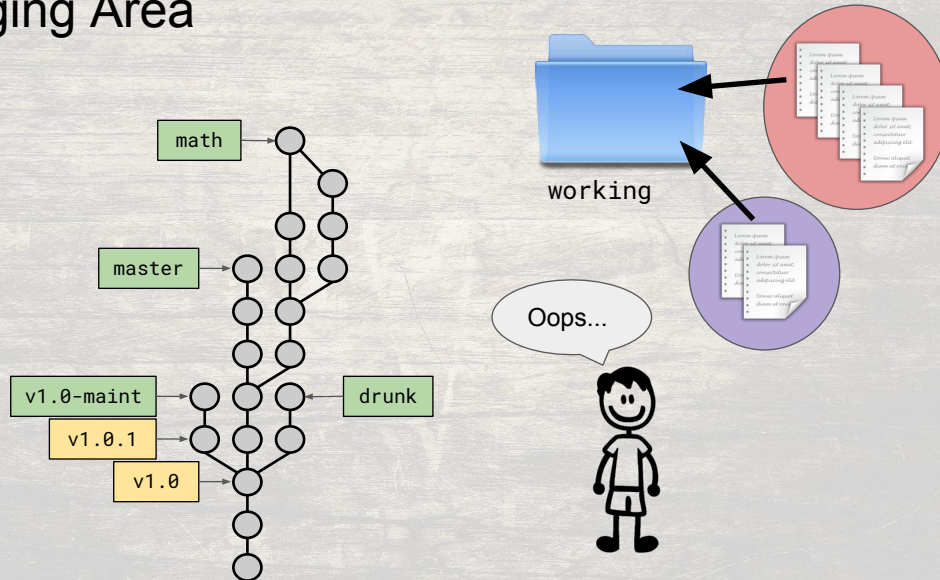
The history of the 'drunk' branch now represents a nicer version of what you did last night.

The other snapshots that you've replaced are no longer needed so you can delete them or just leave them around for posterity.

No branch names are currently pointing at them so it will be hard to find them later on, but if you don't delete them, they'll stick around.



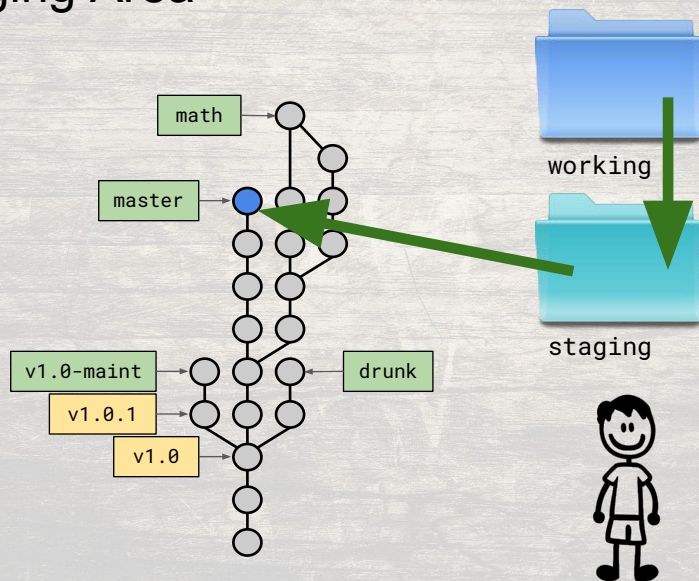
## Staging Area



As much as you try to keep your new modifications related to a single feature or logical chunk, you sometimes get sidetracked and start hacking on something totally unrelated.

Only half-way into this do you realize that your working directory now contains what should really be separated as two discrete snapshots.

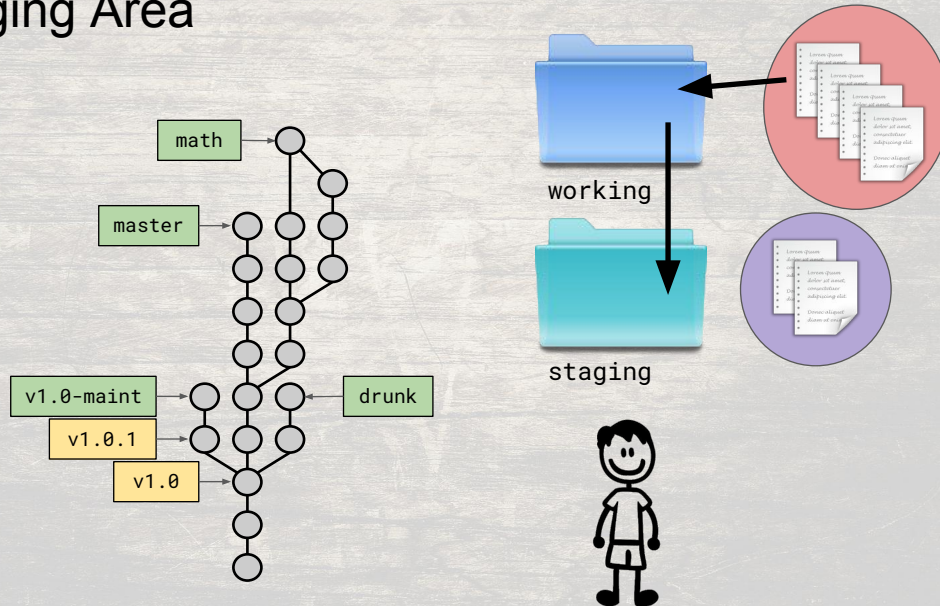
# Staging Area



To help you with this annoying situation, the concept of a “staging” directory is useful. This area acts as an intermediate step between your working directory and a final snapshot.

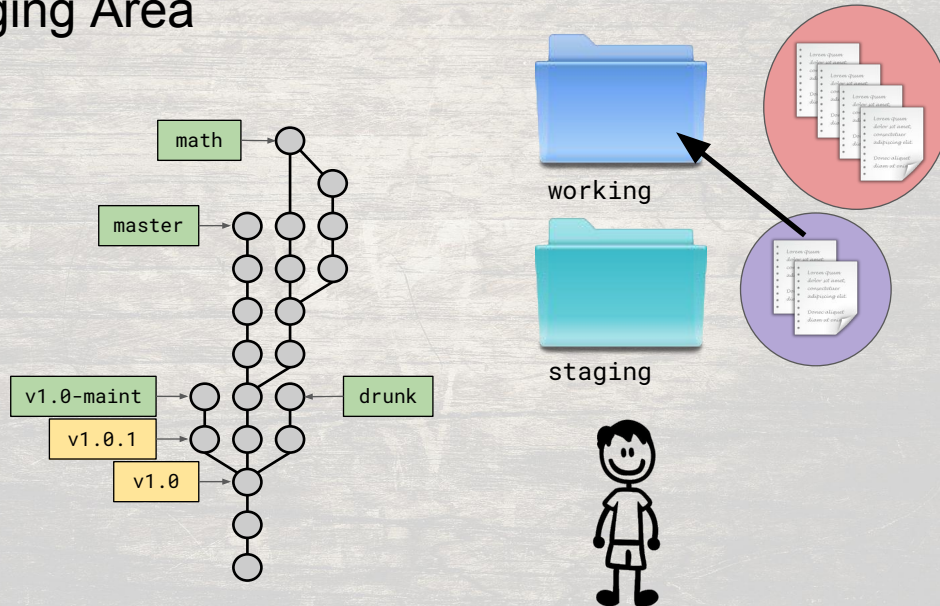
Each time you finish a snapshot, you first copy it to the “staging” directory, and then create the snapshot from the “staging” directory.

# Staging Area



If it belongs, you mimic the change inside “staging”.

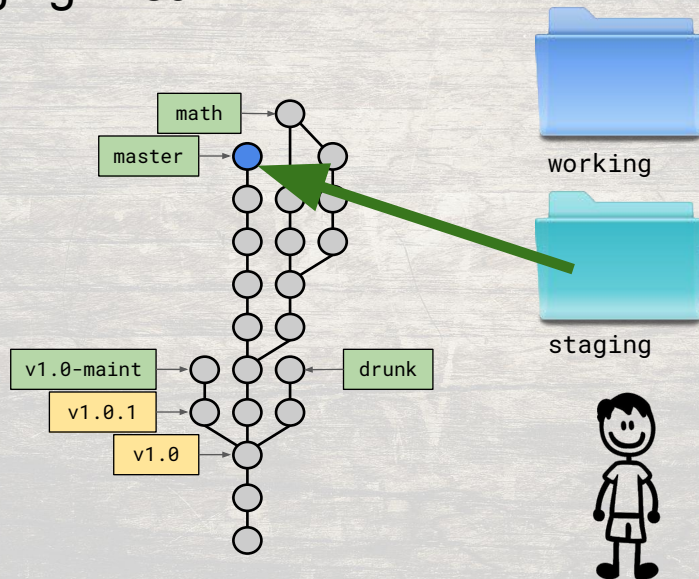
# Staging Area



If it doesn't, you can leave it in “working” and make it part of a later snapshot.



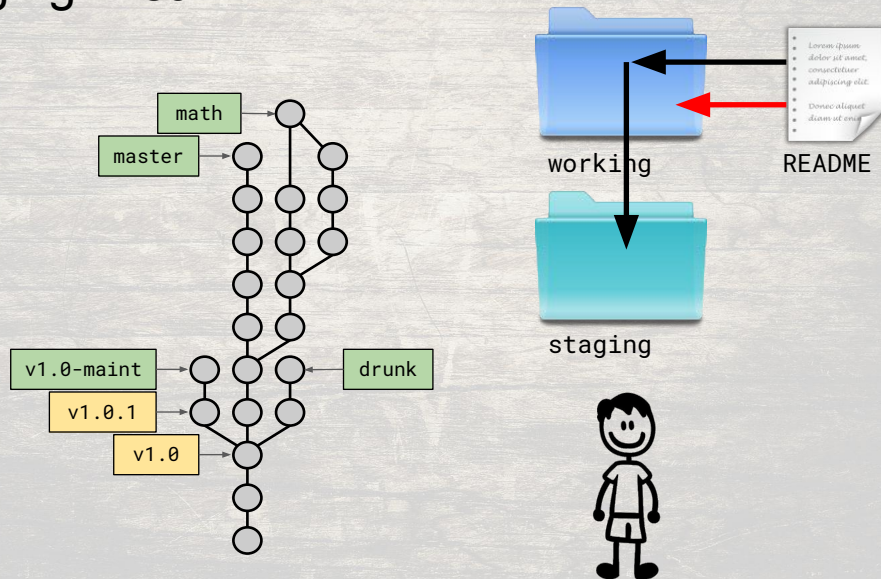
# Staging Area



When you're satisfied with the state of the “staging” directory, you create a new snapshot from it.

This separation of coding and preparing the stage makes it easy to specify what is and is not included in the next snapshot. You no longer have to worry too much about making an accidental, unrelated change in your working directory.

# Staging Area

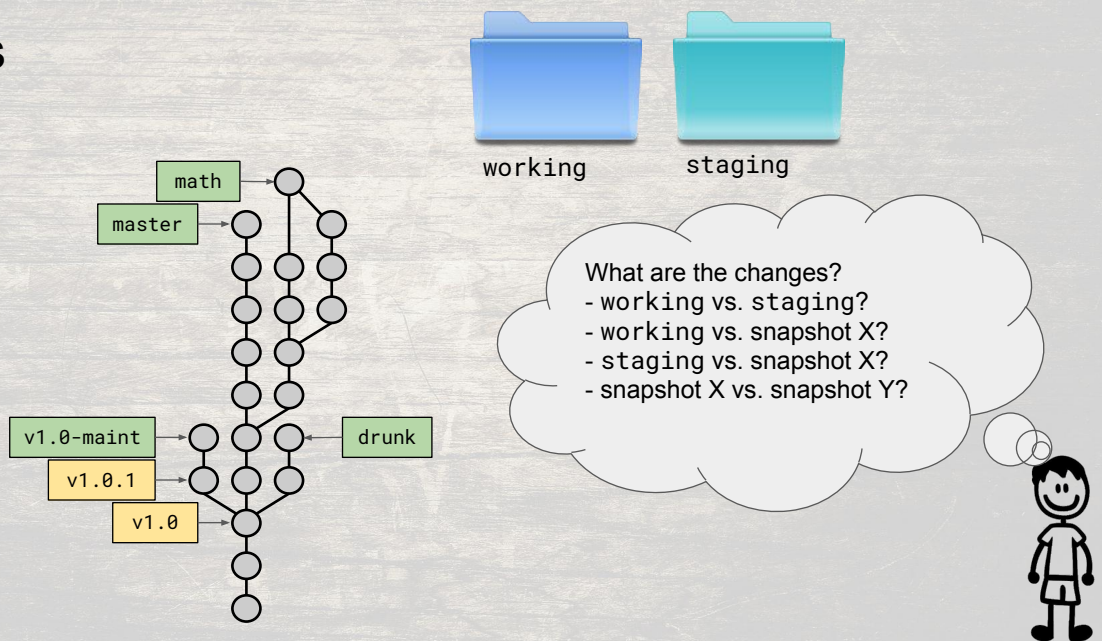


You have to be a bit careful, though. Consider a file named “README”. You make an edit to this file and then mimic that in “staging”. You go on about your business, editing other files. After a bit, you make another change to “README”.

Now you have made two changes to that file, but only *one* is in the staging area! Were you to create a snapshot now, your second change would be absent.

The lesson is this: *\*Every new edit must be added to the staging area if it is to be part of the next snapshot.\**

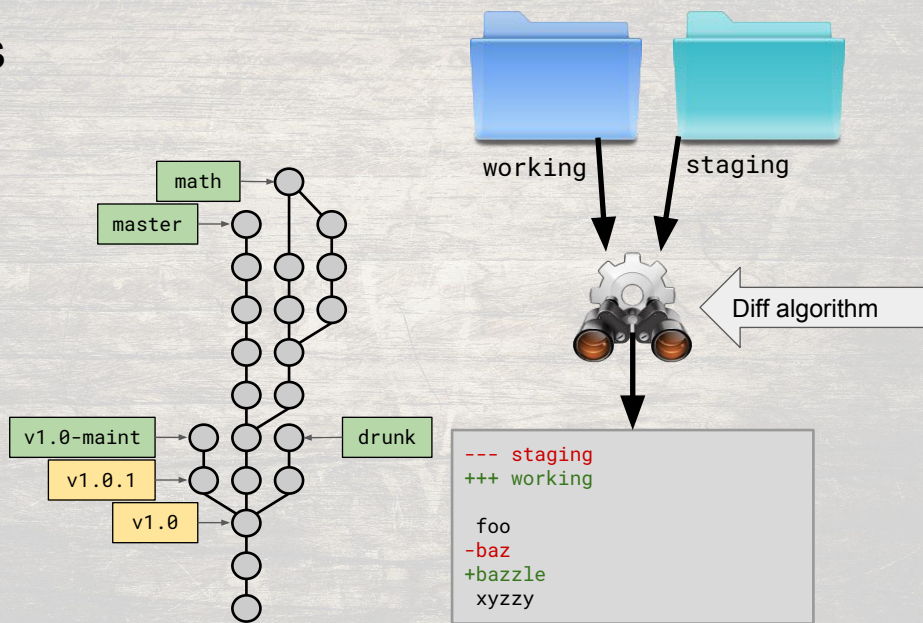
# Diffs



With a working directory, a staging area, and loads of snapshots laying around, it starts to get confusing as to what the specific code changes are between these directories.

A snapshot message only gives you a summary of what changed, not exactly what lines were changed between two files.

# Diffs

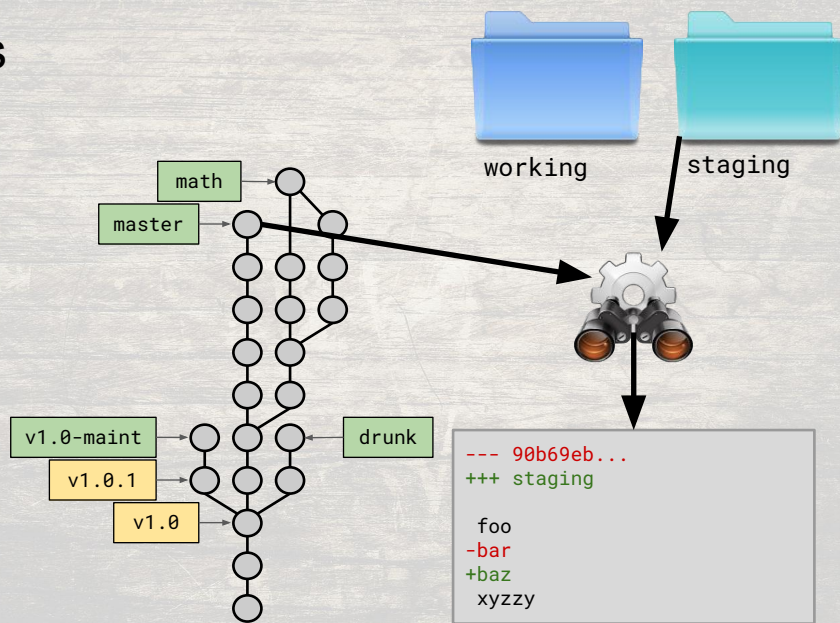


Using a diffing algorithm, you can implement a small program that shows you the differences in two codebases.

As you develop and copy things from your working directory to the staging area, you'll want to easily see what is different between the two, so that you can determine what else needs to be staged.

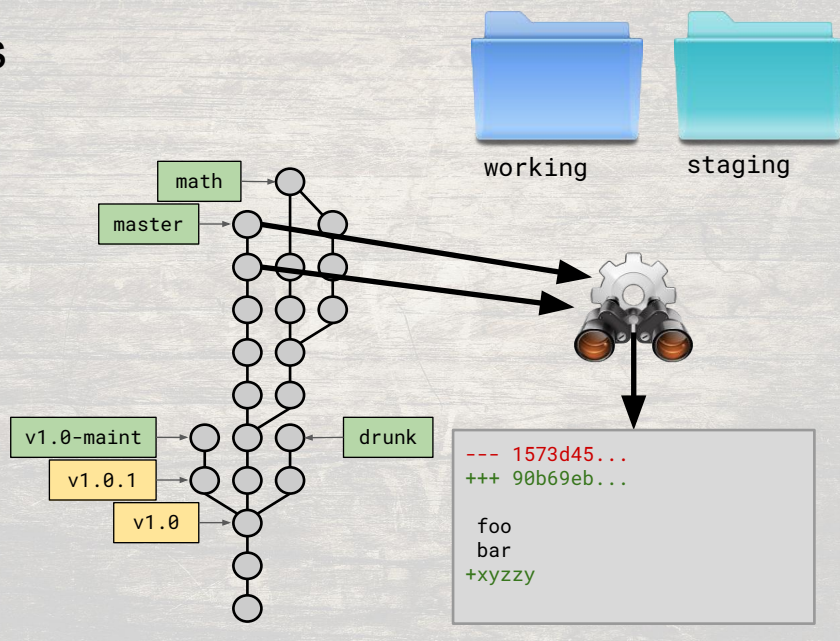


# Diffs



It's also important to see how the staging area is different from the last snapshot, since these changes are what will become part of the next snapshot you produce.

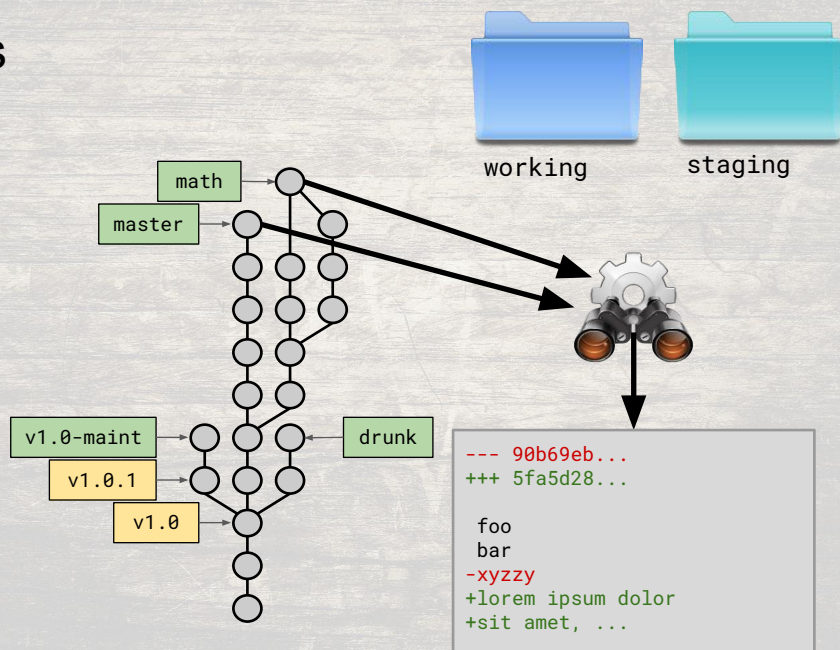
# Diffs



There are many other diffs you might want to see.

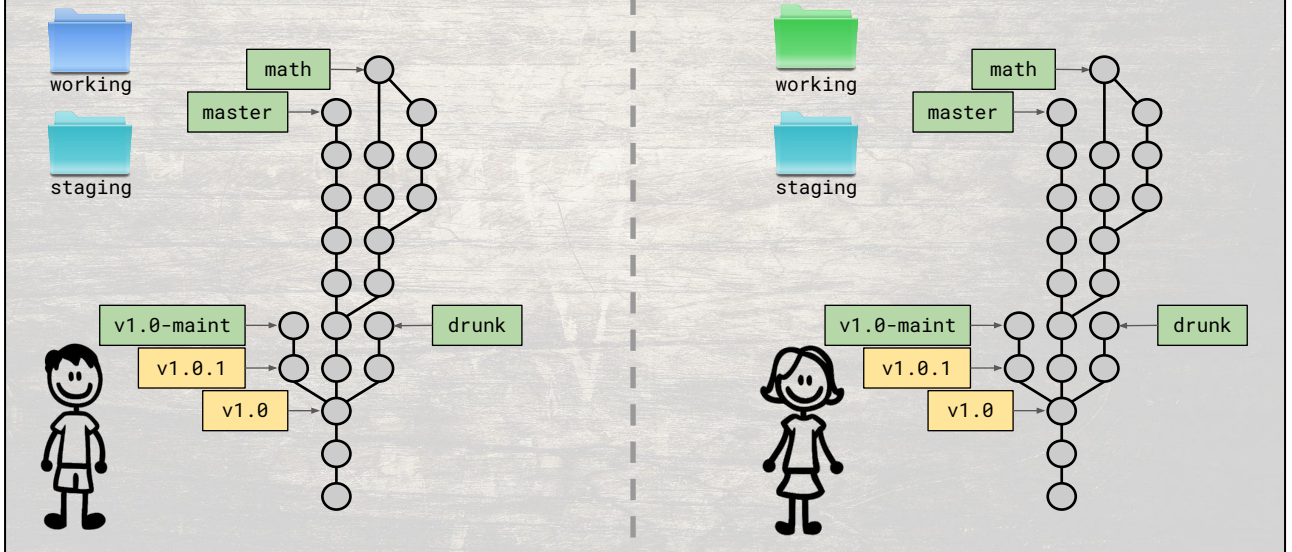
The differences between a specific snapshot and its parent would show you the specific changes (aka. “changeset”) that was introduced by that snapshot.

# Diffs



The diff between two branches would be helpful for making sure your development doesn't wander too far away from the mainline.

## Eliminating Duplication



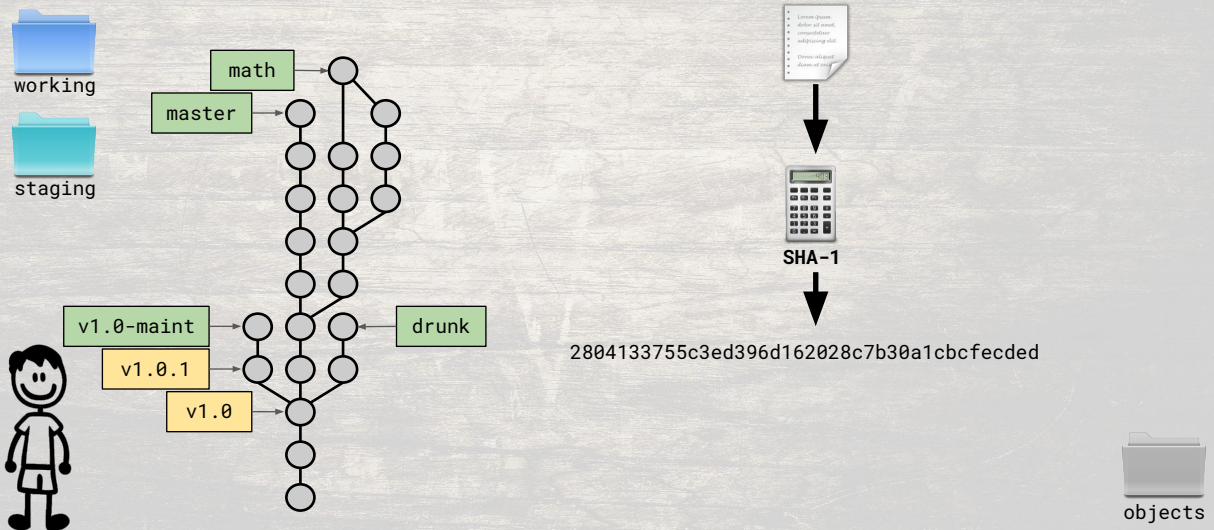
Remember Zoe?

After a few more trips around the world, Zoe starts to complain that her hard drive is filling up with hundreds of nearly identical copies of the software.

You too have been feeling like all the file duplication is wasteful. After a bit of thinking, you come up with something very clever.

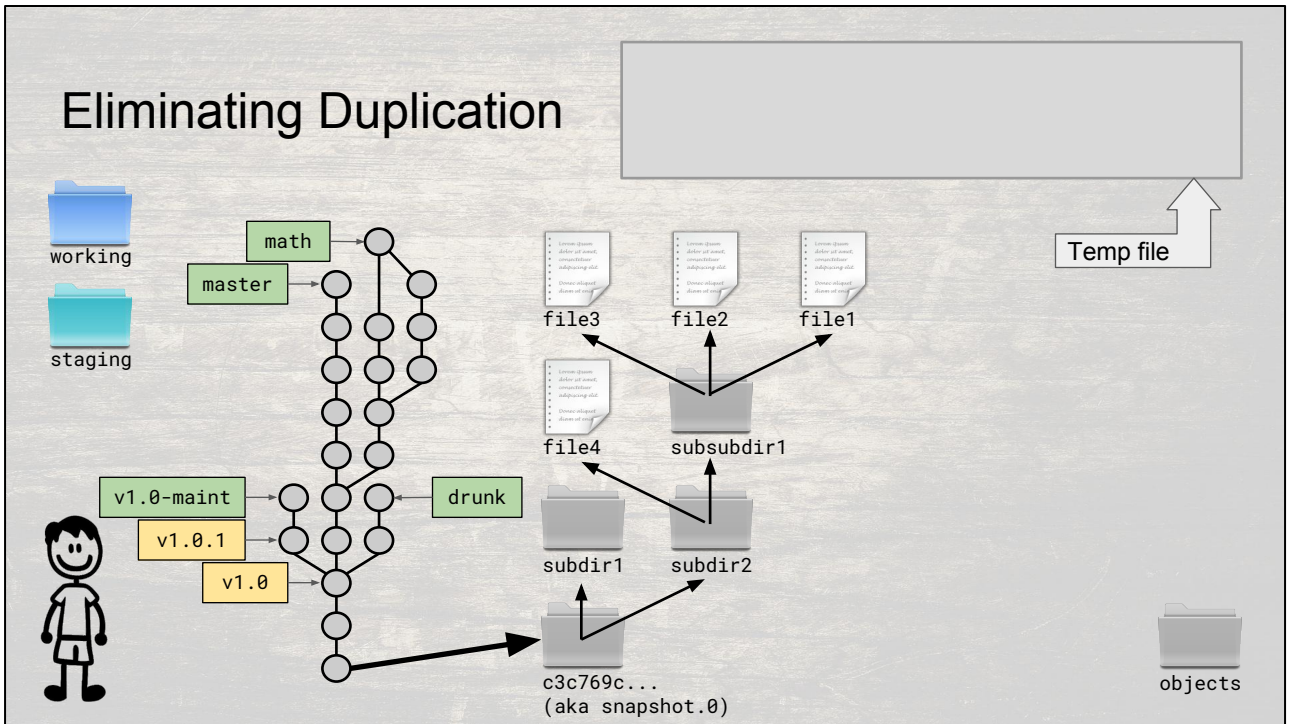


# Eliminating Duplication



You remember that the SHA1 hash produces a short string that is unique for a given file contents.

Starting with the very first snapshot in the project history, you start a big conversion process. First, you create a directory named “objects” outside of the code history.

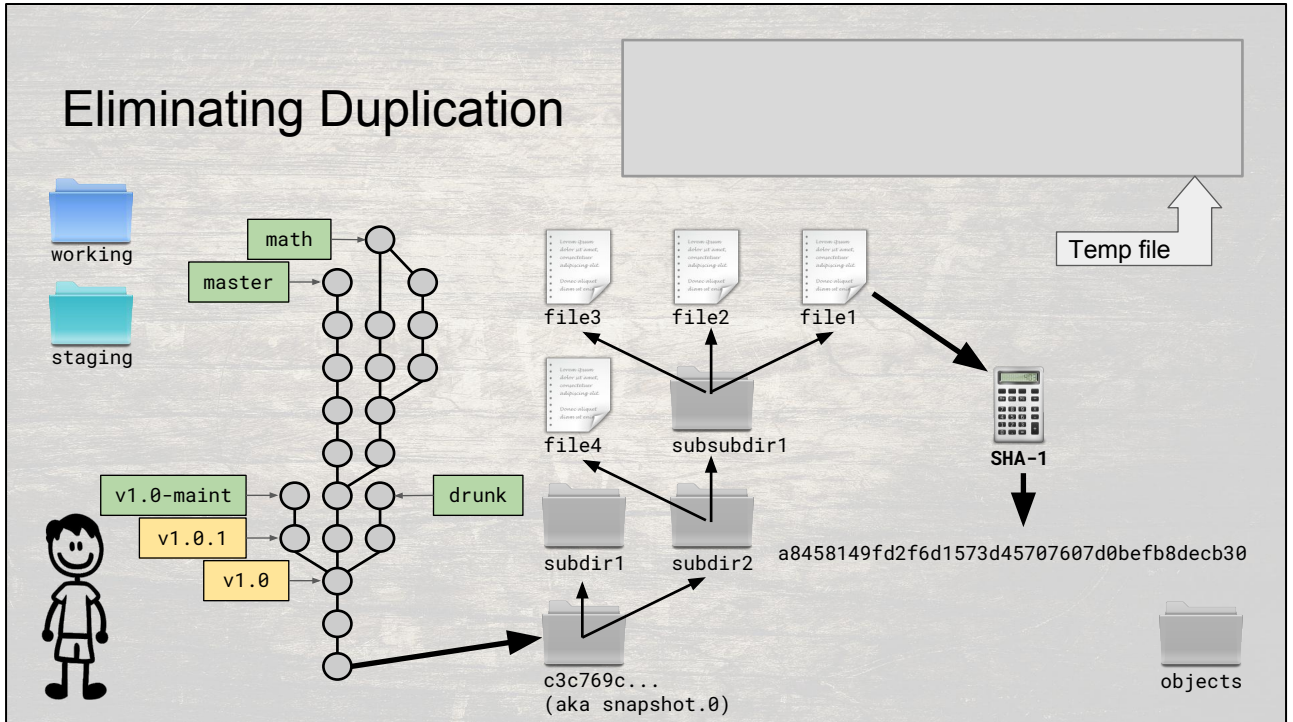


Next, you go into the first snapshot directory, and find the most deeply nested directory in the snapshot.

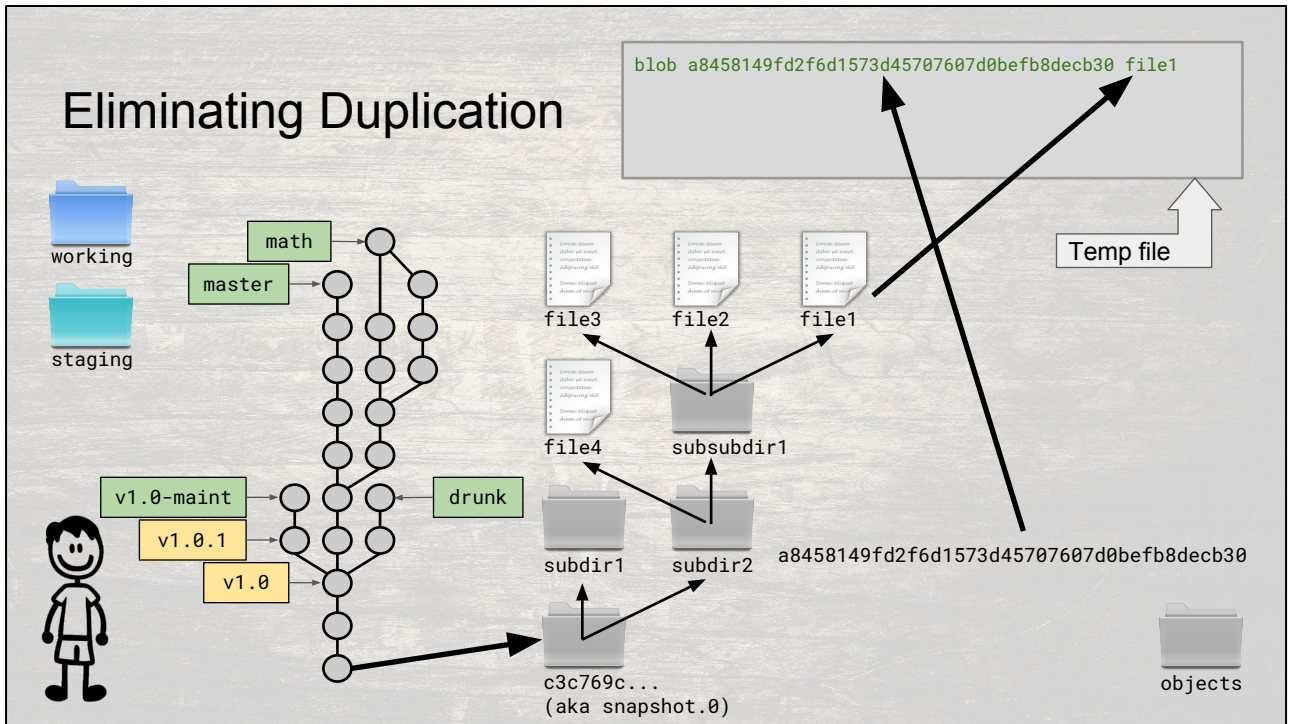
Additionally, you open up a temporary file for writing.

Now, for each file in the most deeply nested directory, you perform the following three steps:

# Eliminating Duplication

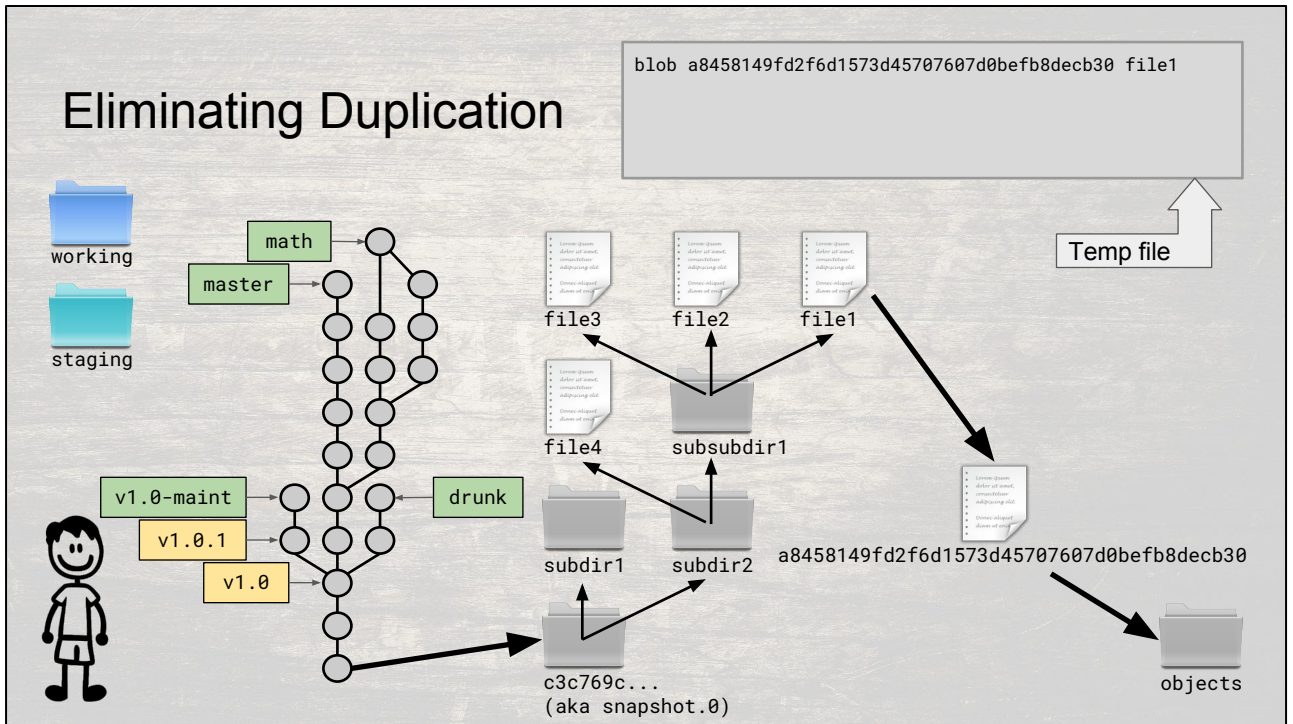


Step 1: Calculate the SHA1 of the file contents.

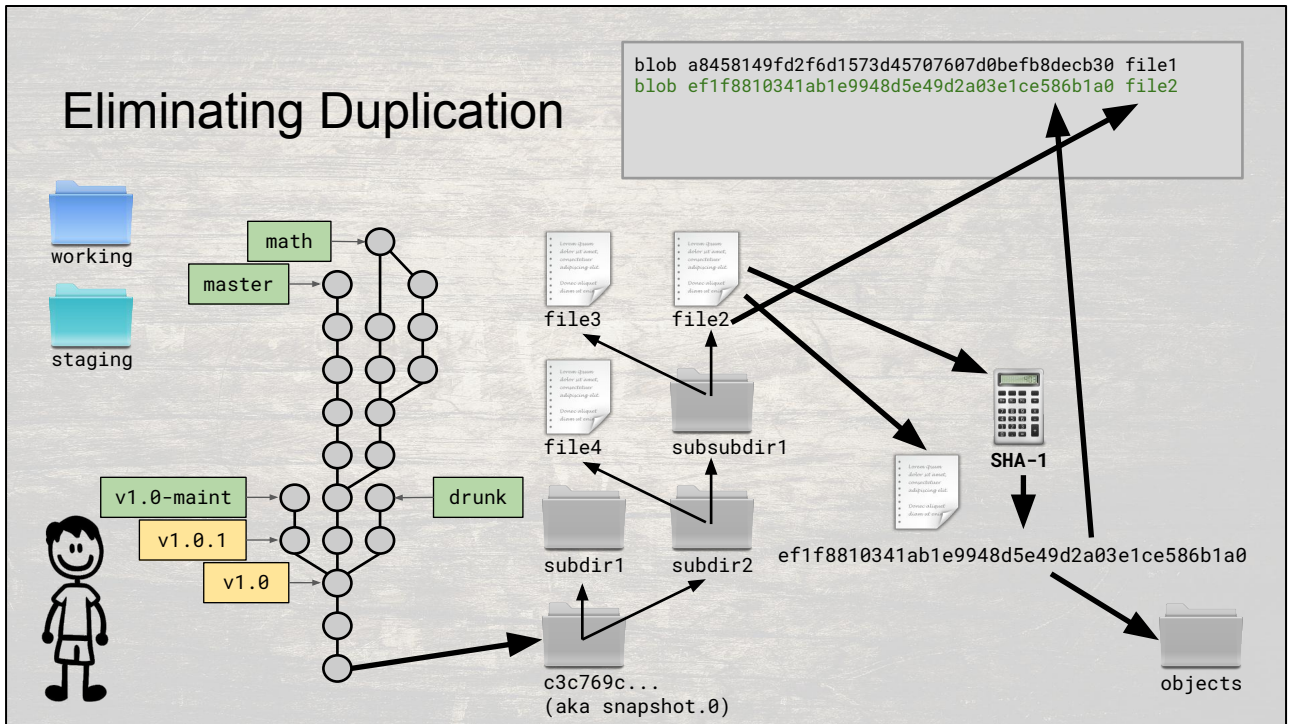


Step 2: Add an entry into the temp file that contains the word 'blob' (binary large object), the SHA1 from the first step, and the filename.





Step 3: Copy the file to the objects directory and rename it to the SHA1 from step 1.

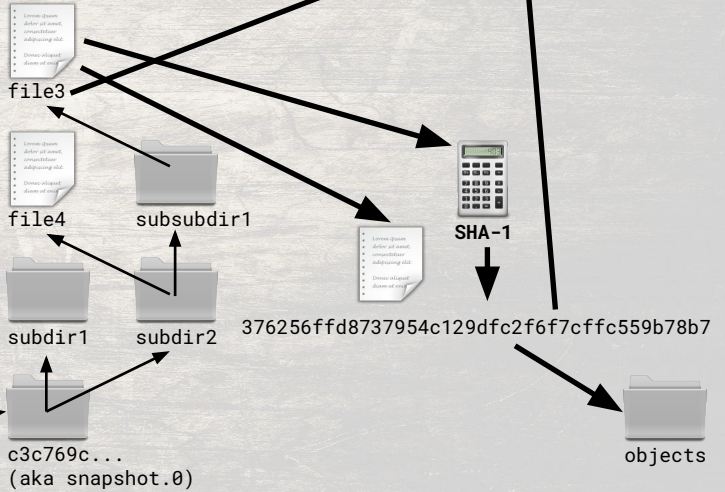
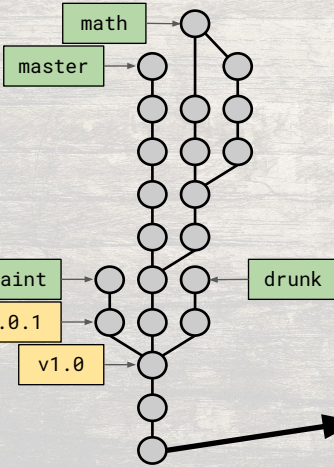


Repeat this process for the other two files in that directory.

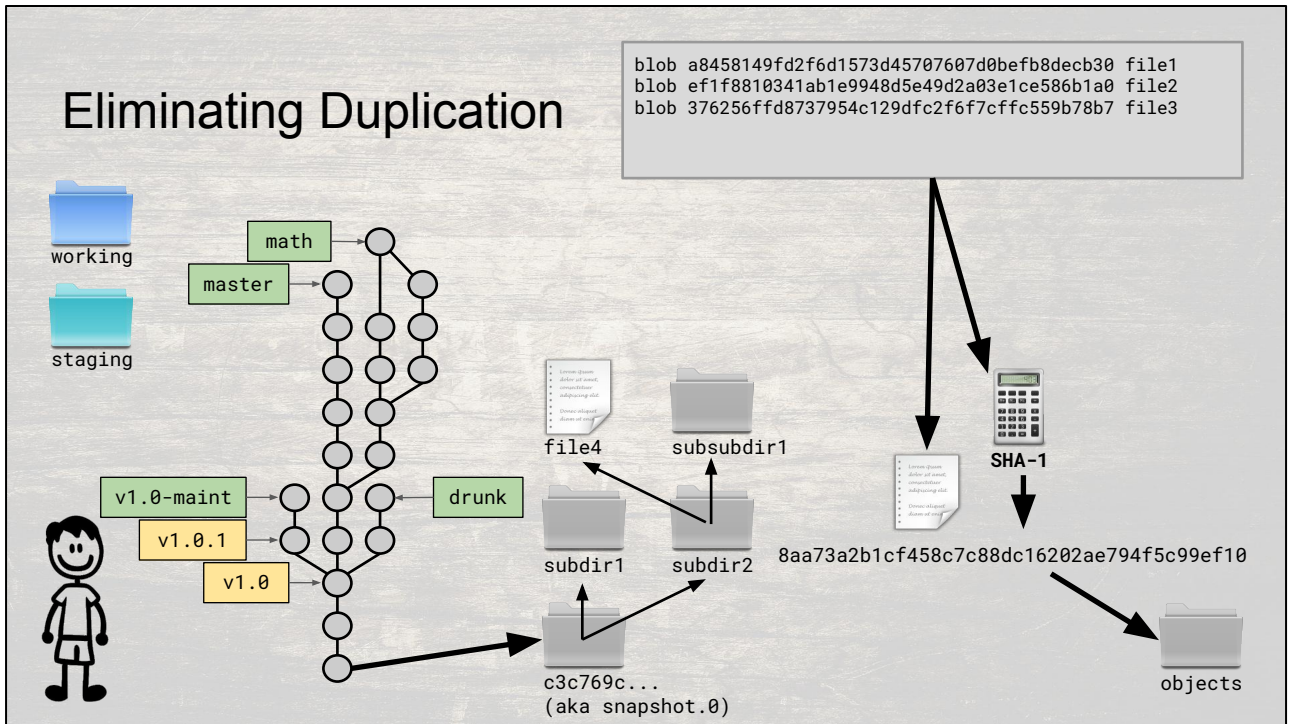
# Eliminating Duplication

```
blob a8458149fd2f6d1573d45707607d0befb8dec30 file1
blob ef1f8810341ab1e9948d5e49d2a03e1ce586b1a0 file2
blob 376256ffd8737954c129dfc2f6f7cffc559b78b7 file3
```

working  
staging



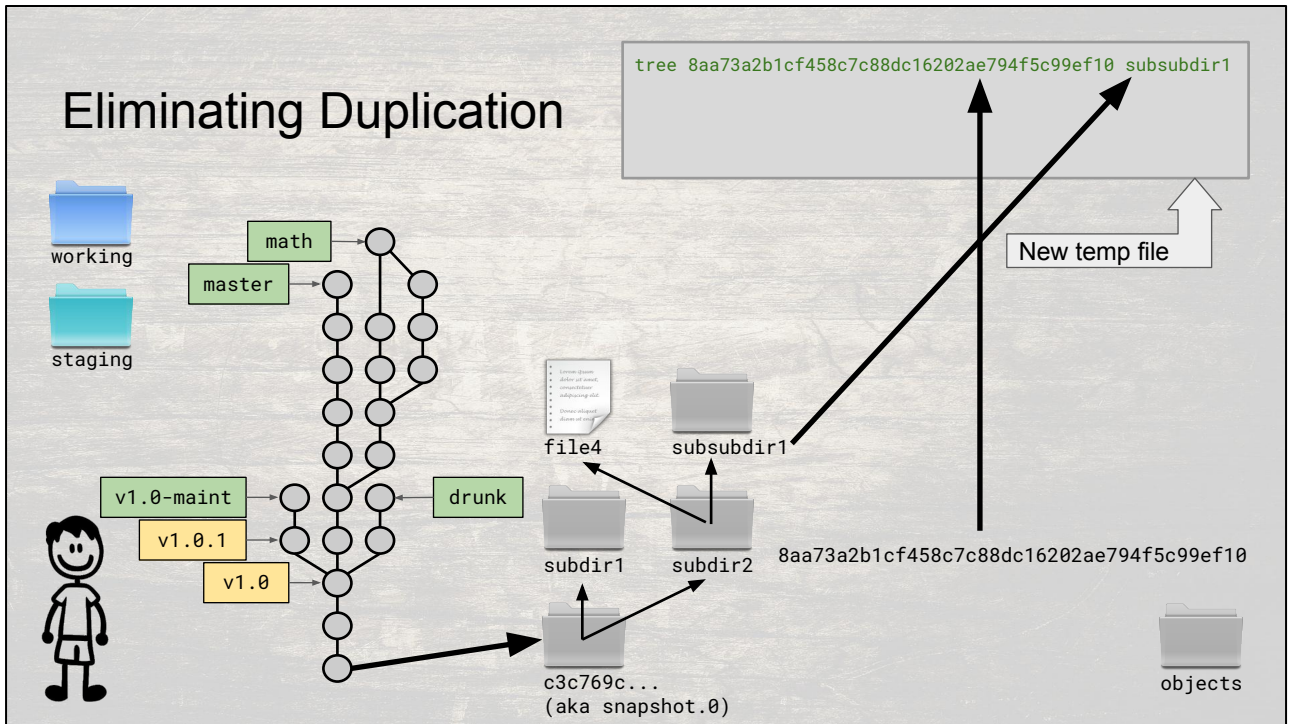
...



Once finished with all the files, find the SHA1 of the temp file contents and use that to name the temp file, also placing it in the objects directory.

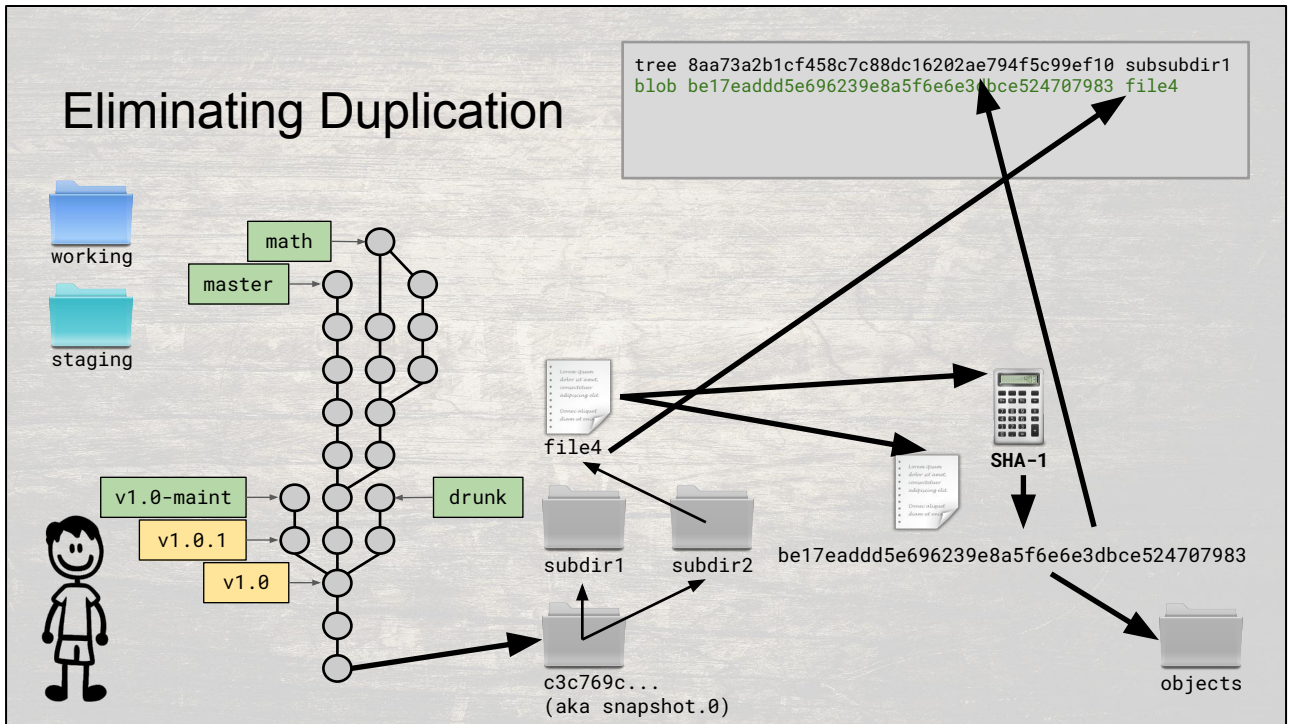
Note that the contents of the temp file exactly mirrors the contents of “subsubdir1”; it is kind of a directory listing. We use this fact in the next phase:





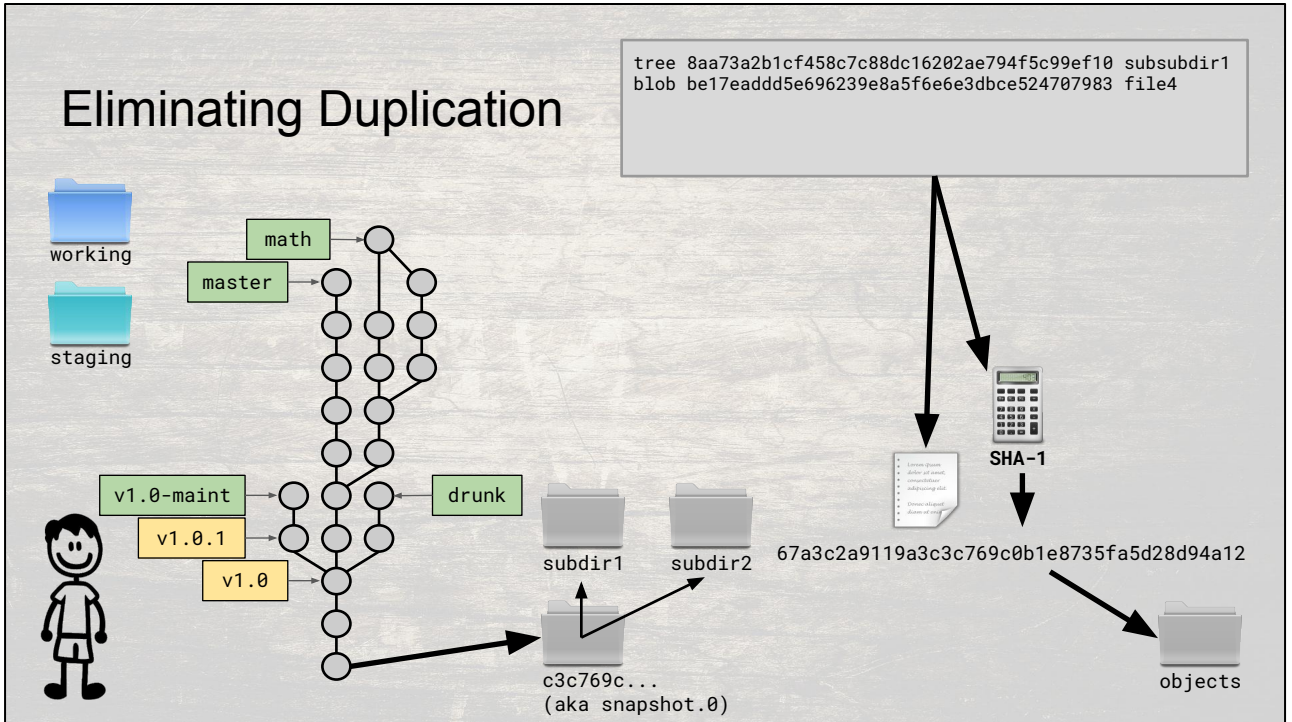
Now, move up one directory and start over. Only this time, when you get to the entry for the directory that you just processed, enter the word 'tree', the SHA1 of the temp file from last time, and the directory's name into the new temp file.

So, to identify the contents of the "subsubdir1" directory, we re-use the SHA1 of the "directory listing" from the last phase.

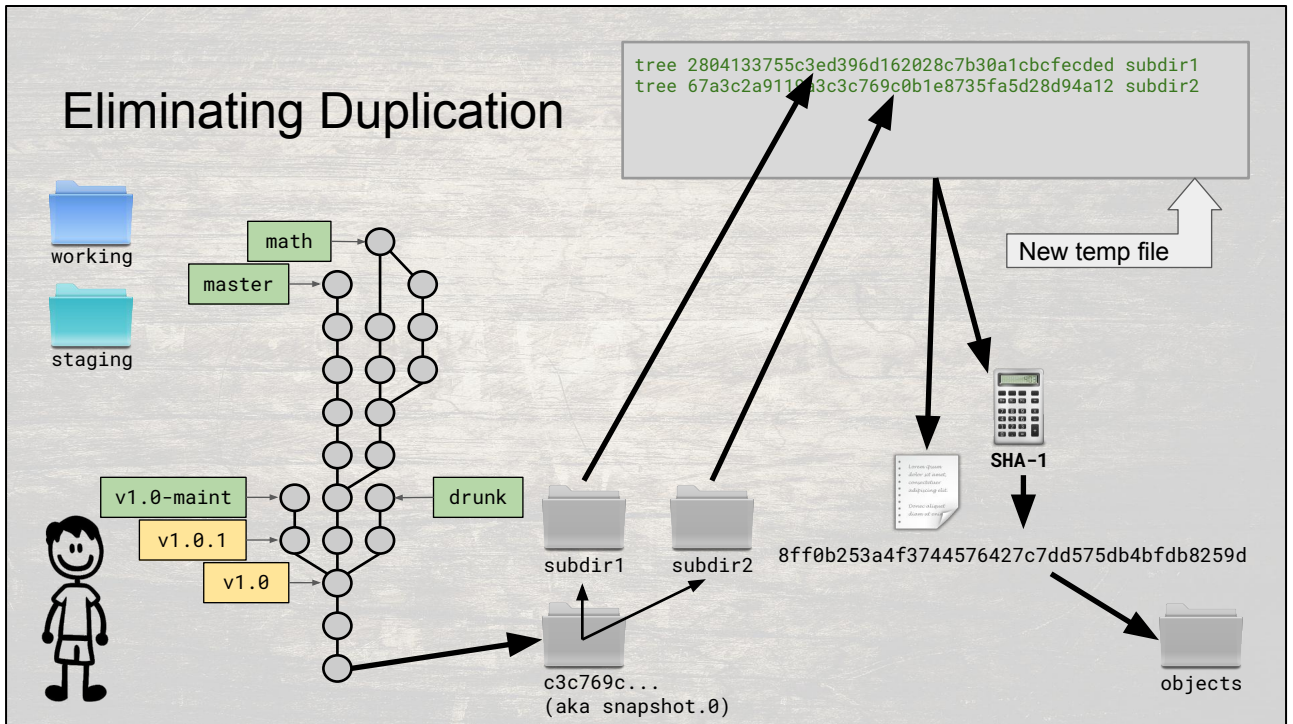


Now, at this level there's also a regular file, "file4". We handle this in the same way as the previous files in "subsubdir1".

# Eliminating Duplication



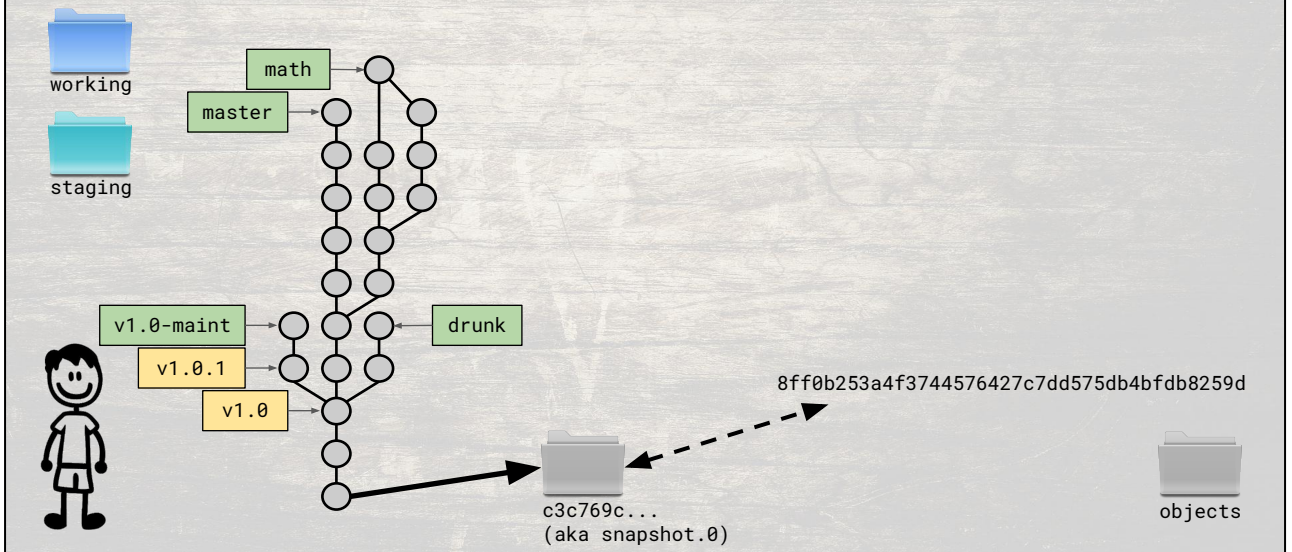
When we finish that directory level, we hash and store the directory listing, and keep going at the next level.



In this fashion you can build up a tree of directory object files that contain the SHA1s and names of the files and directory objects that they contain.



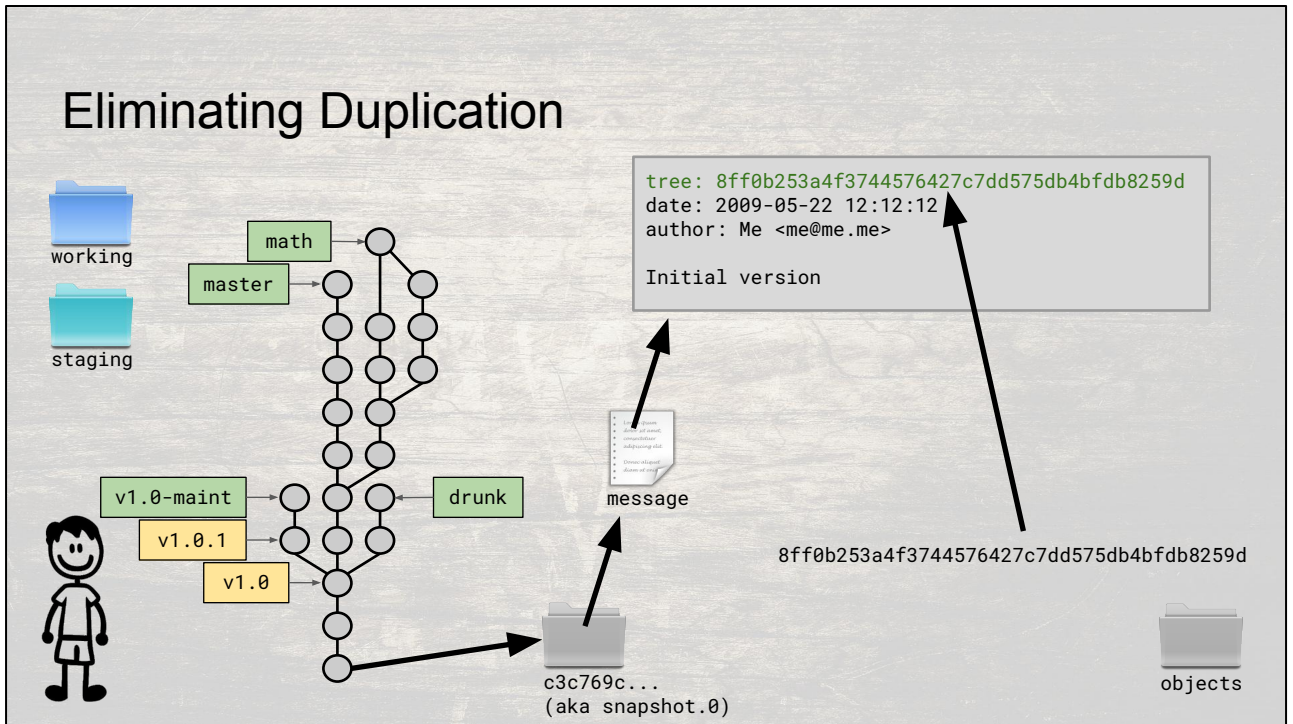
# Eliminating Duplication



Once this has been accomplished for every directory and file in the snapshot, you have the SHA1 of the directory listing for the root folder of your snapshot.

Now you need to record the root tree's SHA1 somewhere, but where?

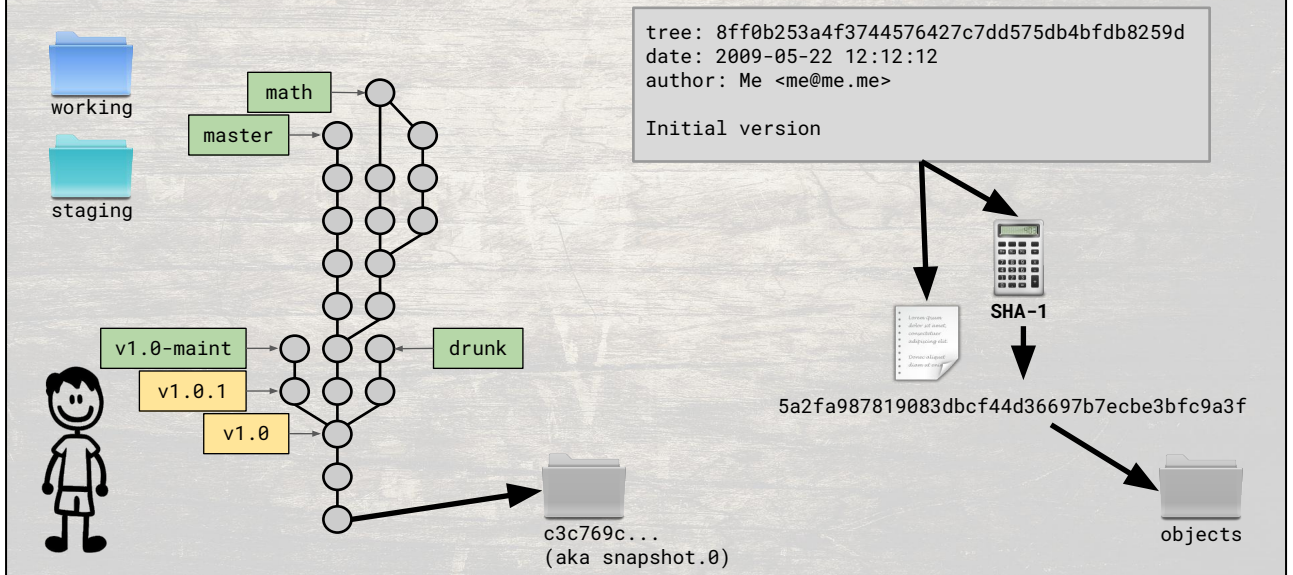
# Eliminating Duplication



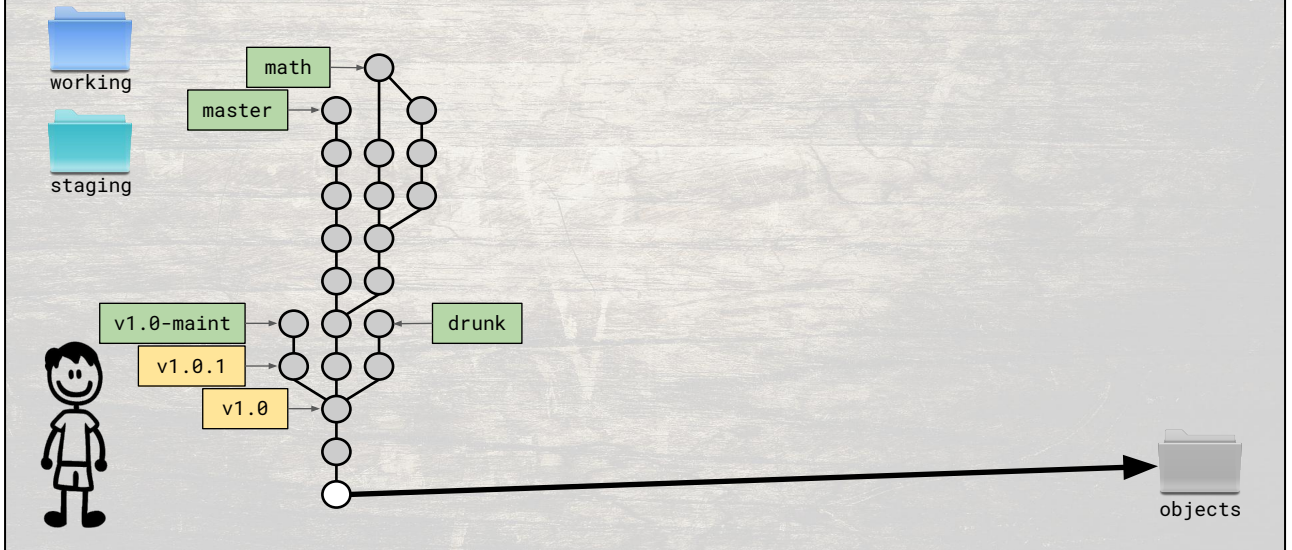
An ideal place to store it is in the snapshot message file.

This way, the uniqueness of the SHA1 of the message also depends on the entire contents of the snapshot, and you can guarantee that two identical snapshot message SHA1s must contain exactly the same files!

# Eliminating Duplication



# Eliminating Duplication



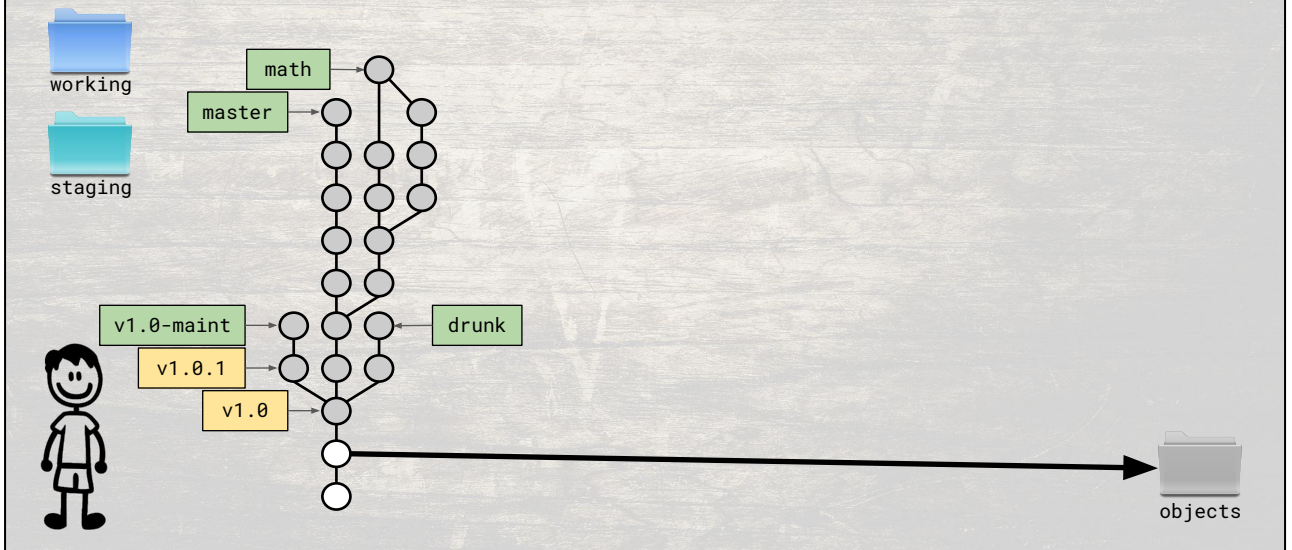
With all of this information stored in the objects directory, you can safely delete the snapshot directory that you used as the source of this operation.

If you want to reconstitute the snapshot at a later date it's simply a matter of following the SHA1 of the root tree stored in the message file and extracting each tree and blob into their corresponding directory and file.

For a single snapshot, this transformation process doesn't get you much. You've basically just converted one filesystem into another and created a lot of work in the process.



# Eliminating Duplication

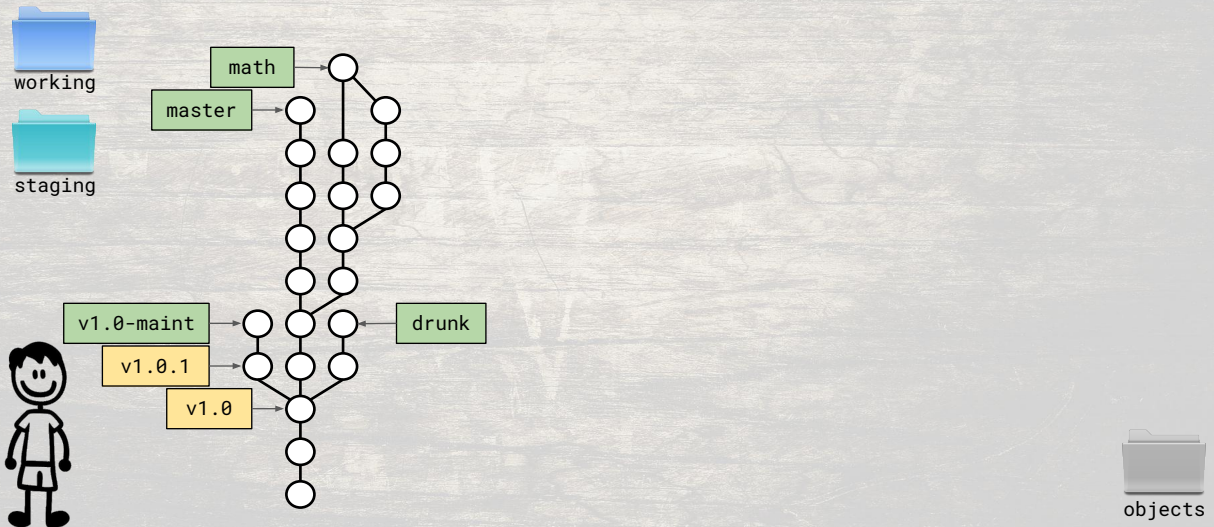


But once you start transforming the second snapshot you discover something:

Many of the files and directories are unchanged from the first snapshot, and they therefore end up with the same SHA1. But since they have the same SHA1, they \*already\* exist in the objects directory, so you don't have to copy them in there.

Imagine two sequential snapshots in which only a single file in the top directory has changed. If the snapshots both contain 10 directories and 100 files, the transformation process will create 10 trees and 100 blobs from the first snapshot but only one new blob and one new tree from the second snapshot!

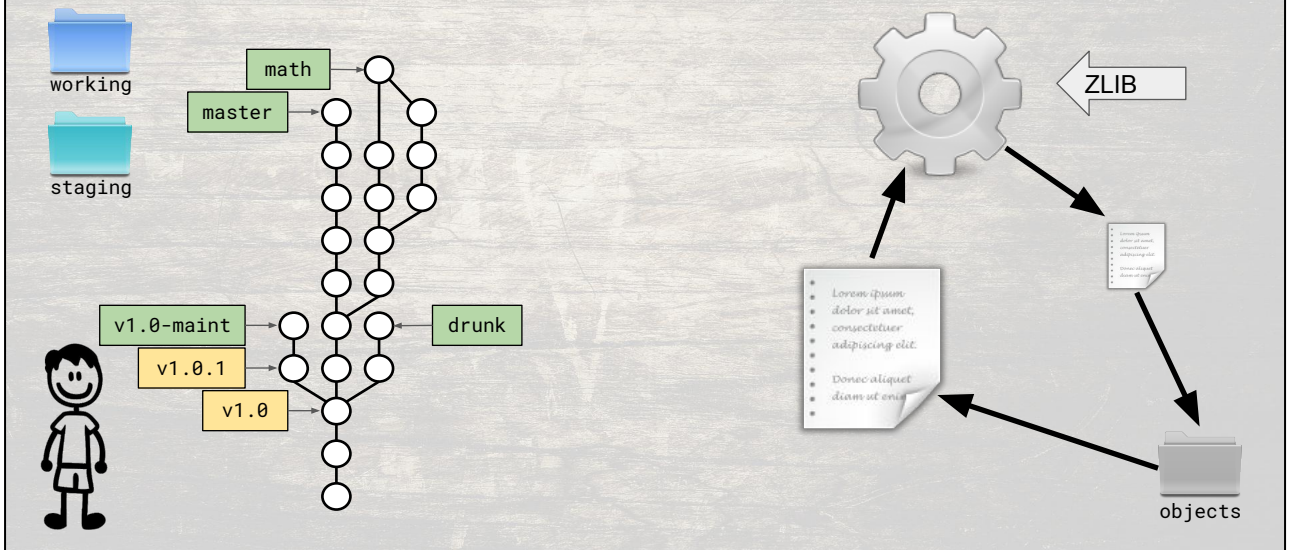
# Eliminating Duplication



As you can see, the real benefits of this system arise from reuse of trees and blobs across snapshots.

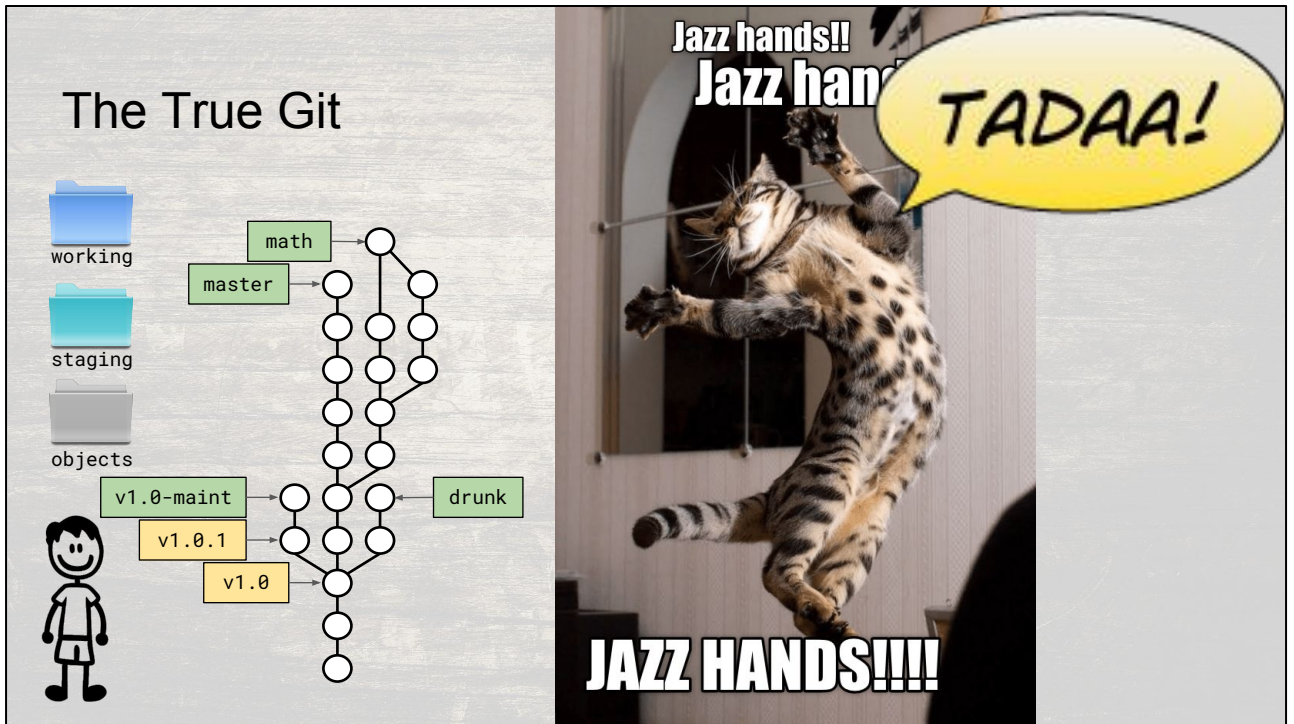
By converting every snapshot directory in the old system to object files in the new system, you can drastically reduce the number of files that are stored on disk. Now, instead of storing perhaps 50 identical copies of a rarely changed file, you only need to keep one.

# Compressing Blobs



Eliminating blob and tree duplication significantly reduces the total storage size of your project history, but that's not the only thing you can do to save space.

Source code is just text. Text can be very efficiently compressed using common compression algorithms. If you compress every blob before computing its SHA1 and saving it to disk you can further reduce the total storage size of the project history by a significant amount.

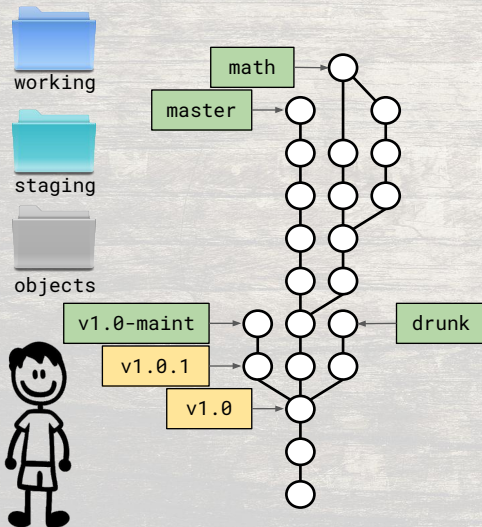


TADAAA!

The VCS you have constructed is now a reasonable approximation of Git.



# The True Git



- This is pretty much Git
- Nicer command line tools for all these operations
- Many, many other tools

The main difference is that Git gives you very nice command lines tools to handle such things as creating new snapshots and switching to old ones (Git uses the term “commit” instead of “snapshot”), tracing history, keeping branch tips up-to-date, fetching changes from other people, merging and diffing branches, and hundreds of other common (and not-so-common) tasks.

## Commands: Getting Started

- First, tell Git who you are:
  - `git config --global user.name "My Name"`
  - `git config --global user.email "my@email.address"`
- Get help:
  - `git <command> -h`
  - `git help <command>`
- Start a new Git repository:
  - `git init`

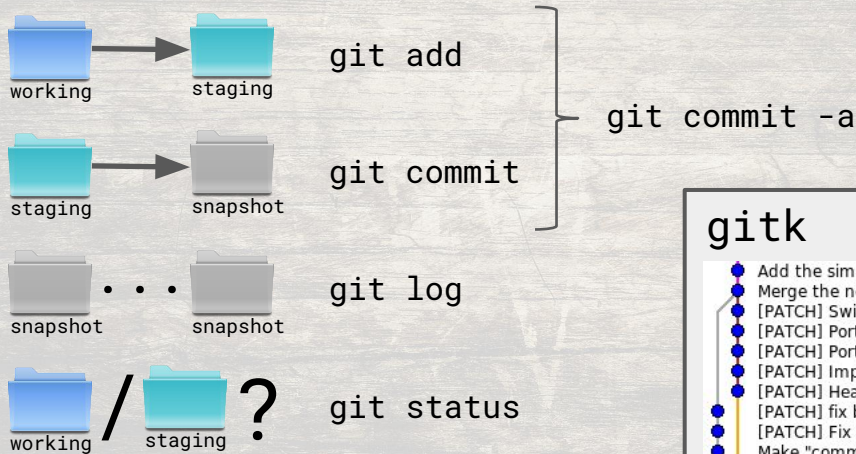
Finally, to get you started with simple usage of Git, here are some basic commands.

First, there are a couple of commands to make sure that your snapshots get the correct author information. You only need to run this once per machine you work on.

You can always get help on specific commands. “-h” gives you a short overview, “git help” gives you the reference docs for a command.

To start a new repo in the current directory, run “git init”. The current directory will be your working directory, and the objects, staging area, branches, tags, and other stuff will live in the “.git” subdirectory.

## Commands: Making snapshots



### gitk

- Add the simple scripts I used to do a merge with conf
- Merge the new object model thing from Daniel Barkal
- [PATCH] Switch implementations of merge-base, port
- [PATCH] Port fsck-cache to use parsing functions
- [PATCH] Port rev-tree to parsing functions
- [PATCH] Implementations of parsing functions
- [PATCH] Header files for object parsing
- [PATCH] fix bug in read-cache.c which loses files whe
- [PATCH] Fix confusing behaviour of update-cache --re
- Make "commit-tree" check the input objects more car
- Make "parse\_commit" return the "struct revision" for t
- Do a very simple "merge-base" that finds the most re
- Make "rev-tree.c" use the new-and-improved "mark\_r

To move changes from your working directory into the staging area, use “git add”.

To make a new snapshot from the staging area, use “git commit”. Use “git commit -a” to make a new commit directly from all changed files in the working directory.

Use “git log” to get a list of all commits on the current branch.

“gitk” is a nice GUI version of this, that also draws a nice graph of the relationship between commits.

Finally, “git status” gives you a summary of the current staged and unstaged changes.

## Commands: Diffing



VS.



`git diff`



VS.



`git diff --staged`



VS.



`git diff HEAD`



VS.



`git diff <from> <to>`

To see the differences between the working directory and the staging area, use “git diff”.

If you want to see the changes scheduled to become the next commit, use “git diff --staged”.

“git diff HEAD” shows you the diff between the working directory and the latest commit.

Finally you can use “git diff <from> <to>” to see the differences between any two commits in the repository.



## Commands: Branches & Tags

- `git branch`
  - `git branch <branch>`
  - `git checkout <branch>`
  - `git tag -l`
  - `git tag <tag>`
- } `git checkout -b <branch>`

“git branch” shows a list of your current branches.

“git branch” is also used to create new branches.

Use “git checkout” to switch to a different branch. “git checkout -b” is a handy shortcut when you want to start working on a new branch.

“git tag -l” lists existing tags.

Use “git tag” to create new tags.

## Commands: Fetching & Merging

- `git remote add <name> <URL>`
  - `git fetch <name>`
  - `git merge <name>/<branch>`
- } `git pull`

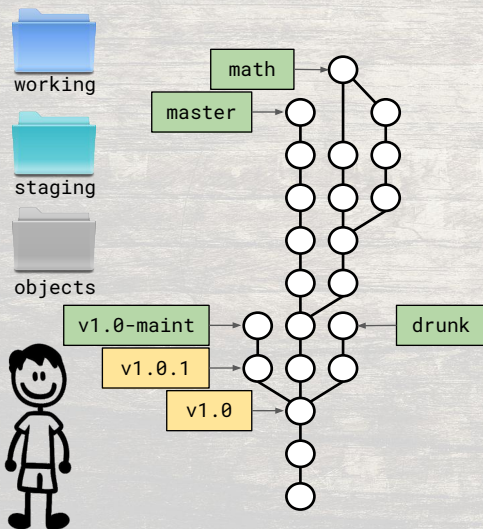
To start talking to another Git repo, use “git remote add”. This gives you a short name used to refer to the other repo (called a “remote”).

Use “git fetch” to fetch the branches from that remote. These are now available in your repo as “<name>/<branch>”, and you can “git log” or “git diff” these branches.

To merge one of these branches into the current branch, use “git merge”.

git pull is a shorthand for those two commands, but don't use it if you don't want to merge.

## Summary



- Keep this parable in mind

- Git is simple and powerful

- One more thing:

`git reflog`

As you continue to learn Git, keep this parable in mind. Git is really very simple underneath, and it is this simplicity that makes it so flexible and powerful.

One last thing before you run off to learn all the Git commands: remember that it is almost impossible to lose work that has been committed. Even when you delete a branch, all that's really happened is that the pointer to that commit has been removed. All of the snapshots are still in the objects directory, you just need to dig up the commit SHA1.

In these cases, look up “git reflog”. It contains a history of what each branch pointed to and in times of crisis, it will save the day.

## Where to go next?

- Git homepage: <http://git-scm.com>
- Pro Git: <http://git-scm.com/book>
- GitHub: <http://github.com>
- Learn Git Branching: <https://learngitbranching.js.org>
- Git Ready: <http://gitready.com>

Here are some resources that you can follow as your next step.

Now, go, and become a Git master!



# Thanks!

## Questions?

@jherland

<[jherland@cisco.com](mailto:jherland@cisco.com)>

<[johan@herland.net](mailto:johan@herland.net)>

