Git Training

Brian Pittman

brian.pittman.3.ctr@us.af.mil

12/7/2015

Obligatory XKCD

Building Blocks of Git

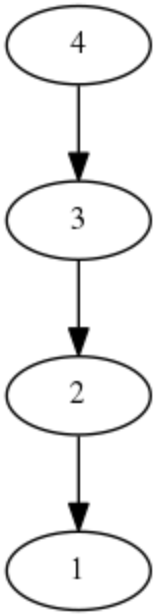
- Blobs
  - Each file in the repo is represented by a blob
- Trees
  - Each directory is represented by a tree
  - Trees can contain other trees (subdirectories) and blobs (files in the directory)
- Commits
  - A snapshot of your working tree at a point in time, plus some metadata (author, commit date, parent commits, SHA1 hash, etc)
- Refs
  - Bookmarks to a certain node in the repository

Building Blocks of Git: SHAs

- A commit is uniquely identified by a 160-bit hex value (the 'SHA'). This is computed from the tree, plus the following pieces of information:
  - the SHA of the parent commit(s) -- every commit except the very first one in the repo has at least one parent commit that the change is based upon
  - the commit message
  - the author name/email/timestamp
  - the committer name/email/timestamp
- The other git object types are referenced by SHAs as well, but these are generally not relevant
- The SHA is a globally unique identifier across all git repos
- Existing commits cannot be modified. Any time it seems like they are, a new commit is being created behind the scenes with a different SHA

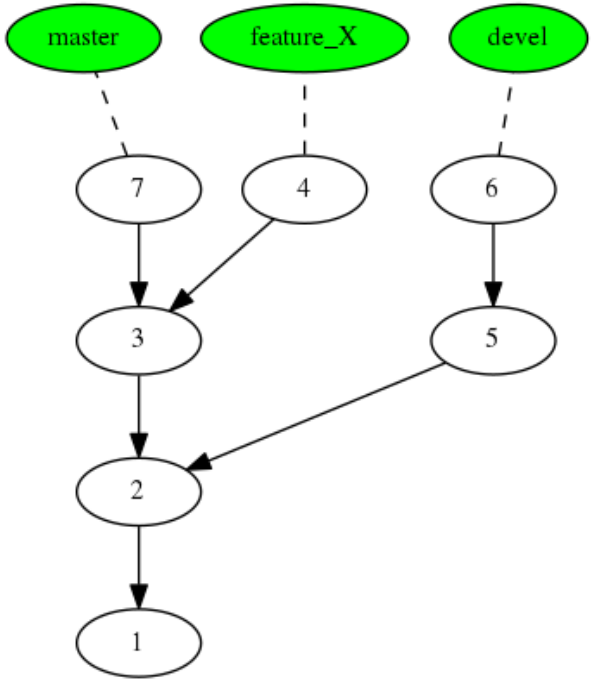
Building Blocks of Git: repositories

- A directed acyclic graph of commits
- Normally represented visually with time moving upward (oldest commits at the bottom)



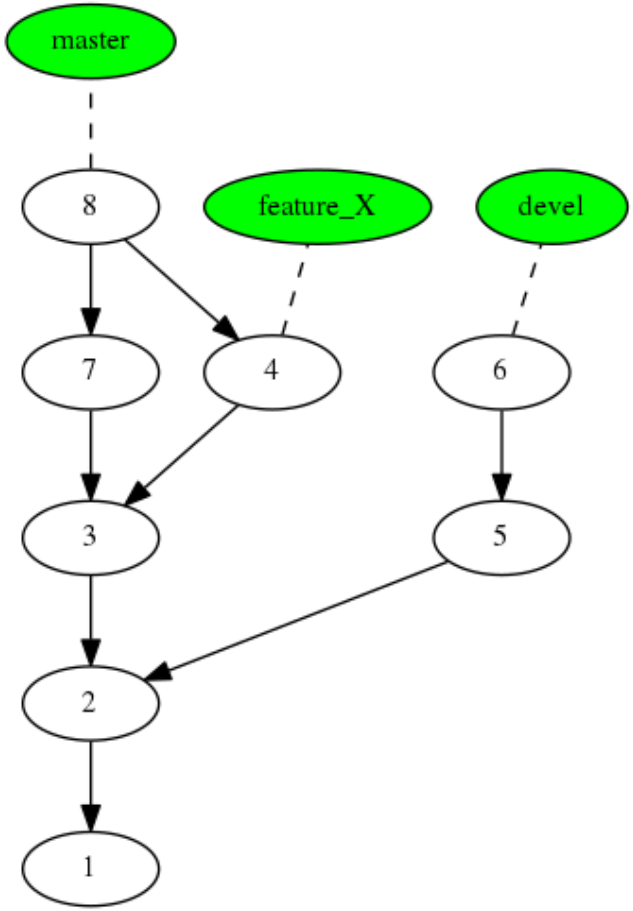
Building Blocks of Git: Branches

- Branches are stored as a reference to the commit at the tip of the branch
- The master branch is created by default



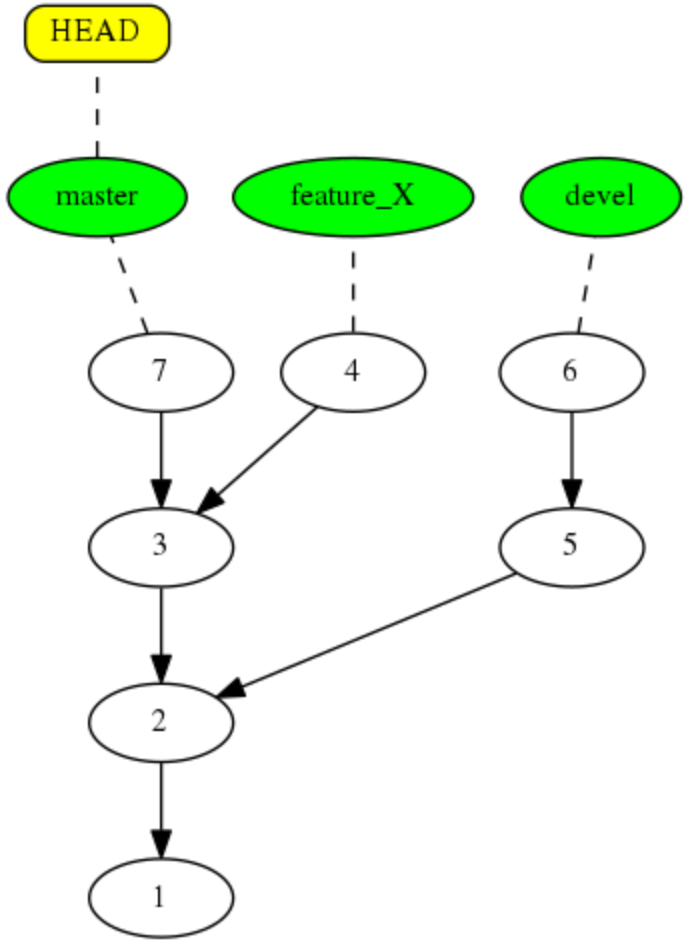
Building Blocks of Git: Merging

- Commit 8 is a merge commit because it has 2 parents



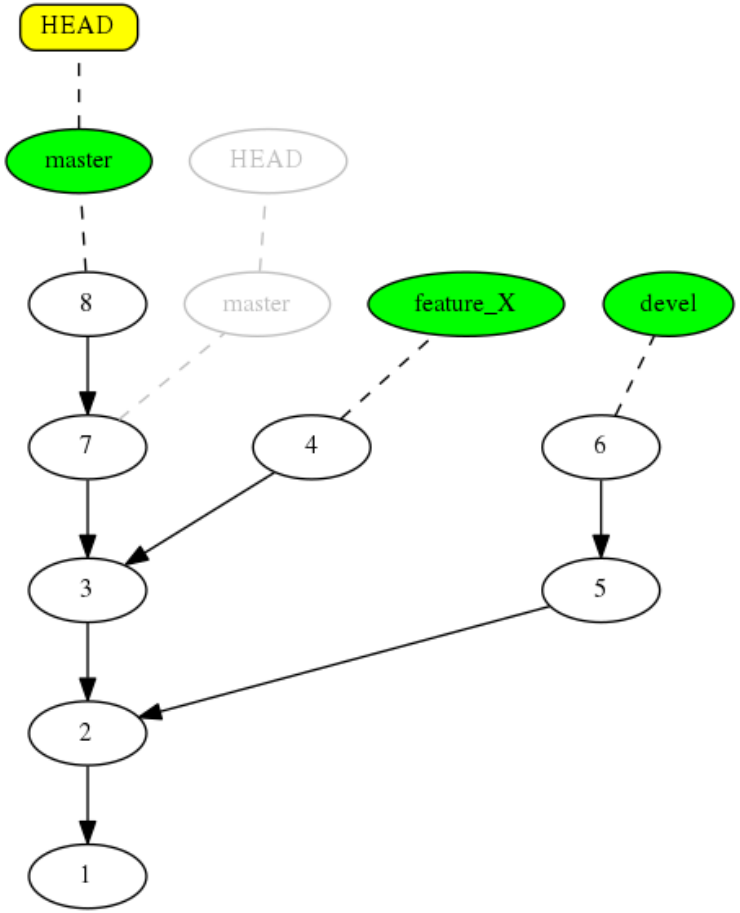
Building Blocks of Git: HEAD

- HEAD is a special reference that points to the currently checked out branch



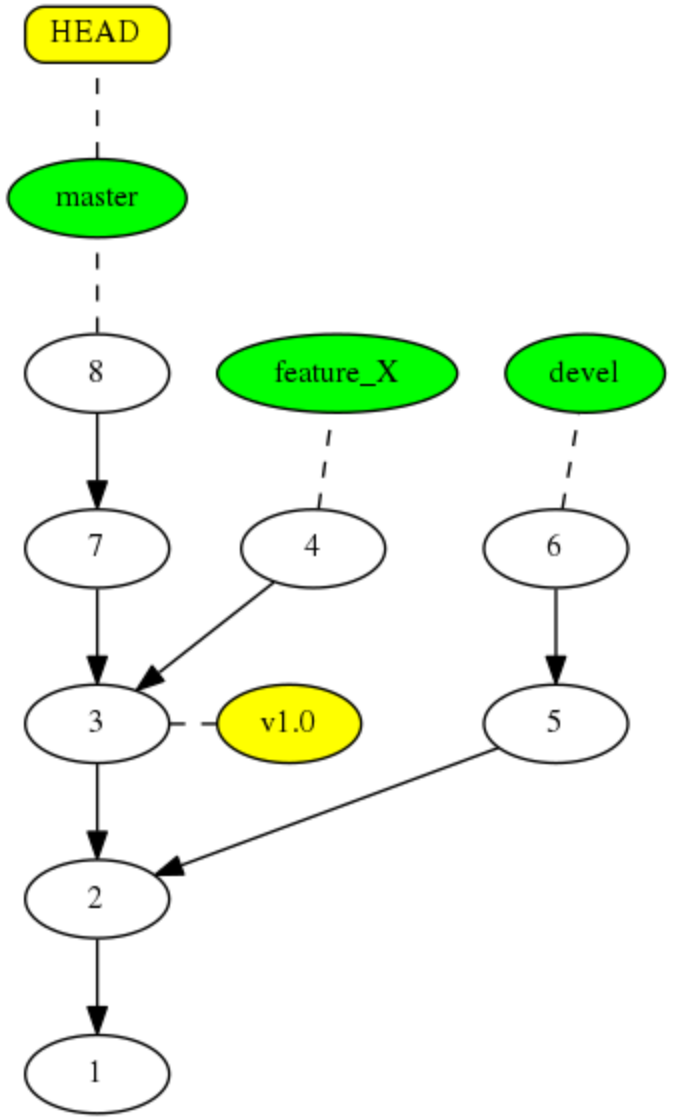
Building Blocks of Git: Committing

- When you make a commit, the current branch ref moves



Building Blocks of Git: Tags

- Tags are like branches, except they never move



Repository Setup: git init

- git init
  - Transform the current directory into a Git repository. This adds a .git folder to the current directory and makes it possible to start recording revisions of the project.
- git init --bare
  - Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag
  - Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite changes. Think of --bare as a way to mark a repository as a storage facility, opposed to a development environment. This means that for virtually all Git workflows, the central repository is bare, and developers local repositories are non-bare.
- git init --shared
  - Specify that the Git repository is to be shared amongst several users.

Repository Setup: git clone

- git clone <repo>
  - Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.
  - Unlike SVN, Git makes no distinction between the working copy and the central repository—they are all full-fledged Git repositories.
  - Creates a remote called “origin” pointing to the location of <repo>

Workflow example: Make a repo from existing code

- cd projectdir
- git init
- git add .
- git commit -m “initial commit”

Repository Setup: git config

- Always set these:
  - git config --global user.name <name>
  - git config --global user.email <email>
- Usability settings:
  - git config --global alias.<alias-name> <git-command>
  - git config --global core.editor <editor>
  - git config --global color.ui always
- For shared-access repositories:
  - git config receive.denyNonFastForwards true
- “git help config” for an exhaustive list

Making Changes: git add

- git add <file>
  - The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit.
  - NOT equivalent to svn add
- git add -p
  - Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

Making Changes: Staging Area

- Sometimes also called the cache, or index
- git status
  - Shows which files are staged, unstaged, or untracked
- git diff
  - Shows a diff between the previous commit and the unstaged changes in the working directory
- git diff --cached
  - Shows a diff between the previous commit and the staged changes in the working directory

Making changes: git commit

- git commit
  - Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you've entered a message, save the file and close the editor to create the actual commit.
- git commit -m "commit message"
  - Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
- git commit -a
  - Commit a snapshot of all changes in the working directory (including unstaged changes, but not untracked files.)

Making changes: git commit

- Commits are local
- Commits can be buffered in your local repository until you are ready to push them to the central repository.
- Commit early/often, without fear of breaking the central repository
- Commit messages: One line summary, detailed description on following lines

Example

- Git add subset of files & commit
- Git add subset of changes within a file and commit
- Examine git status / git log before and after committing

Example Workflow

- Work
- Commit
- Test
- Fix
- Commit
- <Repeat 1-5 as needed>
- Reintegrate other’s changes
- Cleanup
- Push

Inspecting the Repo: Git status

- A summary of the current state of the repo
  - Current branch
  - Ahead/behind origin repository
  - Unstaged/staged changes and untracked files
- Use .gitignore to keep untracked file list clean of build artifacts and other clutter

Inspecting the Repo: git log

- git log
  - Display the entire commit history using the default formatting.
- git log -n <limit>
  - Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits. (git log -3 also works)
- git log --stat
  - Also include which files were altered and the relative number of lines that were added or deleted from each of them.
- git log -p
  - Shows the full diff of each commit
- git log --author="<pattern>“
- git log --oneline
- git log --grep="<pattern>“
- git log <since>..<until>
- git log <file>
- git log --graph --decorate --oneline

Inspecting the Repo: Identifying commits

- Each commit is represented by a 40-character hexadecimal checksum of its contents.
- You only have to specify a enough of the SHA1 to uniquely identify it: the first 5-6 characters are plenty
- HEAD
  - The current commit
- 3157e~1 or 3157e^
  - The parent commit of 3157e
- HEAD~3 or HEAD^^^
  - The great-grandparent of the current commit

Inspecting the Repo: git grep

- git grep <pattern>
  - Search for the specified pattern in tracked files in the working copy.
  - Faster than standard UNIX grep because it only searches tracked files.

Moving Between Commits: git checkout

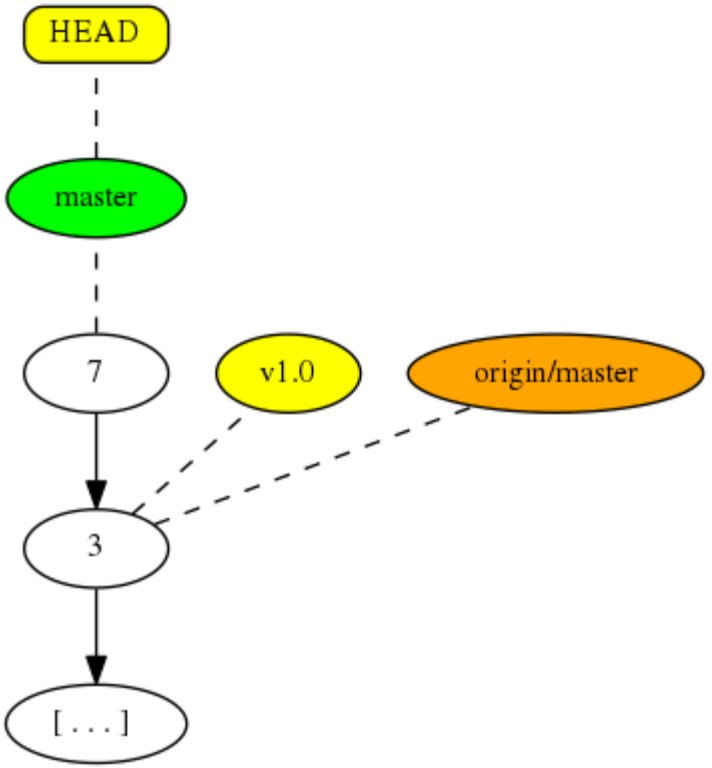
- git checkout <branch>
  - Check out a different branch.
- git checkout <commit> <file>
  - Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.
- git checkout <commit>
  - Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

Tangent: Detached HEAD

- During the normal course of development, the HEAD usually points to master or some other local branch
- When you check out a commit or tag, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state.
- Further commits made from a detached HEAD state will not be saved, because they are not a part of any branch.

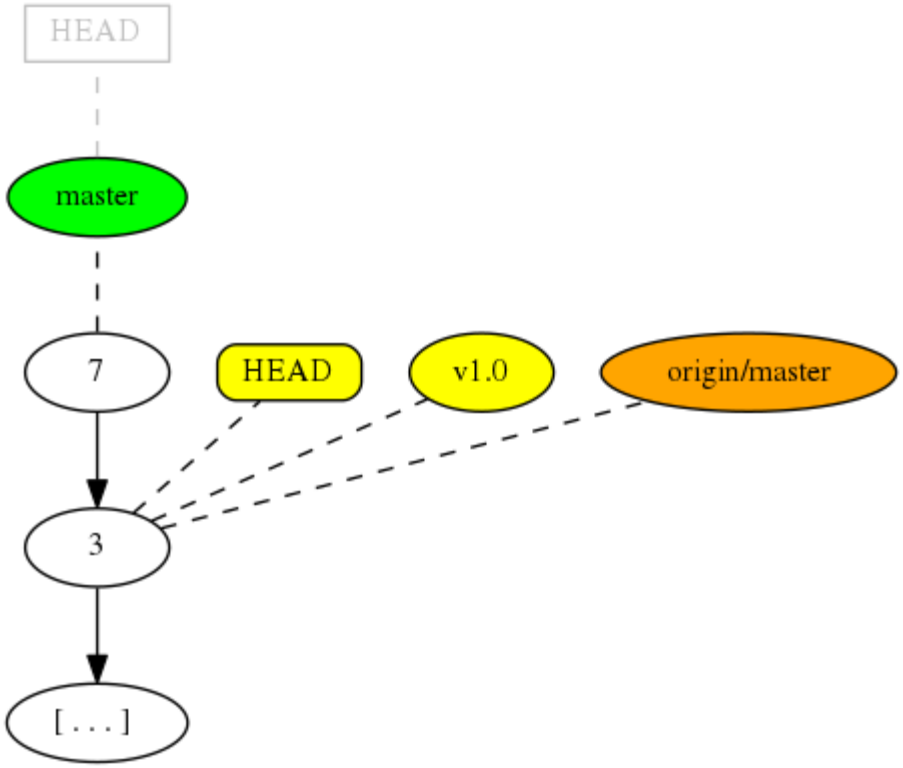
Tangent: Detached HEAD

- Initial state
- HEAD points to master branch



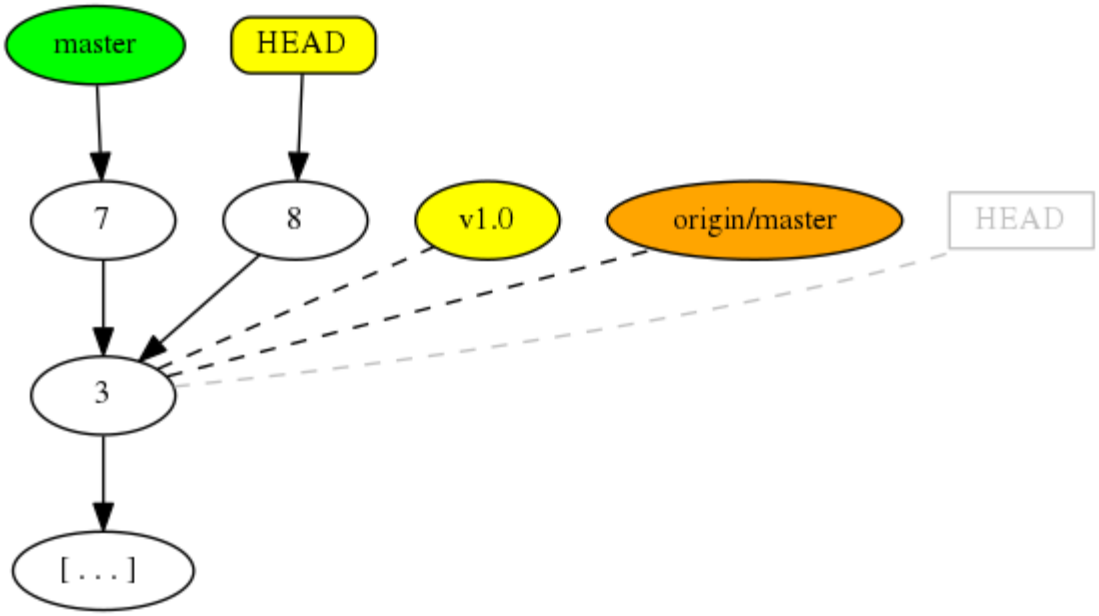
Tangent: Detached HEAD

- Result of “git checkout v1.0”
- Now in a detached HEAD state



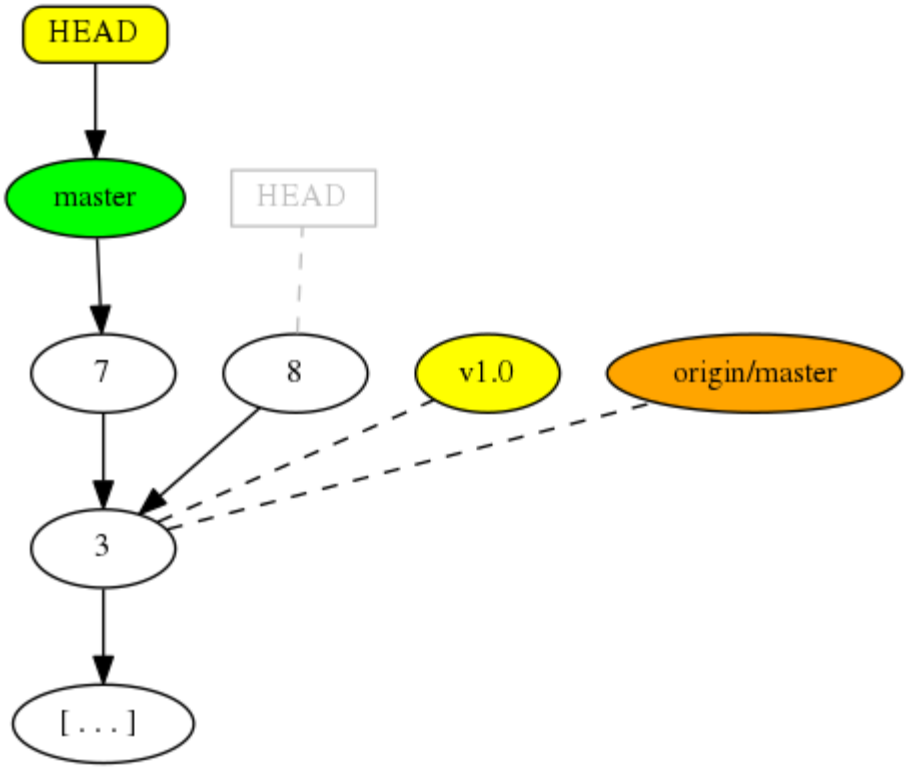
Tangent: Detached HEAD

- Commit 8 is not on any branch



Tangent: Detached HEAD

- Result of “git checkout master”
- Now we have no (easy) way of reaching commit 8, and it is a candidate for garbage collection



Undoing changes

- git checkout <commit> <file>
  - As previously discussed, this form of git checkout can undo changes to a particular file
- git revert <commit>
  - Negates the changes introduced by <commit> by creating a new commit that reverses them.
  - This is a safe way to undo a commit that is already a part of shared history, because the original commit is untouched.
  - NOT like svn revert

Undoing changes

- git reset <file>
  - Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
- git reset <commit>
  - Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory. This can modify shared history!

Undoing changes

- git reset --hard
  - In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. USE WITH CAUTION.
- git reset --hard <commit>
  - Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well. USE WITH CAUTION.
- Use git revert to undo public changes, and git reset to undo private changes.

Undoing changes

- git clean: Removes untracked files from repository. USE WITH CAUTION. Useful for quickly deleting build products
- git clean -n
  - Dry run: don't actually delete anything
- git clean -f
  - Remove untracked files from the current directory. This will *not* remove untracked folders or files specified by .gitignore.
- git clean -df
  - Remove untracked files *and* untracked directories from the current directory.
- git clean -xf
  - Remove untracked files from the current directory as well as any files that Git usually ignores.

Example

- Git checkout
- Git revert
- Git reset
- Git clean

Rewriting History

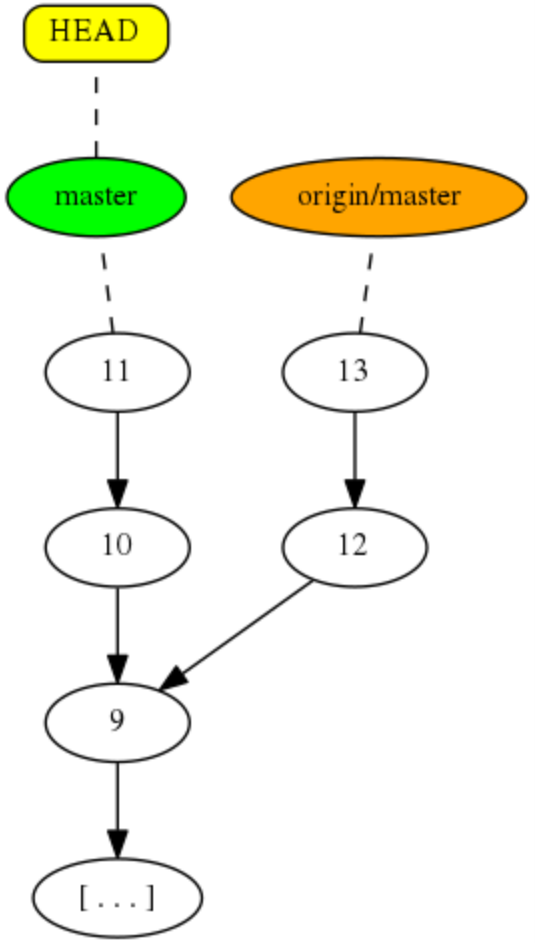
- git commit --amend
  - Combine staged changes with the previous commit instead of making a new commit. Can also be used to simply edit the previous commit message.
  - Don’t use on commits that are part of public history

Rewriting History: rebase

- git rebase <base>
  - Effectively, move a branch to a new <base> commit, branch, tag, etc.
  - This is accomplished by creating new copies of the commits on the new base. The SHA1's will change!
  - Useful for maintaining a linear history.
  - Not to be used on public history.

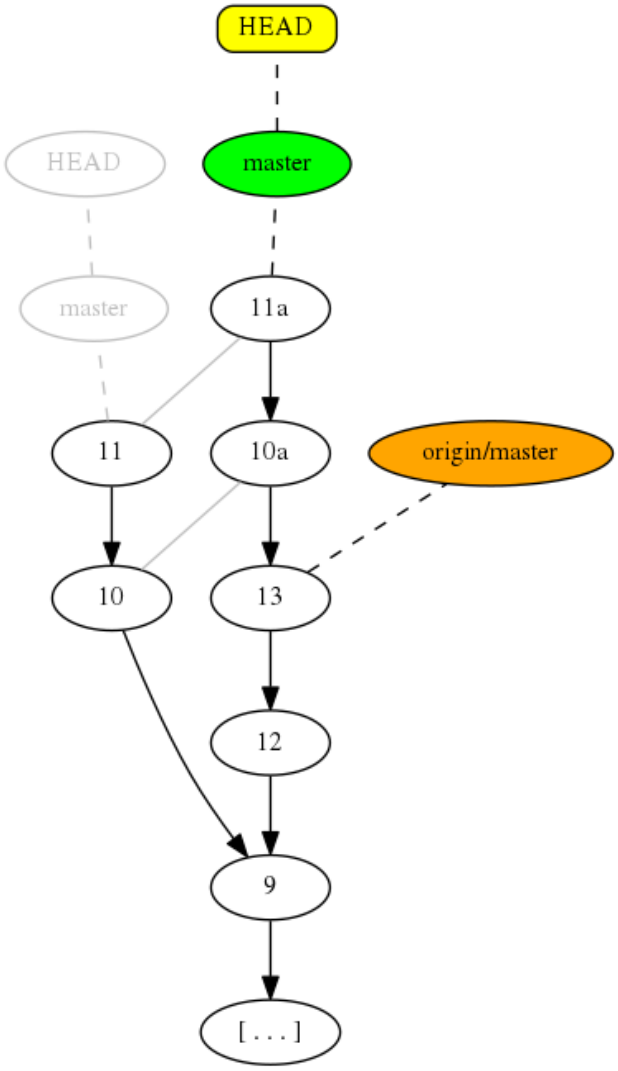
Rewriting History: rebase

- Initial state: origin/master has two commits that are not on our local master



Rewriting History: rebase

- Result of “git rebase origin/master”
- The changes from commits 10 & 11 have been copied to new commits 10a & 11a based on origin/master
- Master now points to 11a
- We now have a linear history that can be easily pushed back to origin



Rewriting History

- git rebase -i <base>
  - Interactively rebase commits, with the opportunity to remove, split, combine, or otherwise alter the commits in the process.
  - Most powerful way to clean up a messy commit history before publishing.
  - Not to be used on public history.

Rewriting History

- git cherry-pick <commit>
  - “Copy” <commit> as a new commit on top of the current HEAD.
  - Useful for grabbing a bugfix commit from a different branch when you don’t want to merge the entire branch.

Example

- Git commit --amend
- Git rebase
- Git rebase -i
- Git cherry-pick

Collaborating

- git remote
  - List remote repositories
- git remote -v
  - Include URLs
- git remote add <name> <url>
  - Add a new remote. In the future, you can use <name> as a shortcut for <url> in other git commands.
- git remote rm <name>
  - Remove a remote

Collaborating

- git fetch <remote>
  - Fetch all of the branches from the remote repository, downloading all commits/files from the other repository.
- git fetch <remote> <branch>
  - Same as above, but only fetch the specified branch.
- This is a safe way to review others commits before merging their work with your own.

Collaborating: remote branches

- Can be checked out just like local branches, but puts you in a detached HEAD state.
- Effectively read-only branches.
- git branch -r
  - List remote branches

Collaborating: Example workflow

- git fetch origin
- git log --oneline master..origin/master
- git checkout master
- git merge origin/master

Collaborating: git pull

- git pull <remote>
  - Fetch the specified remote's copy of the current branch and immediately merge it into the local copy.
  - Same as git fetch <remote> followed by git merge <remote>/<currentbranch>
- git pull --rebase <remote>
  - Same as above, but use git rebase instead of git merge.
  - Makes for a cleaner, linear history, but should not be used if you have already published your history to a different remote.

Example

- Git fetch + git merge
- Git pull
- Git pull --rebase

Collaborating: git push

- git push <remote> <branch>
  - Push the specified <branch> to <remote>, along with all the necessary commits and internal objects.
  - The push will fail if it results in a non-fast-forward merge in the destination repository.
- git push <remote> <branch> --force
  - Force the push even if it results in a non-fast-forward merge. Don't ever use this unless you are absolutely sure you know what you're doing.
  - If receive.denyNonFastForwards is set on <remote> the push will fail even with –force.
- git push <remote> --tags
  - Send all your local tags to the remote repository.
- Only push to bare repositories.

Branching and Merging

- Git branches are much more lightweight than svn, essentially just a reference to a commit, that represents the tip of the branch.
- Using best practices will save lots of heartache when merging
  - Branches should be as short-lived as possible to minimize deviation from master
  - Longstanding branches should periodically merge master changes back into the branch
  - Communicate changes with the rest of dev team
  - Clean commit histories with 1 logical change per commit
  - Have a comprehensive test suite

Branching and Merging

- git branch
  - List all of the branches in your repository
- git branch <branch>
  - Create a new branch called <branch>. The new branch is NOT checked out.
- git branch -d <branch>
  - Delete a branch. Will fail if the branch has unmerged changes.
- git branch -D <branch>
  - Delete a branch, even if it has unmerged changes.
- git branch -m <branch>
  - Rename the current branch to <branch>.

Branching and Merging

- git checkout <branch>
  - Check out the (pre-existing) specified branch, and update the working directory to match.
- git checkout –b <branch>
  - Create a new branch called <branch>, and check it out.
  - Can be used to return from a detached HEAD state.
- Unstaged changes will persist to the new branch, unless they would cause a conflict, in which case the checkout fails.

Branching and Merging

- git merge <branch>
  - Merge the specified branch into the current branch. Git will determine the merge method automatically.
- git merge --no-ff <branch>
  - Always generate a merge commit, even if the merge was a fast-forward.
  - Useful if you want every merge to be documented with its own commit.
- Never start a merge with uncommitted changes in your working tree, as this can make it more difficult to resolve conflicts or abort an unsuccessful merge
- Fast-forward merges occur when there is a linear path from the current branch tip to the target branch. In this case, all git has to do is move the current branch tip up to the target branch tip.
- Non-fast-forward merges occur if the branches have diverged.
- A dedicated commit with two parents is used to tie together the two histories.

Branching and Merging

- Generally, fast-forward merges are used for small features or bugfixes, and 3-way merges are used for the integration of longer-running features.
- git merge --squash <branch>
  - Take all the commits from <branch> and squash them into one commit on the current branch.
  - Useful for integrating a large branch back into master without complicating history. Don’t use this unless the branch is to be deleted after the merge.

Branching and Merging: Strategies

- 3-way merge
  - Advantages
    - Merge history is explicitly shown in repository, and can be used by git to avoid re-resolving conflicts if the branches are merged again later
    - Can safely be performed if the affected branches are already published elsewhere
  - Disadvantages
    - Can make the repository history cluttered and difficult to follow

Branching and Merging: Strategies

- Rebasing
  - Advantages
    - Commit history remains linear and easy to read
  - Disadvantages
    - Modifies history, cannot be used if changes have been published elsewhere
    - No way to tell that the commits originally came from the merged branch

Branching and Merging: Strategies

- merge --squash
  - Advantages
    - Commit history remains linear and easy to read
  - Disadvantages
    - Entire commit history of branch is squashed into one monolithic commit, so granularity of original commits is lost

Example

- Git merge
- Git rebase
- Git merge --squash

Branching and Merging: Conflicts

- Git automatically stages all successful merge results, so “git diff” will show you what is left to resolve
- Resolving merge conflicts uses the same edit/stage/commit workflow as the rest of git
- Workflow:
  - git status/git diff to see conflicted files
  - Edit conflicted files
  - git add each file once conflicts are resolved
  - git commit
- git merge --abort can be used if you wish to abandon an in-progress merge

Branching and Merging: Conflicts

- Git diff uses the “combined diff” format when displaying merge conflicts
- Two columns of data precede each line
- 1st column shows if the line is different between “our” branch (the active branch) and the working tree
- 2nd column shows if the line is different between “their” branch (the branch being merged) and the working tree
- git show <mergecommit>
  - See the combined diff output for a completed merge commit, to see how it was resolved after the fact

Branching and Merging: Conflicts

- git merge -Xignore-all-space
  - Ignore whitespace completely when comparing lines
- git merge -Xignore-space-change
  - Treat sequences of one or more whitespace characters as equivalent when merging
- git merge -Xours
  - In the case of conflicts, always choose “our” side
- git merge –Xtheirs
  - In the case of conflicts, always choose “their” side

Branching and Merging: Conflicts

- git show :1:<filename>
  - Show the common ancestor contents of <filename>
- git show :2:<filename>
  - Show “our” version of <filename>, aka the version on the checked-out branch
- git show :3:<filename>
  - Show “their” version of <filename>, aka the version on the branch being merged in

Branching and Merging: Conflicts

- git diff --ours
  - Compare the working tree to what was in the current branch before the merge, aka see what changes the merge introduced
- git diff --theirs
  - Compare the working tree to what was in the other branch before the merge
- git diff --base
  - See how the working tree has changed from the base commit
- git checkout --ours <filename>
  - Quick way to choose our side of the merge for <filename>
  - Particularly useful when merging binary files
- git checkout --theirs <filename>
  - Opposite of above
- git log --left-right --merge
  - List commits that touch files having a conflict that don't exist on both HEAD and MERGE_HEAD
  - Useful to identify the commits that originally caused a conflict

Branching and Merging: Conflicts

- git checkout --conflict=merge <filename>
  - Re-checkout <filename>, with the conflict markers
  - Useful if you don't like how the merge was going and want to start over for <filename>
- git checkout --conflict=diff3 <filename>
  - Same as above, but use the diff3 format instead
  - Diff3 format shows the contents of the base commit of the merge, in addition to ours and their side
- git config --global merge.conflictstyle diff3
  - Make diff3 the default conflict resolution style

Example

- Conflict resolution

Branching and Merging

- How to delete an unwanted merge commit?
- git reset --hard <commit>
  - Reset the branch to a <commit> before the merge commit
  - Can’t be used on shared history
- git revert -m 1 <merge_commit>
  - Create a new commit that reverts <merge_commit>
  - -m argument tells revert which parent of the merge commit should be kept. 1 is the mainline commit, and 2 is the branch being merged.
  - If you ever want to re-merge the branch again, you’ll need to revert the revert commit first

Tagging

- git tag
  - List tags
- git tag -l “v1.8.5*”
  - List all tags that start with “v1.8.5”
- git tag <tagname>
  - Create a new lightweight tag pointing to the current commit.
  - Lightweight tags are like branches that never change.
- git tag -a <tagname>
  - Create an annotated tag pointing to the current commit.
  - Annotated tags are checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG)
- git tag -d <tagname>
  - Delete an existing tag. Don’t delete tags that are part of public history.
- git push origin <tagname>
  - Push a single tag to origin.
- git push origin --tags
  - Push all tags to origin.

Stashing

- git stash
  - Move all unstaged and staged changes to tracked files onto a stack of changes that you can reapply at any time
- git stash save <message>
  - Same as above, but also give the stash a descriptive message
- git stash list
  - List all stashed changes
- git stash apply
  - Apply the changes from the top stash on the stack to the working tree
- Git stash apply <stash>
  - Apply the changes from <stash> to the working tree
- git stash pop
  - Reapply the top stash from the stack to the working tree and remove it from the stack

Example

- git stash

Archive

- git archive --format=tar --prefix=v1.0release/ v1.0 -o v1.0release.tar
  - Quickly generate a pristine copy of any point in the repository history
  - No need to checkout first
  - Particularly useful for generating a copy of a tagged point in history

Reflog

- Every time HEAD is updated a new entry is added to the reflog
- git reflog
  - Display the reflog
- git reset --hard HEAD@{3}
  - Reset the repository to the state of the 3rd entry in the reflog
- This is git's ultimate undo button
- Particularly useful for retrieving commits made on a detached HEAD, as their SHAs will still be listed in the reflog
- Note that the reflog cannot help you recover changes that were never included in any commit.

Example

- Reflog usage

Bisect

- Performs a binary search over a range of commits to find exactly when a bug was introduced
- git bisect start
- git bisect bad <commit>
- git bisect good <commit>
- git bisect reset
  - Use this when finished to put your working tree back the way it was

Backups

- Local commits are great, but until you “git push” they exist only in your local repo
- If you have many local commits or branches that you don’t want to push back to the central repo, make sure they are backed up somewhere

Other resources

- git help <subcommand>
  - Show the git help for any git <subcommand>
- http://eagain.net/articles/git-for-computer-scientists/
  - If you really want to dive deep into the internals
- https://git-scm.com/book/en/v2
  - Exhaustive guide to using git