Version Control with Git: Porcelain Commands

From the User's Viewpoint

This is usually from the perspective of a software developer/manager, and in some cases, the end user trying to use the software.

Three things are under Git's control:

0. Keep track of the working files (ordinary files), your source code, the "current state" of the repository.
1. Object database recording history of project development, where the project is modeled with a tree of files.
2. **Index**, recording your plans for the *future*.

Getting Started

1. Create one remotely on a server (like GitHub) and clone it

```
git clone URL
```

2. Create one locally and then link it to a remote):

```
git init
git remote add NAME URL
```

Cloning copies a repository AND creates corresponding working files.

Git sets up a special `.git` subdirectory, which is what makes the project directory a "repository".

NOTE: We don't necessarily have a "boss and servant" relationship between upstream and downstream repositories. Often times, a clone can become more active/popular than the original, in which case, the latter will start to sync with the former instead of vice versa.

Exploring the Log

Display information about every commit leading up to the current version, in reverse order by default:

```
git log
```

We note that each commit has a unique ID, a long string after the word "commit", which is the **checksum** of the commit contents. A checksum is like a fixed bit integer that is a function of the bytes of the content it is encoding. This function must not produce **collisions**, where different contents produce the same checksum.

Formatting the Log

```
git log --pretty=fuller
```

The fuller format shows that every commit has an **author** AND a **committer**, each with their own dates. Git distinguishes between these two contributors. This separation is routine for many larger projects. The author would be the person that writes the code, and the committer would be the overseer that reviews the and confirms the changes. These fields *establish responsibility* for changes, a major reason for using version control in the first place.

An interesting thing you may observe about repositories that have been developed for a long time is that you may notice commits with timestamps dating back to before Git was even around. This is because projects may have migrated from other version control systems, like RVS and CVS, and in copying over the history, the date data are all preserved.

Other examples:

```
git log --stat
git log --oneline
git log --pretty=format:"%h - %an, %ar: %s"
git log --oneline --decorate --graph --all

# Look for differences that change the occurrences of specified string
git log -S<string>
```

Narrowing the Log

You can use the special `..` syntax to specify a commit range. The following displays the history between commits with ref `A` (exclusive) and `B` (inclusive), so like `(A, B]`. This is like "show me everything that led up to B, but exclude everything that led up to A":

```
git log A..B
```

Getting the *entire* history *up to* commit with ID `B`:

```
git log B
```

You can also use **"version arithmetic"** to get references to commits based on aliases, like branch names, tag names, `HEAD`, etc. More on commit notation in Git Internals. The official documentation is also nice: https://git-scm.com/docs/gitrevisions.

> The special pointer `HEAD` by references the *current version* of the repository. This is kind of like a "you are here" pointer. You can move it around such as checking out to other branches, a specific commit, etc. in which case `HEAD` moves to the corresponding commit and Git changes your project directory to match the snapshot of that commit.

You can use the `^` syntax to specify the parent of a reference, so `HEAD^` means the commit just before `HEAD`, `HEAD^^` means the grandparent commit, etc.

Example about showing the most recent commit:

```
git log HEAD^..HEAD
git log HEAD^!  # shorthand
```

Commit Messages

Commit messages are important because in essence, they help "market" your changes. They tell readers of the repository why certain commits were made and whether it was a "good" commit by explaining the *motivation* behind the changes. "Why are you making this change? Why shouldn't I just revert it?"

The rationale behind commit messages are similar to why you should comment your code. Oh yeah have I mentioned that:

> **COMMENTS SAVE LIVES. ALWAYS COMMENT YOUR CODE. PLEASE.**

There is overlap between comments in the source code and commit messages, but the primary distinction is the *audience*. Commit messages are more historically oriented, what you would tell the "software historian," people interested in the development of the repository as a whole. Comments in the source code are for the "current developer," people interested in having to study or change your code.

There are many style guidelines for commit messages out there, but here's Eggert's (which looks pretty standard in my experience).

Example commit message from the MIT repository shown in lecture:

```
Fix issues found by ASAN and Coverity

* tests/test_driver.pl: Preserve the LSAN_OPTIONS variable.
* tests/scripts/targets/ONESHELL: Don't set a local variable.
* tests/scripts/functions/let: Test empty let variable.
```

The first line should be at most 50 characters, and this acts as the "subject line" for the commit, like the elevator pitch. This should give any readers the *gist* of the commit.

The second line should be empty, separating the subject line from the body.

The remaining lines should be at most 50 lines, each at most 72 characters per line. Here you describe the finer details of the commit. You can use paragraphs, `*`-bulleted lists, etc.

Working Files and Index

Technically a plumbing command, this is Git's version of `ls`, where it displays the current *working files*:

```
git ls-files
```

This helps us distinguish between general files in the directory and files that currently *matter* with respect to the repository. These are the files that are currently being **tracked** by Git.

```
rm $(git ls-files)  # lol!
```

Oh yeah, finding content within files is such a common pattern that `grep` is built into Git:

```
git grep waitpid
```

The **index** is what you have ready for the next commit, but have not committed yet.

Commits and Staging

Commits are like checkpoints for your code, snapshots that are saved in the repository. Commits *append the index to the history*.

You know how it goes:

```
git add PATHSPEC
git commit -m MESSAGE
git push
```

There is an intermediate phase between modified/unmodified files and commit called the **staging area**. You can add files to this phase with `git add`

Unstaging a file:

```
git restore ---staged FILE
```

Syncing your local repository with the remote one:

```
git fetch
git pull  # effectively fetching + merging
```

Viewing Status

Checking the status of your repository:

```
git status
```

Each file can be in the following states:

- Staged
- Not staged but modified
- Untracked

In general, files go through these states:

Viewing Differences

`git diff` is similar to the GNU `diff` command, and like a more detailed version of `git status`.

> **HISTORICALLY:** The algorithm is very complex and was developed by a professor at the University of Arizona who went on to work on the Human Genome Project.

Viewing the difference between the *index* and the *working files*: `Δ(index vs working files)`:

```
git diff
```

This views the difference between the latest commit and the index: `Δ(latest commit vs index)`:

```
git diff --cached
git diff --staged  # equivalent
```

And this is `Δ(last commit vs working files)`

```
git diff HEAD
```

Compare the grandparent commit to the latest commit:

```
git diff HEAD^^..HEAD
```

More examples of viewing the difference between two commits:

```
# Typically with SHAs of the specific commits you want
git diff REF..REF

# But you can also abbreviate the hashes:
```

```
git diff 5c6cb30..53bf6bd
git diff 5c6c..54bf

# But this has a limit. This fails:
git diff 5c6..53b

# As usual you can use the HEAD ref to reference commits relative to
# the last commit:
git diff HEAD~..HEAD
git diff HEAD~4..HEAD
git diff HEAD^..HEAD
```

Making Changes

Typical workflow:

1. Edit the working files.
2. Run `git add FILES...` to add the specified file contents to the index (the **staging area**, the **cache**). You can keep editing files and add any new changes to the staging area with the same command.
3. Run one of the `git diff` commands to verify that the changes are what you want.
4. Run `git commit`, which takes your index, makes a new commit, and puts it into the object database with the auto-generated checksum. In effect, it changes the commit `HEAD` references.

You're probably familiar with `git commit -m MESSAGE` that every Git crash course teaches you. This is useful for one-liners, but the default `git commit` actually drops you into your configured editor and allows you to write longer commit messages with the subject line and body format detailed above.

There is also `git commit -m MESSAGE FILE`, where `FILE` contains the extended message. This is useful for automating messages in scripting. Example:

```
git commit -m 'Fix issues from previous patch' README.git
```

Removing all **untracked files**:

```
git clean
```

This is useful for removing files created as part of some build process. If you're not sure, you can run a "what if" with the `-n` option:

```
git clean -n
```

> The `--dry-run`/`-n` switch is common to a lot of Git (and Unix in general) commands. It's a good way to "preview" the effects of a potentially destructive command instead of running it blindly right away.

There's also the `-x` option which cleans files that will even be ignored:

```
git clean -nx  # you best see what that would do first lol
git clean -x
```

Configuring Git

The .gitignore File

A special file inside the repository containing file patterns that Git should not track. The file pattern syntax is similar to the familiar **globbing pattern** as the shell.

.gitignore is like a configuration file that instructs how users run Git. It's under Git's control i.e. it'll show up in `git ls-files`.

What files should be ignored?

Files that we do not want to put under version control. Obvious candidates include:

- Temporary files, `\#*`
- Machine-dependent code, `*.o`
- Imported files (from other packages)
- Authentication information (passwords/keys/etc.)
- Hashes of passwords? If it's intended for authentication, this would be just as bad as raw passwords, so ignore them too. Hashes enable **rainbow attacks** on the passwords where attackers try to crack the checksum algorithm.

The .git/config File

You can view the current configuration of the Git program with:

```
git config -l
```

This outputs the information stored in the editable `.git/config` file, which is specific to the current repository. Cloning a repository also copies the configuration file.

CAUTION: One notable problem (which is standard across any software) is that if there is a syntax error in the configuration file, Git stops working altogether.

`.git/config` is NOT under version control because it determines how Git itself functions and because it would introduce the problem of recursion. `.gitignore` IS under version control because it's like a message from the developer and contains information about how to manage the project actually being version controlled. You also don't need to worry about what's in `.gitignore` to use Git itself.

The ~/.gitconfig File

After resolving the configuration in the current repository, Git then falls back to this configuration file. Contains *global* configuration information for Git, like username and email.

You can edit this file manually with your editor of choice, but you can also use `git config` to write directly from the command line:

```
git config --global KEY VALUE
```

*Setting up Git on New Machines:**

```
git config --global user.name "Vincent Lin"
git config --global user.email vinlin24@outlook.com
```

The user name is not actually that important. It's mostly used for identifying contributors at a glance with things like `git log` I assume. The email however is *critical* because remote services like GitHub use that to identify the account of the contributor.

Other cool things you can specify in this file:

```
[core]
    # If you don't like being dropped into Vim by default,
    # This sets it to VS Code (you can also use Emacs, etc.)
    editor = code

[alias]
    # Abbreviations
    s = status
    co = checkout
    cm = commit -m
    # etc.
    # I have much more ehe
```

Show a Commit

`git show` is a generic command that shows a commit "object". Commit objects live in the database and are really just the recorded changes from the previous commit along with some metadata.

The ubiquitous `--pretty` option can be used here too for more verbose output:

```
git show --pretty=fuller
```

Working with Remotes

A **remote**, named `origin` by convention, is the **upstream** repository from which the local repository was cloned or set to track.

> The concept of **upstream/downstream** comes from the fact that clones may be sourced from repositories that are themselves sourced from another branch, forming a chain of origin - a "stream".

Fetching

This consults the remote server for upstream changes and syncs the clone's "opinion" of what upstream looks like:

```
git fetch
```

If it outputs nothing, it means the local clone is up-to-date with the upstream remote. `fetch` does not change the working files nor does it alter any branches, only your *local* copies of the *remote* branches, like `origin/main` instead of `main`.

`fetch` is incremental, only fetching the changes since the last call to fetch.

Pulling

`git pull` is roughly equivalent to a `git fetch` followed by `git merge` of those upstream changes into the current branch. This command is actually rarely ideal on large development projects because more often than not it may not be what you want to do. When it works, `pull` DOES change the working files.

A good habit to get into is to first download and preview the upstream changes with `git fetch`, and then when you're ready to integrate those changes, run `git pull` or merge them manually.

Fetching vs. Pulling

The commit objects are downloaded from upstream and made available on your local repository, you just don't immediately notice them because they update your copies of the **remote** branches; `git fetch` is a benign command because it leaves your **local** branches untouched. Suppose you had a simple history on your local machine. If you have a remote repository set up, with your `main` set to **track** the remote's version aka `origin/main`, your local repository would look like:

```
A <- B <- C (main, origin/main)
```

But maybe in reality the upstream repository (e.g. on GitHub) has updated since then:

```
A <- B <- C <- D <- E (origin/main)
```

Git can't know this *until* you run `fetch`, because that's when it communicates with the remote server. When you run `git fetch origin main`, your local repository does two things:

1. Download the new commit objects, `D` and `E`.
2. Update your branch tip representing the remote branch i.e. `origin/main`.

```
        (main)
          v
A <- B <- C <- D <- E (origin/main)
```

Notice your `main` didn't move - it doesn't even know what happened. You as the developer can see what was really brought in by specifying `origin/main` at the command line, like with:

```
git log main..origin/main
```

When you're ready to update your **local** `main` with these new changes, *then* you can run something like:

```
git pull origin main
```

But as Victor pointed out in @659_f1, `git pull` really is just `git fetch` followed by `git merge`, so since you already did the `fetch` part, you can also just complete the merge:

```
git merge origin/main
```

Now your local repository is updated to match the state of your upstream:

```
A <- B <- C <- D <- E (main, origin/main)
```

Recovering from Mistakes

Before the Commit

The working file is wrong, but the most recent version saved in the repository is safe. You haven't committed your erroneous changes yet, so simply:

1. Edit the file `F`.
2. Update the index: `git add F`.

After the Commit

The repository now has a bad version of the code in its history.

Option A: Commit a new, fixed version. The bad commit would still be recorded in the the old and fixed commit. This would be an honest representation of the development history, but often times this is not wanted.

```

```

```
(old)-->(broken)-->(fixed)
                      ^
                    HEAD
```

> An iron-clad rule in Git is that *you cannot change history*. This is because every commit is uniquely identified by the SHA-1 ID.

Option B: However, you can cheat this rule with the `git commit --amend` approach, which creates a new child from the parent commit and moves `HEAD` to it:

```
(old)-->(broken) # effectively discarded
  |
  +---->(fixed)
          ^
        HEAD
```

This is very risky as an upstream repository. If someone happens to `fetch` when `HEAD` still points to the broken commit, then when they `fetch` with the new altered tree, Git and the users will get confused.

Option C: You can change the state of the repository back to another version.

Reverting to the previous commit:

```
git reset HEAD^
```

This would fail if there are changes in the working files, in which case, you can throw the changes away and revert anyway with:

```
git reset --hard HEAD^
```

Branching

A single commit can have multiple children.

A **branch** in Git is like a lightweight, movable *name* for a commit that is the at the tip of a line of maintenance.

> At the end of the day, Git is all about pointers! 😄

One branch, the "main"/"master" branch is typically reserved for *mainline development*. There may be other branches for things like *maintenance development*, *old releases*, *hot fixes*, etc.

Patching Across Branches

Suppose a security hole was discovered in an old commit, which multiple branches share as an ancestor. You can fix the bug on the mainline branch, but that doesn't solve it for other branches.

The solution is to **cherry-pick fixes**. You manually apply the same Δ to all versions that have the same bug.

Suppose there's an alternate branch named `maint`.

```
# Your familiar sequence
git add F
git commit -m "Make an emergency fix"

# Prepare the patch to apply to other branches
git diff HEAD^! > t.diff

# t.diff is a working file, preserved across checkout
git checkout maint

# Apply patch to this branch's working files
patch < t.diff
git add F
git commit -m "Make an emergency fix"
```

The `patch` command is actually external to `diff`. It reads the output of the diff file and modifies the old file so that it looks like the new file:

```
diff -u A B > AvsB.diff
patch < AvsB.diff
```

This modifies `A` to look like `B`.

Attempting to apply a patch to a since edited version of a file may fail to work. It may still work if the changes to the original files does not *collide* with what the patch is attempting to change.

`diff` operates on **hunks**, batches of lines that represent a change. Patching goes through each hunk and applies the change. If the hunks do not match, then it will reject the change into an `rej` file, prompting you to fix it by hand.

NOTE: The output of `diff` is NOT deterministic. There is no requirement of the algorithm to modify a file in a specific way as long as the final copy is correct.

Patching and Cherry-Picking

Create a patch file(s) for a commit

```
git format-patch [-o DIR] REF
```

```
git am FILES...
git apply FILES...
git cherry-pick REF
```

Manipulating Branches

Listing all branches, optionally verbosely:

```
git branch -a [-v]
```

Switching between branches:

```
git checkout REF
```

Creating a branch off a commit, defaulting to `HEAD`, and checking out to it:

```
git checkout -b NAME [REF=HEAD]
```

EXAMPLE: creating and checking out to a new branch named `newbr` off of the grandparent of the current `HEAD`:

```
git checkout -b newbr HEAD^^
```

Branch names must be unique. Git won't let you create or rename a branch to an existing name. As you know from HW6, this is because information about branches are stored as physical files in the file system under `.git`.

Because branches are just names, *deleting* a branch does not modify any commits, just a reference that used to point to one.

```
git branch [--delete | -d] NAME
```

You can try to do something weird like:

```
git branch -d master
```

The objects would still be there, but Git will lose track of where they are, so Git would warns you. You can *forcefully* delete a branch despite Git's warnings with:

```
git branch -D NAME
```

Renaming a specific branch, defaulting to the current branch:

```
git branch -m NEW_NAME [OLD_NAME=HEAD]
```

Detached HEAD State

You can checkout to an arbitrary commit by ID/tag name:

```
git checkout REF
```

But this puts you in **detached HEAD state**, which is when `HEAD` is not pointing to any branch tip. Git warns you that you can look around but not make further changes. You cannot commit in this state because Git does not know how.

However, if you want to make changes from this version of the codebase, you can checkout to a new branch off this commit as you normally would:

```
git checkout -b mybranch
```

(1) Merging

Merging is one way to bring together multiple lines of development. You integrate changes **from** one branch **to** another branch. More technically, it is when you create a commit from two or more parents commits, which may or may not be named branch tips.

Mechanism

Suppose:

```
                    this is the merge commit
                            v
()<--(A)<--()<--()<--(Y)<--(Merged)<-- ...
      |                       |
      +------()<----(X)<------+
```

Git finds the common ancestor `A`, of the parent commits `X` and `Y`. Then, it runs computation on all 3. It is as if Git runs:

```
diff3 X A Y > combined.diff  # "3-way diff"
```

This file describes changes to change the common ancestor `A` to *either* `X` or `Y`. Git then applies those changes and creates a *new* commit instance for it, the **merge commit**.

The command to merge a branch named `BRANCH_NAME` into the current branch:

```
git merge BRANCH_NAME
```

What this does is:

1. Compute 3-way merges.
2. Replace working files accordingly.

Merge Conflicts

More often than not, the changes *collide*, resulting in a **merge conflict**. If there's a collision, Git modifies the affected files with a "replica" of the collision with the special notation. Without a GUI, conflicts in their raw form actually look like:

```
<<<<<<<
A (Current Change)
=======
B (Incoming Change)
>>>>>>>
```

Git actually modifies the content of the conflicting file with this pattern, conflicting text separated by special barriers. The one with `<` brackets shows the **CURRENT** content, and the one with `>` brackets shows the **INCOMING** content. To resolve the conflict, you need to edit this block to only include one version of this content ("accept current change" or "accept incoming change"). You can also accept both changes. You could also leave the file in this conflicted state with the barriers, but that's stupid practice because if you do this on a source file, it will almost definitely be a syntax error.

Resolving conflicts: usually you would edit the conflicting files in your editor of choice and regularly check for further instructions with `git status`. You can just conclude the merge with:

```
git merge --continue  # but not recommended
```

(2) Rebasing

Alternatively, one can **rebase** a commit onto another branch. This takes away the problem where reviewers have to worry about common ancestry and a bunch of diffs. They only need to examine a linear history.

```
( )<--(common)<--( )<--( )<--(main)
          |
          +--( )<--(my)
          Δ1     Δ2
```

Often times, the main branch undergoes complex changes since the branch last diverged, so it may be easier to just move the commits of the merging branch right to the tip of the branch it's merging into.

```
( )<--(common)<--( )<--( )<--(main)
                             |
                             +--( )<--(my)
                             Δ'1    Δ'2
```

Git does this by taking the deltas Δ1 and Δ2 and then combining them with changes at `main` since `common` to compute new deltas Δ'1 and Δ'2. From the user's perspective, it's like we plucked off the `my` branch and attached it to the tip of `main`.

Merging vs. Rebasing Pros/Cons

Merging	**Rebasing**
✚ Only one commit is created per merge.	━ A new commit is made for every commit you rebase.
✚ Does not change existing commit history, so you're less likely to screw over others working on the same branch.	━ Changes existing commit history. If you misuse this command, you could mess up an important branch like `main` for everyone else. See the Golden Rule of Rebasing.
✚ Your steps can be fully retraced because you know when each merge was performed.	━ It is difficult to see when a rebase actually occurred.
━ If you have to merge often, these merge commits may pollute your history and make it harder to understand.	✚ No unnecessary merge commits polluting your history, making it easier to understand.
━ Your history will still have interweaving branches that may make navigation (`git log`, `git bisect`, etc.) harder.	✚ Keeps your history as linear as possible, making it easier to navigate with such commands.

Another downside is that you may repeatedly rebase if progress resumes on the `main` branch while review was still pending.

IMPORTANT: The fact that rebasing manually linearizes history is a point of philosophical concern. Some may see this as an upside because it makes it easier to review, but others may see rewriting history as a bad thing.

> **ASIDE:** Often times, project managers use Git history to see which programmers are better and who deserve bonuses. This is not always the best approach. The history does not tell you the full story. There is a very tenuous relation between number of commits and the value of a programmer to a project. A glaring example at this time of writing is the mass layoffs at Twitter where *genius* Elon Musk thought to fire people who have written the least lines of code.

> The *essence* of Git and any version control system is that you are editing *changes* to source code, not the source code itself. Understanding this is what sets a **software developer** apart from an ordinary "programmer".

> If you find yourself in an environment where people are lying on purpose about commits in order to improve their job prospects or something, you're in the wrong company. **- Dr. Eggert**

(3) Stashing

Rebasing is less "formal" than merging. Changes are looser, but are recorded as a sequence of commit nonetheless. Stashing is even less formal than rebasing. Changes are only floating around in your repository, waiting to be reapplied.

The scenario looks something like:

1. You're working on the next change in your branch.
2. You want to switch to some other branch NOW. You *could* commit what you have right now, but that is bad practice because you're essentially committing junk. "You want your repository to be in good shape at all times."
3. Instead, you could save your changes in an external file:

   ```
   git diff > mywork.diff
   git checkout -f  # Discard working files
   ```

4. Checkout to the other branch and do work on it

```
git checkout main
```

5. Checkout back your original branch and patch it.

```
patch < mywork.diff -r1
```

Git actually provides a way to do this within Git itself, using the `stash` command. At step 3, you would do something like:

```
git stash push
```

This saves the state of your working files in some part of the index. When you want to retrieve this state, you can get it from the stash stack with:

```
git stash apply
```

Bisecting

Suppose you have a linear piece of history where somewhere between a stable version and the most recent commit, something went wrong. You can think of this problem of finding the first faulty commit as partitioning the timeline into OK and NG ("not good") sections, hence *bisecting*.

The timeline is "sorted" in that if you think of OK=0 and NG=1, the history will always be such that all NGs follow OKs.

```
                    |
(v4.3)<--( )<--( )<---( )<--( )<--(main)
OK        OK    OK  |  NG    NG     NG
                    |
```

This then becomes a classic *binary search* problem, where we can identify the first NG commit in O(logN) time.

Starting a bisect in Git:

```
#                 NG   OK
git bisect start HEAD v4.3
```

Then we tell Git to run your check script on each commit and use the exit status to determine if the commit is OK or NG:

```
#              vvvvvvvvvv any shell command
git bisect run make check
```

In this case, we use a Makefile with a `check` target that defines some test cases for the program

Of course, this also introduces the problem that if your test cases are buggy, then you may get false alarms. If you know ahead of time that a commit, say `v3`, will produce unreliable test results, you can skip it with:

```
git bisect skip v3
```

Collaborative Best Practices

Branches:

- `master` or `main`: stable branch
- `develop`: for development
- Each team member may create their branches for individual features/bugs

Protect the `master` branch. DON'T force-push; it could destroy the commit history. Don't be that guy.

On GitHub, use **pull requests** to merge changes into other branches. Pull requests may undergo a **review**. You can also use **issues** to assign bugs or features to team members.

Avoid merging temporary files or code meant for debugging (like dummy data).