An announcement about grading and homework deadlines

Version Control Systems

Has this ever happened to you?

"Oh no, my code broke but I can't remember what I changed 😭"

"Hmm 🤔 probably need this later, so I'll comment it out"

"Hey Tom, we should copy our whole project before we make any big changes 👍"

"Can't believe we deleted our whole project and have to redo it 💀💀💀"

Deleting a file accidentally

Remembering I use version control

Version control to the rescue

What version control lets you do

- Record the history of your code
- Go back to old versions
- Save and view descriptions of each change you make
- Share changes with others
- …and the same for other people's changes!

What questions do you have?

Featuring: Git

- Developed in 2005 for the Linux kernel
- Mature and modern VCS
- Kind of frustrating to learn initially
- But it's *everywhere*

Git is *not* GitHub

…or GitLab, or Bitbucket, or sourcehut.

Git is a *distributed* version control system (DVCS):

- Needs no centralized host
- Every Git operation happens locally, needing no internet connection
  - (except ones that copy changes to and from a remote copy of the code)
- GitHub often hosts the authoritative copy of a project, but its copy isn't special in any other way

What questions do you have?

Git repositories

- Encompass the current and historical state of a single file tree
  - i.e. track the contents of a directory over time
- All Git operations are performed within a repository
- Each repository is independent

What's in a repository?

Your normal files, plus a hidden directory that Git uses to hold history and metadata.

Files:

- Your source code (or other data) in its "current" state
  - In Git parlance, your *working tree*

Metadata:

- *Blobs*, containing snapshots of individual files from different points in time
- *Trees* (made up of blobs), containing snapshots of the whole directory from different points in time.
- *Commits*, which arrange trees into a *history*, including a message for each change
- *Branches*, which assign names to commits you frequently work with (generally ones representing the latest version of something you're working on)

What questions do you have?

Making a repository

```
$ git init myrepo
$
```

or

```
$ mkdir myrepo
$ cd myrepo
$ git init
$
```

```
$ tree .git
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
│   ├── <these template files have been omitted for brevity>
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

9 directories, 16 files
$
```

What questions do you have?

demo time!

Summary

- Repos have commits
- Commits have a tree and metadata
- Trees have blobs (files)
- `git show`
- Simple flow for working on code:
  - Edit
  - Add
  - Commit

Quick Q: in person lecture?

Git: greatest hits

(also see `man gittutorial`, `man gitglossary`)

Querying

`git status`

What's going on in the repo right now?

Shows various pieces of information:

- Current branch
- Uncommitted changes
  - Untracked files
  - Tracked (*staged*) files
- Hints about commands you might use next

`--short (-s)`: Shows short-format listing of uncommitted changed files, one per line.

```
$ git status
On branch main
nothing to commit, working tree clean

$ touch new-file
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be
committed)

    new-file

nothing added to commit but untracked files present (use "git
add" to track)
$
```

What is the staging area (a.k.a. the index)?

A snapshot of the working tree indicating what will become part of the next commit you make.

- Only snapshots files with modifications
- Only snapshots files you've added to it with `git add`
- Can be updated at any time with `git add`, `git reset`, and others
- Not shared: each copy of a repository has its own index
- Resets every time you make a commit

Ways to refer to a commit

(See `man gitrevisions` for more detail.)

- **Hash** (e.g. `46e2f3b`): either the full SHA-1 hash, or enough of the beginning of the hash to match only a single object.
- **Ref** (e.g. `main`): generally a local or remote branch name. Looks in `.git/refs/` under the hood.
  - The special ref HEAD refers to the commit or branch that you're currently "on." HEAD is what commands like `git diff --cached` and `git status` compare against, and it's the parent of new commits you make.

When you need to specify a commit or branch, you can generally use either of these methods.

git log

What history is visible from the given commit (or HEAD) for the given files?

Shows all the parents of a given commit. Defaults to HEAD.

`-- <paths>`: Shows only commits for the given paths. `--` stops Git from treating a path as a branch.

`--patch (-p)`: Show the diff for each commit.

`--oneline`: Show each commit as a single line.

```
$ git log
commit e67e22ba38560e1a644d09e04afc4374bd5a2ebc (HEAD ->
main)
Author: Thomas Hebb <tommyhebb@gmail.com>
Date:   Wed Oct 6 08:35:19 2021 -0400

     Add a longer file in directory2

<...>

$ git log --oneline add-symlink
ff553ab (add-symlink) Add a symlink to file1
46e2f3b Initial commit

$ git log --oneline -- file2
8320c08 (HEAD -> main) Fix file2 to match file zoo
fe7b7f5 Add file2
$
```

`git show`

What's in this object (usually a commit)?

Shows commits, blobs, trees, tags in a human-readable format. Defaults to showing HEAD. Like `git log`, but doesn't show parents.

`--no-patch`: Don't show a commit's diff (same as `git log` without `-p`).

`--stat`: Show only a summary of the diff (number of lines added and removed in each file).

```
$ git show 46e2f3b
commit 46e2f3b72116845599bc868cb8aa65ddf23c5b9d
Author: Thomas Hebb <tommyhebb@gmail.com>
Date:   Tue Oct 5 18:20:02 2021 -0400

     Initial commit

     I like this file from the file zoo. Let's make a repo
with it!

diff --git a/file1 b/file1
new file mode 100644
index 0000000..a28a390
--- /dev/null
+++ b/file1
@@ -0,0 +1 @@
+I'm a file
$
```

What questions do you have?

git blame

Find out what commit last changed each line of a file.

Note that any difference in a line, no matter how small, counts as a change: you sometimes have to go through several iterations of blame to find the change you're interested in.

<commit> --: Shows the file (and it's blame) as it was at the given commit.

-w: Skips commits that only changed a line's whitespace.

```
$ git blame file1
^46e2f3b (Thomas Hebb 2021-10-05 18:20:02 -0400 1) I'm a file
6d6a4d11 (Thomas Hebb 2021-10-19 18:59:31 -0400 2) I have two
lines
9360e395 (Thomas Hebb 2021-10-19 18:59:48 -0400 3) And now I
have three!

$ git blame 9360e395^ -- file1
^46e2f3b (Thomas Hebb 2021-10-05 18:20:02 -0400 1) I'm a file
6d6a4d11 (Thomas Hebb 2021-10-19 18:59:31 -0400 2) I have two
lines
$
```

`git diff`

What changed (since HEAD, since a given commit, or between two commits)?

By default, shows unstaged changes (i.e. the difference between the index and the working tree).

`--cached`: Shows the index compared to HEAD or the given commit.

`<commit>`: Shows the working tree compared to `<commit>`.

`<commit1> <commit2>`: Shows `<commit2>` compared to `<commit1>`.

```
$ echo 'A second line!' >>file1
$ git diff
diff --git a/file1 b/file1
index a28a390..6cec40f 100644
--- a/file1
+++ b/file1
@@ -1 +1,2 @@
 I'm a file
+A second line!
$
```

What questions do you have?

Operations on the working tree

`git add`

Update the index with a file from the working tree.

Takes a snapshot of the given file(s) and updates the index with that snapshot.

`--patch` (`-p`): Interactively select which changes within each file to add to the index (see next slide).

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be
committed)

    new-file

nothing added to commit but untracked files present (use "git
add" to track)
$ git add new-file
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   new-file
$
```

git add --patch/-p

Update the index with specific changes from the working tree.

Splits changes between the index and working tree into individual *hunks* (groups of changed lines), and asks you whether each one should be added to the index.

"y": Includes the hunk
"n": Omits the hunk
"s": Splits the hunk into smaller ones
"?": Explains other options

```
$ git add -p
diff --git a/directory2/entryway b/directory2/entryway
index d5ba6ac..4765745 100644
--- a/directory2/entryway
+++ b/directory2/entryway
<...>
@@ -33,4 +29,5 @@ shoot you."
 differences, but I had hoped to clear them up tonight by
inviting him. I did
 not kill him! Please help clear my name!"

-You make your way into the living room.
+"Perhaps that would be easier to do were you not named David
Knifehands," you
+suggest, and brush past David into the living room.
Stage this hunk [y,n,q,a,d,K,g,/,e,?]? y

$
```

`git rm`

Remove a file from the index *and* working tree.

Note that this deletes the file on disk as well as in the index, unlike `git add`, which only ever changes the index. Won't delete a file with uncommitted changes without `-f`.

`--cached`: Removes the file from the index, but leaves the working tree copy.

`-r`: Remove a directory and all files in it.

```
$ git rm file1
rm 'file1'
$ ls
directory1  directory2  file2  file3  missing-link

$ git rm directory2/entryway error: the following file has
changes staged in the index: directory2/entryway (use
--cached to keep the file, or -f to force removal)

$ git status -s
git status -s
 M directory2/entryway
D  file1
$
```

git mv

Rename a file in the working tree and index.

Git cannot represent renames in commits, but this is a convenient way to remove a file from the index and add the same file with a different name.

`--cached`: Alters the index but not the working tree.

`--force (-f)`: Moves the file in the working tree even if it will overwrite an existing file.

```
$ git mv file3 directory1/
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    file3 -> directory1/file3

$
```

What questions do you have?

`git restore`

Restore individual files to versions from a previous commit.

Note that this command does *not* change HEAD to the given commit. That means that the restored contents will appear as unstaged changes.

Note: This command is new and may not exist for you.

`--source` (`-s`): The commit to restore to.

`--patch` (`-p`): Interactively select hunks to restore.

```
$ git log --oneline file2
8320c08 Fix file2 to match file zoo
fe7b7f5 Add file2

$ git restore -s fe7b7f5 file2
$ git diff
diff --git a/file2 b/file2
index a5c1966..48d5349 100644
--- a/file2
+++ b/file2
@@ -1 +1 @@
-Hello, world
+hello wodrl
$
```

`git checkout <commit> <file...>`

Like `git restore`, but more widely available.

Note that the list of files after the commit are crucial. If you omit them, the command does something else (detailed in a later slide). This is why `git restore` was introduced as a separate command.

Unlike `git restore`, adds restored changes to the index.

`--patch` (`-p`): Interactively select hunks to restore.

```
$ git checkout fe7b7f5 file2
Updated 1 path from d5f1e5a
$ git diff --cached
diff --git a/file2 b/file2
index a5c1966..48d5349 100644
--- a/file2
+++ b/file2
@@ -1 +1 @@
-Hello, world
+hello wodrl
$
```

What questions do you have?

`.gitignore` file

Holds a list of file names to exclude from `git add .` and `git status`.

You generally don't want to commit things like compiled programs and editor swap files. The `.gitignore` list makes it harder to accidentally include such files in a commit.

Includes one glob pattern per line. Patterns that start with `/` match files relative to the repository root. Other patterns match files in any directory.

See `man gitignore`.

```
$ touch newfile
$ git status
?? newfile

$ cat >.gitignore
/newfile
<Ctrl-d>

$ git status -s  # newfile no longer shown!
?? .gitignore
$
```

What questions do you have?

Branching

Branches and the commit graph

A Git history is made up of an arbitrary number of commits, each of which has one or more parent commits. This forms a *directed acyclic graph* (*DAG*).

Branches are how we give names to the leaves of the graph (commits with no children).

A hash will always point to exactly the same object—there is no way it won't, since an object's hash is derived from its contents.

But branches (and, more generally, refs) can change what they point to.

```
* f4d4d12 (HEAD -> main) Add a longer file in directory2
* 095961d Add more files
*   878b7f0 Merge branch 'add-file3' into main
|\
| * cf70a44 (add-file3) Update file3
| * d8b477d Add file3
* | 48a918c Add missing-link
|/
* 8320c08 Fix file2 to match file zoo
* fe7b7f5 Add file2
* 31f7261 Add directory with "hello world" file from zoo
| * ff553ab (add-symlink) Add a symlink to file1
|/
* 46e2f3b Initial commit
```

What questions do you have?

What does "rewriting history" mean?

- Branches generally only move "forwards"
  - i.e. the old commit they pointed to is an ancestor of the new one
- A branch moving "backwards" or "sideways" is often called *rewriting history*.
  - i.e. the new commit is a parent or cousin of the old one
  - One or more commits are removed from the branch's history
  - New commits may be added that are fixed versions of the old ones
  - The new commits have no relation to the old ones from Git's point of view
- Key point: existing commits never change
  - A commit is identified by its hash
  - A commit's hash is inextricably tied to its contents
  - It's impossible to change a commit but keep the same hash

`git branch`

List, create, or delete branches.

With no arguments, tells you what branches exist in your local repository.

`--all` (`-a`): Lists remote (as well as local) branches.

With a branch name, creates that branch at the given point.

`--move` (`-m`): Renames a branch.

`--delete` (`-d`): Deletes a branch.

`--force` (`-f`): Overwrites

```
$ git branch
  add-file3
  add-symlink
* main

$ git branch foo
$ git branch
  add-file3
  add-symlink
* main
  foo

$ git branch -m foo bar
$
```

```
$ git branch
  add-file3
  add-symlink
* main
  bar

$ git branch -d bar
Deleted branch bar (was
f4d4d12).
$ git branch
  add-file3
  add-symlink
* main
$
```

`git checkout <branch>` (no files)

Switch to a different branch (i.e. change HEAD) and update files to match.

Fails if you've modified any files that would be changed by switching branches.

`-b`: Create a new branch with the given name that points to the same commit as your current HEAD. (Same as running `git branch` first.)

`--force (-f)`: Discard local changes instead of failing.

`--merge (-m)`: Merge local changes, like a cherry pick.

```
$ git branch
  add-file3
  add-symlink
* main
$ git show --oneline --no-patch HEAD
f4d4d12 (HEAD -> main) Add a longer file in directory2
$ ls
directory1  directory2  file1  file2  file3  missing-link
$ git checkout add-symlink
$ git branch
  add-file3
* add-symlink
  main
$ git show --oneline --no-patch HEAD
f4d4d12 (HEAD -> main) Add a longer file in directory2
$ ls
file1  file1-link
$
```

Checking out a commit directly

If you give `git checkout` a commit hash instead of a branch name, it will point HEAD directly at that commit; *you will no longer be on any branch*. This is known as a *detached HEAD* state.

- Useful for viewing a previous commit
- Never useful for making new commits!
  - Not part of any branch
  - Can only be found if you know their hash directly
  - Not permanent
    - Git can delete objects that are not reachable from a branch
    - `git gc` forces it to do this, but it also sometimes does it automatically

What questions do you have?

Operations on commits

`git commit`

Create a new commit and update the current branch to point to it.

The new commit's parent will be HEAD when `git commit` is run.

`-m`: Uses the given commit message instead of opening a text editor.

`--amend`: Replaces the current branch's tip with a new version of the same commit, with the same parents. Points the branch at the new version so the old one is no longer in its history.

```
$ git log --oneline
9360e39 (HEAD -> main) Add third line to file1
6d6a4d1 Add second line to file1
<...>
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   file1

$ git commit -m "Add fourth line to file1"
$ git log --oneline
git log --oneline
4d1e8fc (HEAD -> main) Add fourth line to file1
9360e39 Add third line to file1
<...>
$
```

How to write commit messages

Title with less than 65 chars in imperative mood, like

> Add lecture notes to main page

Summary containing not just the broad "what" but also the "why" -- think of it as a letter to your future self or another maintainer

git revert

Create a new commit that undoes the changes of a previous one.

`--no-edit`: Use the default commit message instead of opening an editor.

`--no-commit (-n)`: Undoes the changes in the working tree and index, but skips actually committing those changes. Useful if you want to make further edits.

```
$ git show --stat --oneline HEAD
513f37f (HEAD -> main) Add a longer file in directory2
 directory2/entryway | 32 ++++++++++++++++++++++++++++++++
 1 file changed, 32 insertions(+)

$ git revert --no-edit HEAD
[main 919aa08] Revert "Add a longer file in directory2"
 Date: Thu Oct 14 18:21:09 2021 -0400
 1 file changed, 32 deletions(-)
 delete mode 100644 directory2/entryway

$ git show --stat --oneline HEAD
919aa08 (HEAD -> main) Revert "Add a longer file in
directory2"
 directory2/entryway | 32 --------------------------------
 1 file changed, 32 deletions(-)
$
```

What questions do you have?

`git cherry-pick`

Apply a single commit to your current HEAD

Does not *change* that commit but instead re-applies a copy of it (new hash)

```
$ git cherry-pick some-branch
[main 39b9276] Some commit title
 Date: Tue Oct 5 15:10:17 2021 -0700
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 my-file
$
```

git rebase

Rewrite history

The **-i** or **--interactive** flag allows you to piecemeal build your history in your editor of choice

Especially useful if you want to edit your commits in the middle

```
       Assume the following history exists and the current
branch is "topic":

                     A---B---C topic
                    /
               D---E---F---G master

       From this point, the result of either of the following
commands:

           git rebase master
           git rebase master topic

       would be:

                             A'--B'--C' topic
                            /
               D---E---F---G master
```

What questions do you have?

Fixing Git mistakes

git reset

Reset the index to HEAD (or a given commit).

Remove changes from the staging area. Can optionally take a list of files to reset; if no files given, resets all files.

`--patch` (`-p`): Interactively select which changes within each file to unstage.

```
$ git status -s
 M directory2/entryway
M  file1
 M file3

$ git reset
Unstaged changes after reset:
M    directory2/entryway
M    file1
M    file3

$ git status -s
 M directory2/entryway
 M file1
 M file3
$
```

`git reset --keep`

Change where the current branch points without discarding uncommitted changes.

Like `git checkout`, but changes where the current branch points instead of switching to a new branch.

`--merge` (instead of `--keep`): Tries to merge uncommitted changes to a file with changes between the old and new branch tips instead of failing.

```
$ ls
directory1  directory2  file1  file2  file3  missing-link
$ git show --stat --oneline HEAD
513f37f (HEAD -> main) Add a longer file in directory2
 directory2/entryway | 32 ++++++++++++++++++++++++++++++++
 1 file changed, 32 insertions(+)
$ git status -s
 M file1
 M file3

$ git reset --keep HEAD^

$ ls
directory1  file1  file2  file3  missing-link
$ git status -s
 M file1
 M file3
$
```

git reset --hard

Reset the branch, working tree, and index to a given commit.

DANGEROUS! This command will discard uncommitted changes with no extra confirmation.

Does not remove untracked files, but discards any working tree or index changes to all tracked files.

```
$ ls
directory1  directory2  file1  file2  file3  missing-link
$ git show --stat --oneline HEAD
513f37f (HEAD -> main) Add a longer file in directory2
 directory2/entryway | 32 ++++++++++++++++++++++++++++++++
 1 file changed, 32 insertions(+)
$ git status -s
 M file1
 M file3

$ git reset --hard HEAD^

$ ls
directory1  file1  file2  file3  missing-link
$ git status -s  # Changes gone!
$
```

What questions do you have?

`git reflog`

Show all the places a ref (HEAD by default) has pointed.

Since branches and other refs change over time (with commits, amends, rebases, resets, etc), it can be useful to see where a ref pointed in the past.

You can `git reset` to somewhere in the reflog to undo an accidental commit, a botched rebase, or any other accidental operation concerning branches.

(Look for command-specific ways like a `--abort` flag first, though.)

```
willow% git reflog
39b9276 (HEAD -> main) HEAD@{0}: reset: moving to main
39b9276 (HEAD -> main) HEAD@{1}: checkout: moving from 39b9276f9eb16022bcab892702376b2759639114 to main
39b9276 (HEAD -> main) HEAD@{2}: checkout: moving from main to 39b9276f9eb16022bcab892702376b2759639114
39b9276 (HEAD -> main) HEAD@{3}: cherry-pick: wsjosdlfkjsdf
060aae9 (origin/main) HEAD@{4}: checkout: moving from 72950b0f342261f0c00feab3701e843503f4cf0a to main
72950b0 HEAD@{5}: commit: wsjosdlfkjsdf
3adbcb1 HEAD@{6}: checkout: moving from main to 3adbcb184865e03836cf2f759959afa1135824d6
060aae9 (origin/main) HEAD@{7}: reset: moving to origin/main
40e9818 HEAD@{8}: commit: tet
0d0cbd7 HEAD@{9}: commit: I'm a real program
d0133e4 HEAD@{10}: commit: TODO
060aae9 (origin/main) HEAD@{11}: rebase finished: returning to refs/heads/main
060aae9 (origin/main) HEAD@{12}: rebase: checkout origin/main
0b5f30d HEAD@{13}: commit: Update wording on /dev/ and citing sources
26ed856 HEAD@{14}: commit: Link to lectures 5 and 6 from homepage
53bca02 HEAD@{15}: commit: Add draft of lecture 6 notes
0324be2 HEAD@{16}: commit: Add draft of lecture 5 notes
6249be1 HEAD@{17}: rebase finished: returning to refs/heads/main
6249be1 HEAD@{18}: rebase: checkout origin/main
bef4f1e HEAD@{19}: commit: Add CC BY-SA license
f7f67ed HEAD@{20}: cherry-pick: Minor administrivia updates
0ef5a91 HEAD@{21}: rebase finished: returning to refs/heads/main
```

What questions do you have?

Odds and ends

The Git stash

Imagine you're in the middle of writing a feature when suddenly you get an urgent bug report! How can you get your working tree back to a clean state so you can start debugging?

1. `git reset --hard`
   - Bad idea: loses all your in-progress changes!
2. Create a feature branch, commit WIP changes, switch back to main branch
   - Three separate operations
   - Will need to amend commit later
3. **`git stash`** (see `man git-stash`)
   - Quick and lightweight way to save uncommitted changes for later
   - Easy to get back with `git stash pop`
   - Can get confusing if you have lots of stashed changes, though

git bisect

For those who would like to go spelunking to find when something broke.

- Provide with known good commit and bad commit (for your particular issue)
- Have a command to run to check if the current commit is good
- Binary search on your commit history
- Two modes: manual and automatic

```
$ git log --oneline --graph
* a2ca8cb (HEAD -> bisect-demo) Allow the user to pick a string
* 6c5ece9 Use a more descriptive function name
* b6aa4c1 Follow Google style guide
* 00a1ccf Fix capitalization
* 7cc1b18 Using constants is good style
* 7ea7ece New phrase
* 01f344f First commit of my sample C program
$ git bisect start HEAD 01f344f --
<...>
$
```

What questions do you have?

Using Git with friends

(or coworkers, if you don't have friends)

Distributed vs centralized version control

Git is a *distributed* version control system (DVCS):

- History is stored locally with every copy of a repository
- New commits and objects become part of the local history first
- Sharing commits and objects is an explicit operation
- No authoritative copy of a repo (except socially)

Compare to *centralized* VCSes like Subversion, CVS, and Perforce:

- A repository's state and history are stored on a single server
- Clients talk to that server to fetch files and make changes
- New commits are automatically visible to everyone

Git sharing subcommands

`git remote`: Tell Git what other copies of the repository you care about.
Set up or view *remotes*, aliases for repositories stored elsewhere that you can exchange objects with. Git can access a remote stored on the local filesystem, on an HTTP server, or on any system accessible via SSH.

`git fetch`: Copy branches from a remote to your local repository.
Branches from a remote are prefixed with the remote's name and a slash. For example, `origin/main`.

`git push`: Copy a branch from your local repository to a remote.
Modifies branches on the remote, so requires write access.

What questions do you have?

git remote

Add, remove, modify, or query remotes.

With no arguments, lists remotes. `-v` shows URLs.

`add`: Adds a remote with the given name and URL.

`remove`: Removes a remote.

`rename`: Changes a remote's name.

`set-url`: Changes a remote's url.

```
$ git remote add origin /comp/50ISDT/examples/git-zoo/
$ git remote
origin
$ git remote add tom /h/thebb01/git-zoo/
$ git remote -v
origin    /comp/50ISDT/examples/git-zoo/ (fetch)
origin    /comp/50ISDT/examples/git-zoo/ (push)
tom    /h/thebb01/git-zoo/ (fetch)
tom    /h/thebb01/git-zoo/ (push)
$ git remote rename tom max
$ git remote set-url max /h/mberns01/repos/zoo/
$ git remote -v
max    /h/mberns01/repos/zoo/ (fetch)
max    /h/mberns01/repos/zoo/ (push)
origin    /comp/50ISDT/examples/git-zoo/ (fetch)
origin    /comp/50ISDT/examples/git-zoo/ (push)
$ git remote remove max
$
```

`git fetch`

Copy objects and branches from a remote to local remote-tracking branches.

Updates *remote-tracking branches* for the given remote. Doesn't affect your local branches, index, or working tree.

Can take a list of one or more branch names to fetch; if none are given, fetches all branches.

`--all`: Fetches from all remotes at once.

```
$ git init  # New, empty repository
$ git remote add origin /comp/50ISDT/examples/git-zoo/
$ git fetch origin
remote: Enumerating objects: 36, done.
<...>
From /comp/50ISDT/examples/git-zoo
 * [new branch]       add-file3   -> origin/add-file3
 * [new branch]       add-symlink -> origin/add-symlink
 * [new branch]       main        -> origin/main
$ git branch main origin/main
Branch 'main' set up to track remote branch 'main' from
'origin'.
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
$ ls
directory1  directory2  file1  file2  file3  missing-link
$
```

git push

Copy one or more branches, and all their objects, to a remote.

Takes a remote name followed by one or more branch names. Creates or updates the named branches on the remote from matching local branches.

If local and remote branches have different names, specify them both separated by a colon.

Specifying a remote branch preceded by a colon deletes that branch on the remote.

```
$ mkdir ../tmp/ && cd ../tmp/ && git init && cd -
Initialized empty Git repository in /h/thebb01/example/tmp/.git/

$ git remote add example ../tmp/
$ git push example main
Enumerating objects: 33, done.
Counting objects: 100% (33/33), done.
Delta compression using up to 6 threads
Compressing objects: 100% (21/21), done.
Writing objects: 100% (33/33), 3.48 KiB | 209.00 KiB/s, done.
Total 33 (delta 9), reused 0 (delta 0)
To ../tmp
 * [new branch]       main -> main

$ cd ../tmp/
$ git branch -v
  main 513f37f Add a longer file in directory2
$
```

What questions do you have?

SSH and HTTP(S) remotes

```
laptop$ mkdir zoo && cd zoo && git init
Initialized empty Git repository in /home/thebb/zoo/.git/

laptop$ git remote add origin thebb01@homework.cs.tufts.edu:/comp/50ISDT/examples/git-zoo

laptop$ git fetch origin
remote: Enumerating objects: 36, done.
remote: Counting objects: 100% (36/36), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 36 (delta 10), reused 32 (delta 8)
Unpacking objects: 100% (36/36), 3.66 KiB | 85.00 KiB/s, done.
From homework.cs.tufts.edu:/comp/50ISDT/examples/git-zoo
 * [new branch]      add-file3   -> origin/add-file3
 * [new branch]      add-symlink -> origin/add-symlink
 * [new branch]      main        -> origin/main

laptop$
```

`git clone`

Shorthand for `git init` + `git remote add` + `git fetch` + `git checkout -b`

Takes a URL, which will be set as the `origin` remote, and an optional directory name to clone to.

```
$ git clone git@github.com:tekknolagi/isdt.git
Cloning into 'isdt'...
remote: Enumerating objects: 464, done.
remote: Counting objects: 100% (464/464), done.
remote: Compressing objects: 100% (284/284), done.
remote: Total 464 (delta 257), reused 334 (delta 141), pack-reused 0
Receiving objects: 100% (464/464), 271.34 KiB | 3.01 MiB/s, done.
Resolving deltas: 100% (257/257), done.
$
```

What questions do you have?

Integrating changes from remotes

Just like integrating changes from a local branch!

Three main strategies:

- **Fast-forward**: If no new local commits and no history-rewriting remote commits, add all the new remote commits to the local branch.
- **Merge**: If new local commits and new remote commits, use `git merge` to create a merge commit derived from both sets of changes.
- **Rebase**: If local commits have not been shared anywhere, can alternatively rebase them on top of new remote commits.

git pull

Fetches changes from the remote branch that your checked-out branch tracks (see git branch -vv), then merges or rebases those changes into the local branch.

Warning: `git pull` relies on a lot of implicit configuration (like remote branch mappings) and can have different results (fast-forward vs merge commit) depending on the state of the remote. It's often better to explicitly run `git fetch` followed by your desired integration command.

`--ff-only`: Forces a fast-forward merge. Fails if not possible.

`--rebase (-r)`: Rebases local changes on top of new remote state instead of merging.

Conflicts

What happens if a merge, rebase, or cherry-pick involves two commits that changed a single base version of a file in two different ways?

- All these operations call `git merge-file`
- Does its best to reconcile the changes on its own by splitting into hunks
- If two hunks conflict, enters *conflict resolution*:
  - Conflicting hunks are marked in the working tree
  - Special `<<<<<<<` lines indicate conflicts
  - Up to you to resolve manually
  - Use `--abort` if there are too many conflicts
- GUI tools for merging files: WinMerge, Meld

What questions do you have?

Common open-source development workflow

- One repository copy designated as source of truth
  - e.g. torvalds/linux
- *Maintainers* are allowed to push to this copy
- Anyone else wanting to contribute communicates their changes to maintainers out-of-band
  - For Linux, patches on a mailing list
  - For many other project, GitHub *pull requests*

What questions do you have?

Maintaining a codebase

Limitations of collaboration via `git push`

- Requires you to fully trust every contributor.
- Easy for bad code to be committed.
- No built-in way to comment on a commit.
- No way to enforce code review policies.

Solution: merge/pull requests and patches

Out-of-band way for a contributor to propose a change to a repository.

- **Merge request** (a.k.a. pull request): Instead of a contributor pushing their feature branch upstream, they push it to their own *fork* and ask the upstream maintainer to pull it from there.
- **Patch**: No pushes or pulls happen at all. The contributor formats each commit they've made as a textual *patch* using `git show` or `git format-patch`, sends it to the maintainer (e.g. in an email), and the maintainer *applies* that patch using `git am` or `git apply`.

Either way, maintainers can comment on the change via the same communication channel it was sent over and only accept it if they approve of it.

What questions do you have?

Revising proposed changes

Two schools of thought:

1. Add new commits containing fixes on top of the original ones
    - Pros: easy to see what changed in each revision, commit message can explain what got fixed
    - Cons: no way to "fix" a bad commit message, makes `git blame` harder
2. Rewrite history to produce new, fixed versions of each commit
    - Pros: results in clean, readable commits at the end of the day
    - Cons: requires maintainers to deal with rewritten history prior to acceptance

(See https://github.com/tekknolagi/isdt/pull/15#issuecomment-920592835.)

What questions do you have?

Bare vs full repositories

Beyond `git`

Hosting services ("forges")

These web-based services provide:

- Repository hosting (e.g. a remote everyone can use)
- Code browsing
- Merge requests / code review
- Issue tracking / social features
- Access control
- Release management

Centrally hosted (free or paid):

- **GitHub**
- **GitLab**
- **Savannah**
- **sourcehut**

Self hosted:

- **Gitea**
- **Gogs**
- **gitweb** (code browsing only)
- **cgit** (code browsing only)

demo time!

A history of version control

- SCCS (1977)
- RCS (1982)
  - Single files
- *CVS* (1986)
- *Perforce* (1995)
  - Proprietary
  - Centralized
  - Good at tracking binary files
  - Used by many game developers
- *Subversion* (2000)
- BitKeeper (2000)
  - Proprietary
  - Used by Linux for a while
- Darcs (2003)
- Git (2005)
  - Built by Linus for Linux
- Mercurial (2005)
- Fossil (2006)
- Pijul (2020)
  - Based on a sound theory of patches

(Systems in *italics* rely on a centralized server.)

Case study: Git vs Mercurial

- Mercurial does not have an equivalent to Git branches
  - Each commit is individually marked as shown or hidden in the DAG
  - No need for branches to dictate which commits "matter"
  - Lines of development can be tracked using *bookmarks*, which serve a similar purpose
- Rebasing and rewriting history are more acceptable
  - In part due to the lack of branches, Mercurial's tooling for handling rewritten history is better
  - Very easy to move bookmarks around
- Mercurial has no staging area
  - A file's contents are not remembered at all until it's committed
  - Simpler to reason about, but results in more amends
  - Better UI for selecting changes to go into a specific commit (`hg commit -i`)

Source control at Big Tech

Challenges:

- Huge numbers of commits and committers: log/blame still has to be fast
- Tightly integrated code: need to change multiple codebases atomically
- Total size of code is very big: cannot all fit on a single workstation

Solutions:

- Extend existing DVCS to have centralized storage and tight tooling integration
  - Facebook: EdenSCM
  - Microsoft: VFS for Git
- Build custom solution on top of existing Big Data infrastructure
  - Google: Piper

Git GUI wrappers

- **gitk**
  - Open-source, distributed alongside Git
- **GitKraken**, **SourceTree**
  - Closed-source, free GUIs
- **GitHub Desktop**
  - Closed-source GUI
  - Integrates tightly with GitHub remotes
  - Supports GitHub Pull Requests, Issues

Using Git from your IDE/editor

Some editors allow you to see and commit changes inside their environment:

- **VS Code**: built-in (see demo)
- **Sublime Text**: built-in (Merge)
- **Emacs**: Magit
- **Vim**: fugitive

These integrations are usually not nearly as fully-featured as the CLI or a dedicated GUI, however.

Separate code review tools

- Gerrit
- Phabricator

What questions do you have?