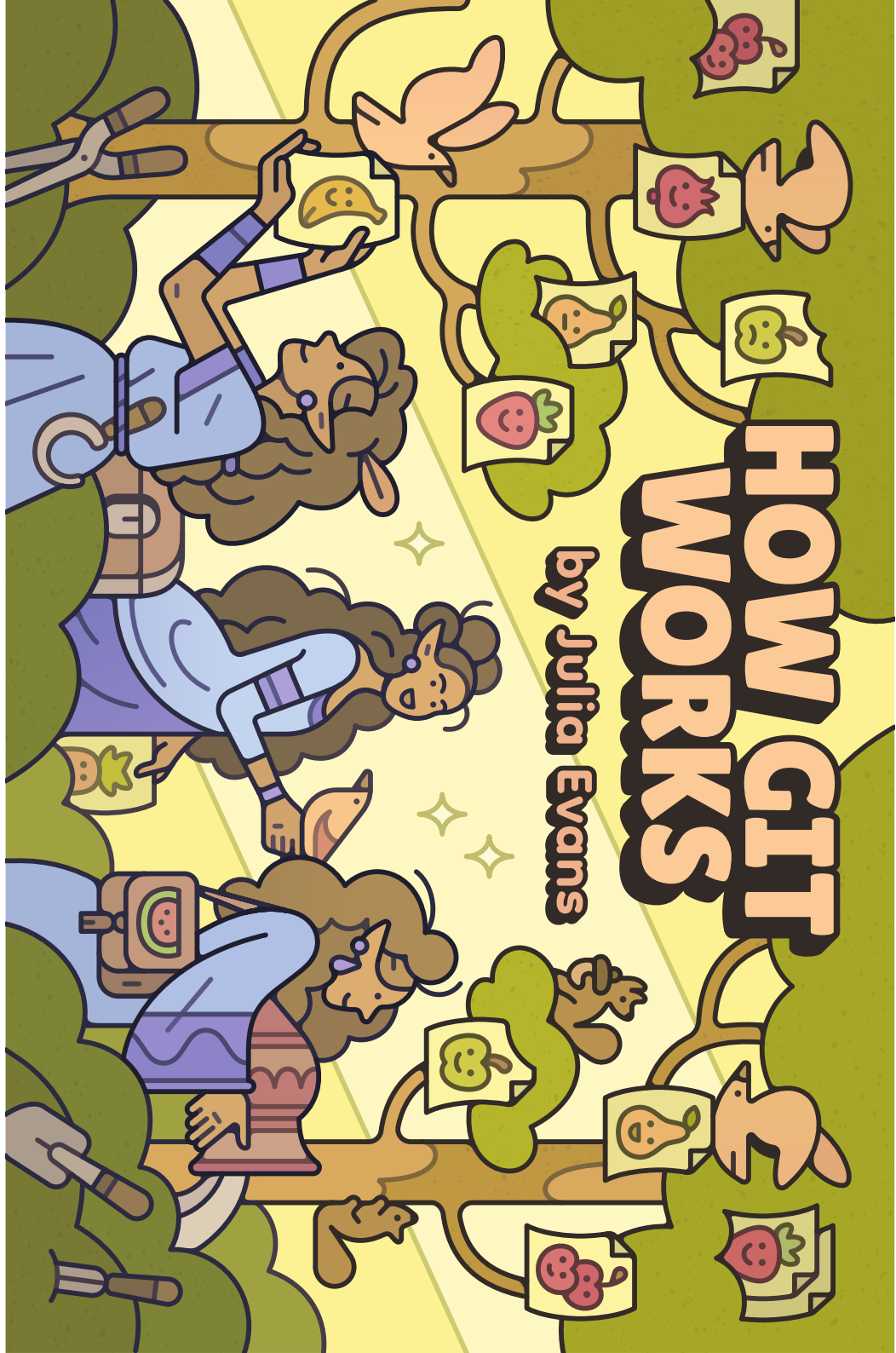
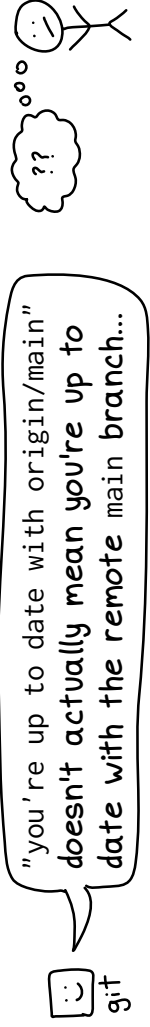




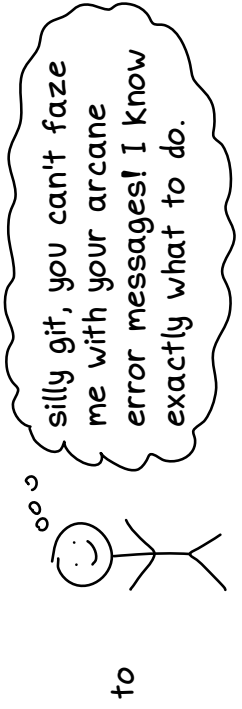
♡ this?
more at
★ wizardsines.com ★

about this zine

If you find git confusing, don't worry! You're not alone. It's VERY normal to be perplexed by it even if you've been using git for a long time.



This zine's goal is to take you from:



Once you know what's going on under the hood, you can get yourself out of any git mess.
☺ ☺ ☺ ↓ let's go learn! ↓ ☺ ☺ ☺

thanks for reading

As always, my favourite way to learn more about git is to experiment.
Make a new repository for testing! Make branches in it!
Try a rebase! See what happens!

There are also a million tools that can make git easier, for example:

- ★ a **shell prompt**. I use the one built into fish
- ★ an **editor integration**. I use vim-gitgutter
- ★ a **merge conflict tool**. I use meld
- ★ tools to **display diffs**, like delta
- ★ a **GUI**, like lazygit or GitUp on Mac OS



This zine comes with a printable cheat sheet! It's here:

<https://wizzardzines.com/git-cheat-sheet.pdf>

acknowledgements

Cover illustration: Vladimir Kašiković
Pairing: Marie Claire LeBlanc Flanagan
Technical review: James Cogan
Copy editing: Gersande La Flèche
and thanks to all 66 beta readers

reflog

a reflog is a log of
commit IDs

I use the reflog to find "lost" commits: it contains every commit ID that the branch/tag/HEAD has pointed to.

how to use the reflog

- ① run git reflog
- ② sadly stare at output until you find a log message that looks right
- ③ look at the commit
git show \$COMMIT_ID
git log \$COMMIT_ID
- ④ repeat until you find the thing
- ⑤ use something like
git reset --hard \$COMMIT_ID or
git branch \$NAME \$COMMIT_ID
to put the commit on a branch

some differences between git log main and git reflog main

- ★ reflog entries older than 90 days might get deleted by git gc
- ★ the reflog can show you where your branch was before a rebase. git log can't
- ★ the reflog isn't shared between repositories. git log is.
- ★ if I'm looking at the reflog, I'm having a bad day

which reflog to use?

The main two I use are:
git reflog
→ every single commit you've ever had checked out
→ has everything but very noisy
→ it's the reflog for HEAD
git reflog BRANCH
→ just the history for that branch, might be less noisy

the reflog kind of sucks

⚡ if you delete a branch, git deletes its reflog
⚡ if you drop a stash entry, you can't use the reflog to get it back
⚡ reflog entries don't correspond exactly to git commands you ran
But it's the best we have.

git fsck: the last resort

If a commit isn't in the reflog (for example if you "lost" it with git stash drop), there's still hope! You can use git fsck to list every commit ID that's unreferenced. I've never done this though: I try to avoid getting into this situation.

table of contents

☹️ commits

meet the commit.....	4
inside the commit.....	5
the diff algorithm.....	6
the staging area.....	7

🔗 merging

meet the merge.....	16
combining diverged branches.....	17
merge conflicts ⚡.....	18
merge commits.....	19

🌿 branches

meet the branch.....	8
what's a branch?.....	9
knowing where you are.....	10
detached HEAD state.....	11
references.....	12
lost commits.....	13

☁️ remotes

meet the remote.....	20
diverged remote branches.....	21
fixing diverged remotes.....	22
remote branch caching.....	23

💣 dealing with disasters

losing your work.....	24
git reset.....	25
the reflog.....	26

🔍 inside .git..... 14-15

meet the commit

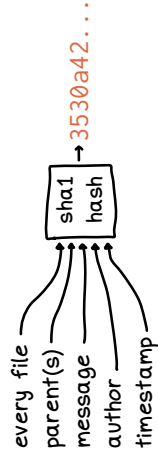
commits never change

once you've made a commit, it's set in stone:

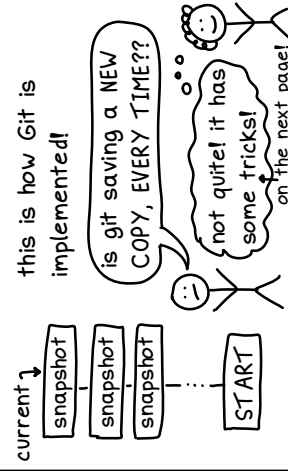
- the **files in it** never change
- its **diff** never changes
- its **history** never changes
- the **message/author** never change

commit hashes

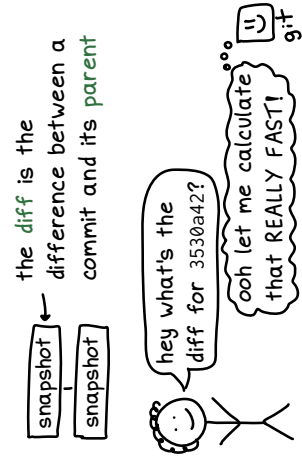
commits never change because their **ID** is calculated from their **contents**



you can also think of commits as a **pile of snapshots**



diffs are calculated from **snapshots**



git has no undo

there's no **unadd**
uncommit
unmerge
unrebase

instead, git has a single dangerous command for undoing:

git reset

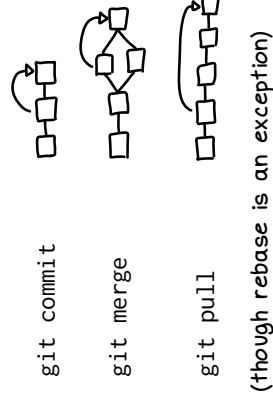
how git reset works

git reset HEAD^ :

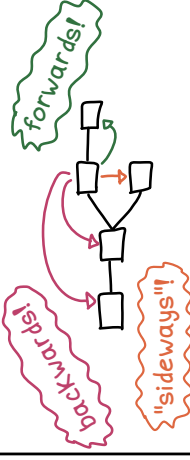
- ① finds the commit ID corresponding to HEAD^ (for example a2b3c4)
- ② forces your current branch to point to a2b3c4
- ③ unstages all changes

reset

most git commands move the current branch **forwards**



git reset can move the current branch **anywhere**



this makes it possible to undo, but you can also really mess up your branch

--hard: the danger option

git reset \$COMMIT_ID

keeps all the files in your working directory **exactly the same**.

git reset --hard \$COMMIT_ID

Throws away all your uncommitted changes. Useful but dangerous.

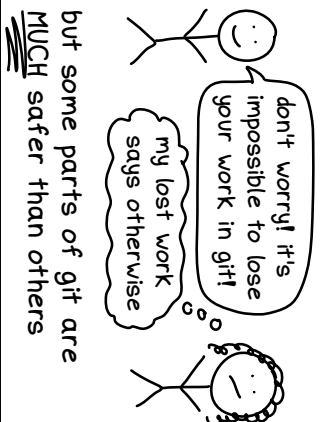
problems reset can cause

⚠ it's easy to **"lose" commits**, especially if you move a branch backwards

⚠ if you use --hard, you can **permanently lose** your uncommitted changes

losing your work

people are always saying:



commits in the history of a branch / tag

📁 never change



📁 never change, except...

⤵️ they're hard to find

⤵️ they'll eventually get deleted by git's **garbage collection**

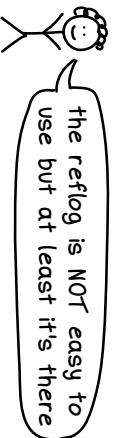
↖️ page 12-13

(usually not for a few months though)

branches and HEAD

📁 change ALL THE TIME

⤵️ BUT there's a history of all the changes in the **reflog**



staging area

📁 changes ALL THE TIME

~~⤵️~~ no history

⤵️ just gotta be careful

the stash

~~⤵️~~ git stash drop deletes entries forever

... but you can technically get them back by scrolling up in your terminal to find the commit ID (if you're lucky) or by using git fsck (if not)

(I only really use git stash to throw away work)

inside the commit

you can see for yourself how git is storing your files!

You just need one command: `git cat-file -p`

First, get a commit ID.

You can get one from `git log`

① read the commit

```
$ git cat-file -p 3530a4
tree 22b920 ← directory ID
parent 56cfdc
author Julia <julia@fake.com> 1697682215 -0500
committer Julia <julia@fake.com> 1697682215 -0500
commit message goes here
```

↖️ I just use git cat-file for fun and learning, never to get things done

② read the directory

```
$ git cat-file -p 22b920
100644 blob 4fffb2 .gitignore
100644 blob e351d9 404.html
100644 blob cab416 Cargo.toml
100644 blob fe442d hello.html
040000 tree 9de29f src
file ID
(IDs are actually 40 characters)
```

③ read a file

```
$ git cat-file -p fe442d
<DOCTYPE html>
<html lang="en">
<body>
<h1>Hello!</h1>
</body>
</html>
```

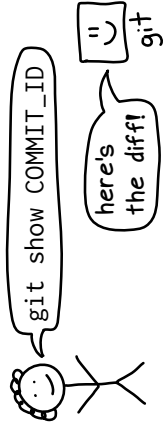
and we're done!

↖️ **fe442d** is the sha1 hash of the contents of the file. It's called a "blob id". Commit and tree IDs are hashes too.

Using a hash to identify each file is how git avoids duplication: if the file's contents don't change, the hash won't change, so git doesn't need to store a new version!

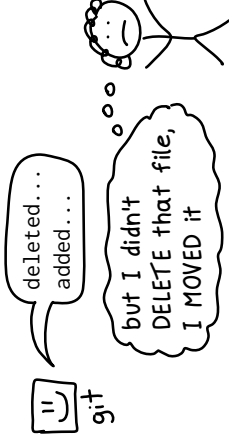
the diff algorithm

git is **CONSTANTLY** showing you diffs



and it makes it seem like git thinks in terms of diffs

have you ever noticed your git diffs don't make sense?



in git, moving a file is the same as deleting the old one and adding the new one

```
git mv old.py new.py
      ↑
      ↓ same
cp old.py new.py
git rm old.py
git add new.py
```

git is just **guessing** about your intentions



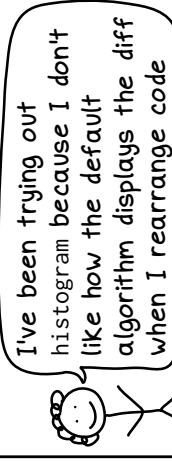
diff is an algorithm

the algorithm:

- takes 2 versions of the code
- compares them
- tries to summarize it in a human readable way

(but it doesn't always do a great job)

git has many diff algorithms



how to try it out:

```
git diff --histogram
```

remote branch caching

the "up to date" in git status is misleading

```
$ git status
```

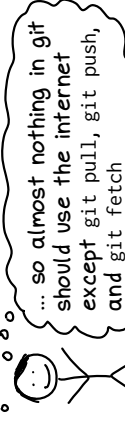
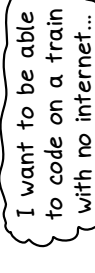
Your branch is up to date with origin/main

This does **NOT** mean that you're up to date with the remote main branch. Let's talk about why!

some old version control systems only worked if you were online

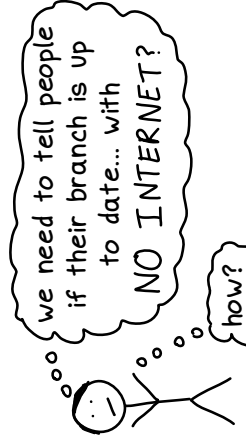


git works offline



git developers

this makes git status weird



solution: CACHING

remote name branch name
↑

Every remote branch has a local cache named like **origin/mybranch**. Git doesn't call it a cache though, it's called a "remote tracking branch"

local branch

mybranch

remote branch

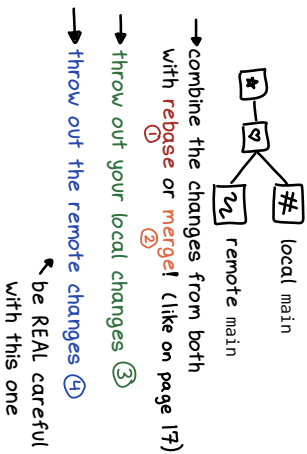
origin mybranch

only updated on git pull, git push, git fetch
↑
git push origin mybranch updates this

(Git has no easy way to see when origin/mybranch was last updated)

fixing diverged remotes

ways to reconcile two diverged branches

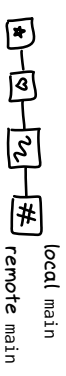


reasons to throw away changes

- I'll throw away local changes if I accidentally committed to main instead of a new branch.
- I'll throw away remote changes if I want to amend a commit after pushing it, and I'm the only one working on that branch.

① rebase

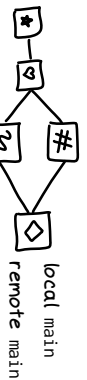
git pull --rebase
git push



Many people like to configure git config pull.rebase true to make this the default when they run git pull.

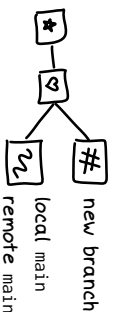
② merge

git pull --no-rebase
git push



③ throw away local changes

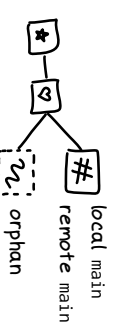
optional: save your changes on main to newbranch so they're not orphaned
git switch -c newbranch
git reset --hard origin/main



④ throw away remote changes

(DANGER!)

git push --force

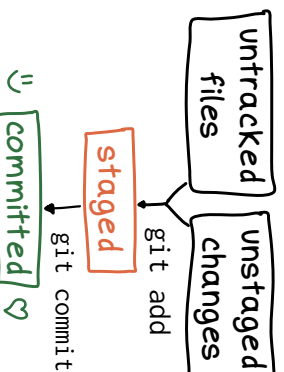


I only do this if there's nobody else working on the branch.

the staging area

git has a 2-stage commit process

- ① tell git what you want to **stage** (git add, git rm, git mv, etc.)
- ② make the **commit** with git commit



git uses 3 terms interchangeably for the staging area

- ① staged (like --staged)
 - ② cache (like --cached)
 - ③ index (like --keep-index)
- it's total chaos but they're all the same thing

tip: you can use git add -p to commit only certain parts of a file

I only want to commit my actual changes, not all the random debugging code I put in

gotcha: git diff only shows unstaged changes

You can use:

- git diff HEAD to see all changes you haven't committed
- git diff --cached to see staged changes

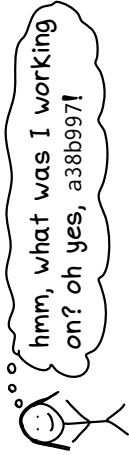
gotcha: git commit -a doesn't automatically add new files

I CONSTANTLY forget to add new files and then get confused about why they didn't get committed

meet the branch

theoretically you could
use git without branches

You could keep track of your
commit IDs manually:



But most people use branches.

the only difference
between the **main** branch
and **any other branch** is
how you treat them

For example: it's common to
never commit to main directly,
and instead commit to other
branches which you merge into
main when you're done.

every branch has 3 things

- a **name** (like main)
- a **latest commit** (like 2e9ffc)
- a **reflog** of how that branch
has evolved over time

page 26

Branches also sometimes have
a corresponding remote branch
which they "track".

all changes to a branch
are recorded in its **reflog**

The reflog records every
rebase, **amended commit**, **pull**,
merge, **reset**, **commit**, etc. You
can look at the reflog like this:

git reflog BRANCHNAME

reflog stands for "**reference log**"
(not re-flog ☹️)

branches are core to how
git stores your work

If your commits are "**lost**"
(not on a branch):

page 13

☹️ they'll become incredibly
difficult to find

☹️ git's garbage collection will
eventually **delete them**

git will let you do **literally**
anything with a branch

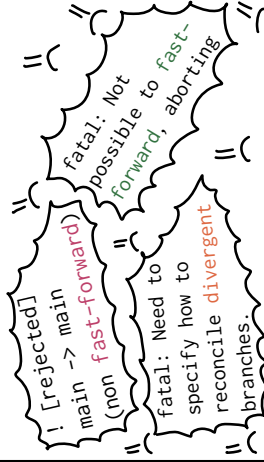
→ when you push/pull a branch,
the local branch name
doesn't have to match the
remote branch name

→ you can **remove** commits from a
branch with git reset

Git often won't protect you
from messing up your branch!

diverged remote branches

when pushing/pulling, the
hardest problems are
caused by **diverged branches**



how to tell if your branches
have **diverged**: **git status**

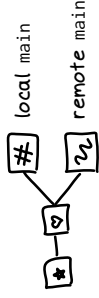
\$ git fetch ← get the latest
\$ git status ← remote state first

Your branch and 'origin/main'
have **diverged**, and have 1 and 1
different commits each,
respectively.

(use "git pull" to merge the
remote branch into yours)

what are **diverged branches**?

both sides have commits that
the other doesn't, like this:



I like to fix my diverged branches
before making more commits.

git fetch and **git pull**

git fetch just fetches the latest
commits from the remote branch

git pull origin main has 2 parts:

- ① run git fetch origin main
- ② run git merge origin/main
or sometimes rebase

(More about how to tell git pull
to merge/rebase on page 16f)

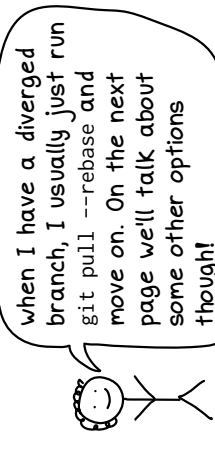
there are 4 possibilities
with a remote branch

① up to date ✓ □ □ □ □ □ LOCAL
□ □ □ □ □ REMOTE

② need to **pull** □ □ □ □ □ LOCAL
□ □ □ □ □ REMOTE

③ need to **push** □ □ □ □ □ LOCAL
□ □ □ □ □ REMOTE

④ **DIVERGED** !! □ □ □ □ □ LOCAL
□ □ □ □ □ REMOTE
↑
need to decide how to solve it



meet the remote

any repository you're pushing to / pulling from is called a **"remote"**

remotes can be:

- ★ hosted by GitHub/GitLab/etc.
- ★ on your own server
- ★ just a folder on your computer

git push syntax

(same for git pull)

```
git push origin main
```

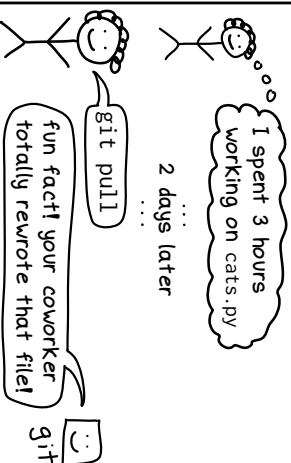
remote ↗ ↖ remote
name branch

the default name for a remote is origin but you can name it anything

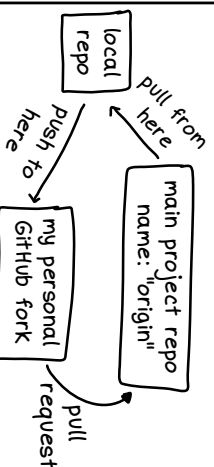
[tip!] I like to configure

push.autoSetupRemote true
to automatically set up tracking the first time I push a new branch

remotes are where the drama happens



example: I use 2 remotes when contributing to open source projects



remotes are configured in .git/config

every remote has a **name** and **URL**

```
[remote "origin"]
url = git@github.com:jvns/myrepo
branch ["main"]
remote = origin
merge = refs/heads/main
```

this sets up "tracking" between local **main** ↔ remote **main** on origin so that git knows what to push to when you run git push or git pull

protocols

Git has 3 main protocols for remotes. The protocol is embedded in the URL.

HTTP (I use this if I only want to **pull**)

<https://github.com/jvns/myrepo>

SSH (I use this if I need to **push**)

[git@github.com:jvns/myrepo](ssh://git@github.com:jvns/myrepo)

local: <file:///home/bork/myrepo>

what's a branch?

q

You can think about a Git branch in 3 different ways:

① just the commits that "branch" off

This is how I usually think about branches: **armadillo** branches off **main**



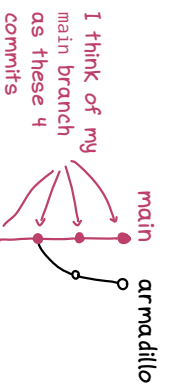
How this shows up in git:

Git **DON'T** KNOW that armadillo is branched off of main: for all it knows, main could be branched off of armadillo! You need to tell it when you merge or rebase, for example:

```
git checkout main
git merge armadillo
```

② every previous commit

Even though git doesn't treat the main branch in any special way, I think of main differently from other branches.



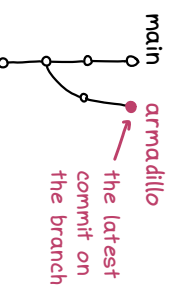
How this shows up in git:

It's what git log **BRANCHNAME** shows you! How git log main works:

```
main ← start here
parent |
parent |
parent |
... |
```

③ just the commit at the end

This is how branches are actually implemented in git.



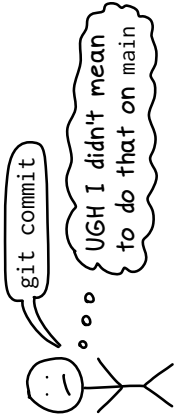
How this shows up in git:

It's how branches are stored internally: a branch is fundamentally a **name** for a **commit ID**.

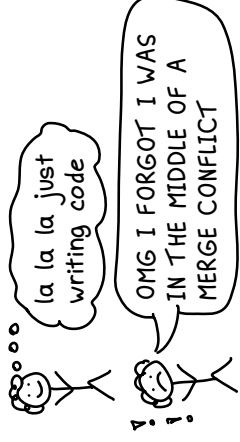
```
.git/refs/heads/main ← branch name
a276f62
↑
ID of the latest commit on the branch
```

knowing where you are

many git disasters are caused by accidentally running a command while on the wrong branch...



... or by forgetting you're in the middle of a multistep operation



I always keep track of 2 things

- ① am I on a **branch**, or am I in **detached HEAD state**? ^{next page!}
- ② am I in the middle of some kind of **multistep operation**? (rebase, merge, bisect, etc.)

I keep my current branch in my shell prompt

~/work/homepage (main) \$
to me it's as important as knowing what directory I'm in
git comes with a script to do this in bash/zsh called git-prompt.sh, but there are tons of ways to get this info (run git status a lot! use a GUI! use a different shell prompt!)

decoder ring for the default git shell prompt

← on a branch, everything is normal

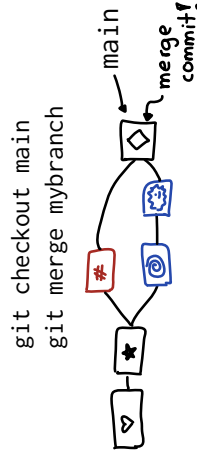
the double brackets (()) mean "detached HEAD state"

this prompt can only happen if you explicitly git checkout a commit/tag/remote-tracking branch

in the middle of a multistep operation:
merge/rebase/cherry-pick/bisect

```
(main)
((2e832b3...))
((v1.0.13))
(main|MERGING)
(main|REBASE 1/1)
(main|CHERRY-PICK)
(main|BISECTING)
```

merging 2 diverged branches creates a commit



Merge commits have a few surprising gotchas!

gotcha: you can keep coding during a merge

If you forget you're doing a merge, it's easy to accidentally keep writing code and add a bunch of unrelated changes into the merge commit.

I use my prompt to remind me.

← page 10

gotcha: git show doesn't always tell you what the merge commit did

It'll sometimes just show the merge commit as "empty" even if the merge did something important (like discard changes from one side).

gotcha: merging isn't symmetric

these merges result in the same code, but the first parent of the merge commit is different: it's the current commit you had checked out when you merged.

merge mybranch into main

```
git checkout main
git merge mybranch
```

merge main into mybranch

```
git checkout mybranch
git merge main
```

A merge commit with the "wrong" first parent makes HEAD^ or HEAD^^^^ behave in an unexpected way: ^ refers to the first parent.

tip: see what a merge did with

git show --remerge-diff

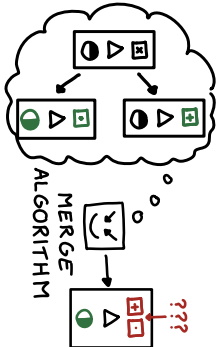
git show --remerge-diff COMMIT_ID

will **re-merge** the parents and show you the **difference** between the original merge and what's actually in the merge commit

merge commits

MERGE CONFLICTS " " "

merge conflicts happen because both branches edited the same lines of code



some ways to resolve merge conflicts

- ① edit the weird text file by hand
 - ② use a dedicated merge conflict tool
 - ③ abort the merge and rewrite the code you were merging from scratch
 - ④ if the conflict is in an autogenerated file, delete and regenerate it
 - ⑤ go have a conversation with the other person about what to do
- often the easiest way!
you can configure git so that git mergetool opens conflicts in your favourite tool. I like meld on linux.
might be easier if there was a big refactor! You can do this with git merge --abort
great for package-lock.json in node!

the weird text file

Git merge conflicts are confusing because they're not displayed in a consistent way:

```
<<<<<< HEAD
def parse(input):
    return input.split("\n")
    ||||| b9447fc
def parse(input):
    return input.split("\n\n")
=====
def parse(text):
    return text.split("\n\n")
>>>>>> a29b3cf
```

↑ **top**
← the original (configure merge.conflicstyle diff3 to get this)
↓ **bottom**

Why?

finishing up

To finish, you need to run one of:

```
git commit           (for git merge)
git rebase --continue (for git rebase)
git cherry-pick --continue (git cherry-pick)
git revert --continue (for git revert)
```

Before that, I might:

- ★ look at my changes with git diff main
- ★ check for unresolved conflicts with git diff --check

detached HEAD state

how git knows what your

current branch is: .git/HEAD

.git/HEAD is a file where git stores either:

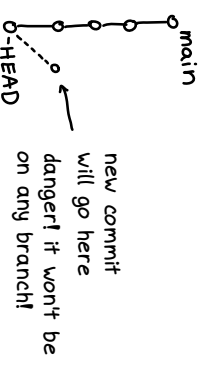
- ① a **branch name**: the current branch
- ② a **commit ID**
this means you don't have a current branch. git calls this "detached HEAD state"

by itself, .git/HEAD being a commit ID is okay

it's a great way to look at an old version of your code!
I don't do it often, but it's super useful during a rebase!

the **only problem** is that new commits you make can get "**lost**"

page 13



git has a little language for referring to commits

the current commit HEAD
the previous commit HEAD~
3 commits ago HEAD~^
3 commits ago HEAD~3

The full documentation (with main@3 days ago & more) is at:
man gitrevisions

ways you can end up in detached HEAD state

You will end up in detached HEAD state if you checkout:

- a **tag**
\$ git checkout v1.3
- a **remote-tracking branch**
\$ git checkout origin/main
- a **commit ID**
\$ git checkout a3ffab9

if you accidentally create commits in detached HEAD state, it's **SUPER** easy to avoid losing them

just create a new branch!
git checkout -b oops
(you can also create a branch with git switch -c if you prefer)

references

git often uses the term
"reference" in error messages

```
$ git switch asdf
fatal: invalid reference: asdf
```

\$ git push
To github.com:jvns/int-exposed
! [rejected] main -> main
error: failed to push some refs to
'github.com:jvns/int-exposed'

"ref" and "reference"
mean the same thing

"reference" often
just means "branch"

branch name
fatal: invalid ~~reference~~: asdf
error: failed to push some ~~refs~~ to
'github.com:jvns/int-exposed'

branches
in my experience, it's:
44% "branch"
3% "tag"
3% "HEAD"
0.01% something else

"reference" is an umbrella term

well, I COULD check if the thing
we failed to push is a branch or
tag or what, and customize the
error message based on that...

git ° ° ° seems complicated, let's
just print out "reference"

5 types of references

References are files. They're almost all in `.git/refs`.
Here's every type of git reference that I've ever used:

HEAD: `.git/HEAD`
branches: `.git/refs/heads/$BRANCH`
tags: `.git/refs/tags/$TAG`
remote-tracking branches: `.git/refs/remotes/$REMOTE/$BRANCH`
stash: `.git/refs/stash`

all of these files
contain a commit ID,
but the way that
commit ID is used
depends on what **type**
of reference it is

(stash is a weird reference: when you run git stash, git creates a "temporary" commit.
Git stores the commits you have stashed in the stash's reflog: `.git/logs/refs/stash`)

combining diverged branches

there are 3 options for
combining branches

rebase merge squash

for example, let's say we're
combining these 2 branches:



rebase

pro: you can keep your git
history simple:

□ - □ - □ - □ - □ - □

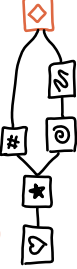
pain: harder to learn " "
harder to undo " "
easier to mess up " "

(I love rebase though!)

① git rebase



② git merge



③ git merge --squash



all 3 methods result in
the EXACT SAME CODE

some differences are:

- ★ the diff git shows you for the final commit
- ★ the commit IDs
- ★ the specific flavour of suffering the method causes

merge

pro: if you mess something up,
the original commits are
still in your branch's history

pain: when I look at histories
like this I feel dread " "



squash

pro: have 20 messy commits?
nobody needs to know!
and it's pretty simple to use

pain: "ugh, someone squashed
their 3000-line branch
into 1 commit" " "

meet the merge

merging is a huge thing in git

But the terminology around merging is a bit confusing:

git merge

isn't the only way to combine branches: you can also use `git rebase!`

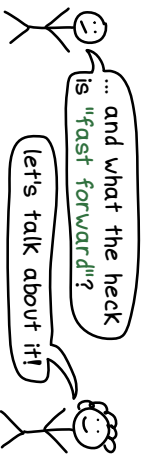
merge conflicts

can happen if you do any of these:

`git merge`
`git rebase`
`git cherry-pick`
`git revert`
`git stash pop`

merge commits

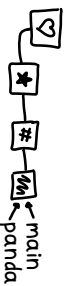
are only created by `git merge`



there are 3 situations when combining branches

① **easy**: no divergence ("fast-forward")

`git merge` moves the main branch forward to where the panda branch is, like this:



② **harder**: diverged branches, **no conflicts**



you have to decide whether to merge or rebase, but it'll succeed

③ **hardest**: diverged branches with **merge conflicts**



you have to decide whether to merge or rebase, AND fix a merge conflict

`git merge` checks for these 3 situations in order

① is this the "easy" situation?

no → fast forward

yes → fast forward

② run the merge. Is there a merge conflict?

yes → done!

no → done!

③ tell you to manually resolve the conflict

git pull needs to combine branches too

`git pull` will ONLY fast forward by default.

If it can't, it'll ask you to specify if you want to rebase or merge.

`git pull --rebase`

runs `git rebase`

`git pull --no-rebase`

runs `git merge`

commits in git are usually saved forever

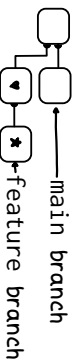
But even if git still has your commits, they're not always easy to find.

Some ways commits get "lost":

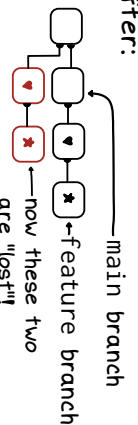
- `git commit --amend`
- `git rebase`
- deleting an unmerged branch
- `git stash drop`

how commits can get lost:
`git rebase`

before:



after:



the three levels of losing commits

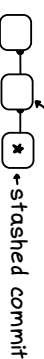
annoying: the commit isn't in the history of any branch/tag, but it's relatively easy to find

nightmare: you need to search every single commit to find it

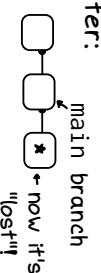
disaster: it's been deleted

how commits can get lost:
`git stash drop`

before: main branch



after:

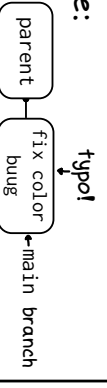


stash is the only way I've seen the "nightmare" situation happen.

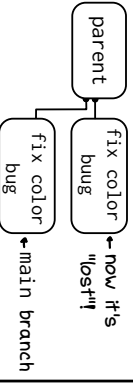
how commits can get lost:

`git commit --amend`

before:



after:



you can find lost commits

I find it very comforting to know that git keeps my lost commits around. How to find them:

annoying: use the `reflog` → page 26

nightmare: use `git fsck`

disaster: impossible (but this has never happened to me)

inside .git

Here's an overview of the main parts of the .git folder!

Don't worry if you don't understand all this yet! We'll get to it.

HEAD is a tiny file that just contains the name of your current branch

.git/HEAD
ref: refs/heads/main

HEAD can also be a commit ID, that's called "detached HEAD state"

a branch is stored as a tiny file that just contains a commit ID. It's stored in a folder called refs/heads.

.git/refs/heads/main
75bbae4 ← (actually 40 characters)

tags are in refs/tags, the stash is in refs/stash More on page 12.

a commit is a small file containing its parent(s), message, tree, and author

.git/objects/75/bbae4
tree c4e6559
parent 037ab87
author Julia <x@y.com> 1697682215
committer Julia <x@y.com> 1697682215
commit message goes here

the files in /objects/ are all compressed, the best way to see them is with git cat-file -p HASH

trees are small files that list the permissions, type, ID, and name of every file in a directory. The files in it are called "blobs".

.git/objects/c4/e6559
100644 blob e351d93 404.html
100755 blob cab4165 hello.py
040000 tree 9de29f7 lib

if you recognize 644 and 755 as unix permissions: beware that they're super restricted! only 644 and 755 are allowed

blobs are the files that contain your actual code

.git/objects/ca/b4165
print("hello world!!!")

storing a new blob with every change can get big, so git gc periodically packs them for efficiency in .git/objects/pack

the reflog stores the history of every branch, tag, and HEAD

.git/logs/refs/heads/main
2028ee0 c1f9a4c
Julia Evans <x@y.com>
1683751582
commit: no ligatures in code

each line of the reflog has:
← before/after commit IDs
← user
← timestamp
← log message

remote-tracking branches store the most recently seen commit ID for a remote branch

.git/refs/remotes/origin/main
a9bbcae

when git status says "you're up to date with origin/main", it's just looking at this. More on page 23.

.git/config is a config file for the repository. it's where git stores the configuration for your remotes (and other local config settings)

.git/config
[remote "origin"]
url = git@github.com:jvnns/int-exposed
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
remote = origin
merge = refs/heads/main

git has global and local settings, the local settings are here and the global ones are in ~/.gitconfig

hooks are optional scripts that you can set up to run (e.g. before a commit) to do anything you want

.git/hooks/pre-commit
#!/bin/bash
any-commands-you-want

the index is one of the only things in git that doesn't have a plain text format. You can see its contents with: git ls-files --stage (though in practice I just use git status)

the staging area stores files when you're preparing to commit

.git/index
(binary file)