# Git Training - Part I

Eric Williams

Red Hat

## What is it?

- Popular version control system invented by Linus Torvalds
- Enables collaboration by many developers
- Used by most teams in the office

## What does it do?

- Tracks changes to files over time, in chronological order
- All changes are attributed to an author/contributor
  - This is very important so you can blame other people for bugs *with evidence!*
- Resolves conflicts
  - For example: two developers making changes to the same file at the same time

## Terminology

## Repository

- The repository is where all files pertaining to a project are stored

- Comprehensive collection of all data including
  - Current state of the project
  - History of changes leading up to that state

- These are stored in a series of *commit objects*

- Repository = repo for short

## Commit Objects

- Every tracked state modification in a repository is represented by a commit object

- Comprised of three things
    - A set of files, which show the state of the project at the time of that commit
    - A parent commit object
    - A SHA1 name: 40 character auto-generated string which uniquely identifies that commit object

## Heads

- A head is just a reference that points to a commit object
  - Conceptually similar to a pointer in C

- By default, there is a head in every repository called *master*
  - Beyond that, a repo can have any number of heads

- If a head is selected at any given time, that head is referred to as the current head
  - The current head’s name is always set to HEAD (caps)

## Branches

- Basically synonymous with a head
    - Every branch is represented by one head
    - Every head represents one branch
    - Default branch is called master
- In day-to-day usage referring to a branch usually means referring to a that commit and all its parent commits (i.e. history)
    - Usually when someone refers to a head, they mean that commit only

## Local Repository Workshop

## Merge vs. Rebase

- Applicable when a branch with a new feature/fix is complete
- How do we get it onto the master branch?
- Every project/community has different preferences
    - Depends on workflow, frequency of contributions, etc.
- Both methods are valid but come with different pros/cons

## Merging

## Merge

- Creates a commit which combines the tip of the master branch (HEAD) and the tip of the feature branch into on commit
- This commit is referred to as a “merge commit”
- The merge commit becomes the new HEAD after the merge is complete

## Pros/Cons

- PRO: non-destructive, doesn’t alter any existing branches
- PRO: simpler to manage, not as complicated as rebase
- CON: tends to pollute the master branch with extra merge commits
    - Becomes an issue in high traffic projects with many contributors
    - Extra merge commits can make bisecting more difficult
        - More on bisecting later

## Merging

- Checkout the branch you want to merge onto
    - In most cases this is master
- *git merge* <branch name>
- Resolve conflicts
    - If no conflicts, then no more work is needed

## Merge Workshop

## Rebasing

## Rebase

- First commit of feature branch is placed sequentially after the tip of the master branch
- No extra commit merging the two
- Partially rewrites the git history by creating brand new commits for each commit in the master branch
    - Requires extra step: rebase feature branch on master, then merge master and feature branch

## Rebase: Pros/Cons

- PRO: cleaner project history, no unnecessary merge commits

- PRO: linear history is maintained
  - Can follow the entire history of the project from the tip of the feature branch back all the way to where master was beforehand

- CON: requires caution
  - Never rebase a public branch onto your feature branch
  - This will result in two different versions of the master branch, which will need to be merged back together

## Rebase Workshop

## Undoing Things

## Reverting

- Things break, commits introduce bugs, etc. -- *this is perfectly normal*
- The most simple form of “undo” in git is a revert
- *git revert* <ID/ref>
    - Creates a new commit that simply un-does the commit which was specified

## Resetting

- Reverting is not ideal if you want to undo something locally without an extra commit
  - A common use case is wanting to undo something on a local development branch
- For this we have the command *git reset*
- Resets the state of the repository back to a certain state in the past, in various ways

## Reset Modes

- Soft: modifies where HEAD points, staged/unstaged changes are not touched; previous commits become staged files
- Mixed: modifies where HEAD points, wipes the index clean (staged files), but doesn't touch unstaged files; previous commits become unstaged files
- Hard: nukes everything, be careful when using
  - Staged and unstaged files reset to the specified commit, HEAD updated, previous commits are gone

## Reset/Revert Workshop

## Reflog

- Git also keeps track of fine grained changes inside the reflog
  - This includes things like checkouts, resets, commits, etc.

- The items listed there are indexable and can be fed to *git reset*
  - Beware however that the reflog isn't permanent, and items far back enough to get cleaned up automatically
  - Unreachable commits are kept for 30 days, unreachable branches for 90

## Cherry-pick

- Picks one commit from a branch and places it on another
  - Useful when only wanting one commit
  - Conflicts will still have to be resolved
- Note that it's useful to specify the *-x* option, which will generate a standardized commit message
  - Usage: *git cherry-pick -x* <ID>

# Reflog/Cherry-pick Workshop