Comprehensive Git

Virtual Logistics

- Put up a green check ✓ or a red **X**:
  - Can you **hear** the instructor?
  - Can you **view** the presentation?

- Virtual Manners
  - Please mute your audio until you wish to speak
  - Unmute at any time and ask questions
  - Also, chat or "hand up" to ask questions
  - Class is interactive - please participate!

Housekeeping

- Roster (attendance)
- Class Times
  - Time zone
  - Breaks, lunch
- Silence cell phones and electronics
- Questions welcome at any time

Course Materials

Put up a green check ✓ or a red **X**:

- **Have you downloaded the course materials?**
  1. Course Manual
  2. Lab Manual
  3. System Requirements (Setup Guide)
  4. Lab Files - explain contents of Setup.zip

     Link to download lab files is in the Setup Guide

- Manuals' tables of contents contain hyperlinks for navigation

Instructor Welcome

Instructor Introduction:

Name
Background
Qualifications / Experience
Something interesting about yourself, hobby, etc.

Course Prerequisites

- Designed for developers
- Contains hands-on technical exercises
- Required setup of the lab environment
- **Have you done your setup?**
  - Put up a green check ✓ or a red **X**

Course Objectives

- Understand basic Git and GitHub concepts
- Effectively use GitHub tools
- Work in a distributed environment
- Understand advanced Git concepts
- Track and recover changes
- Identify and apply correct merging strategies

Setting Expectations

- Hands-on exercises require technical abilities
- **If any of your goals are outside of the scope of this class, the instructor will discuss that here**

Let's Get Started

Introduction to Git

Why Do We Need Version Control?

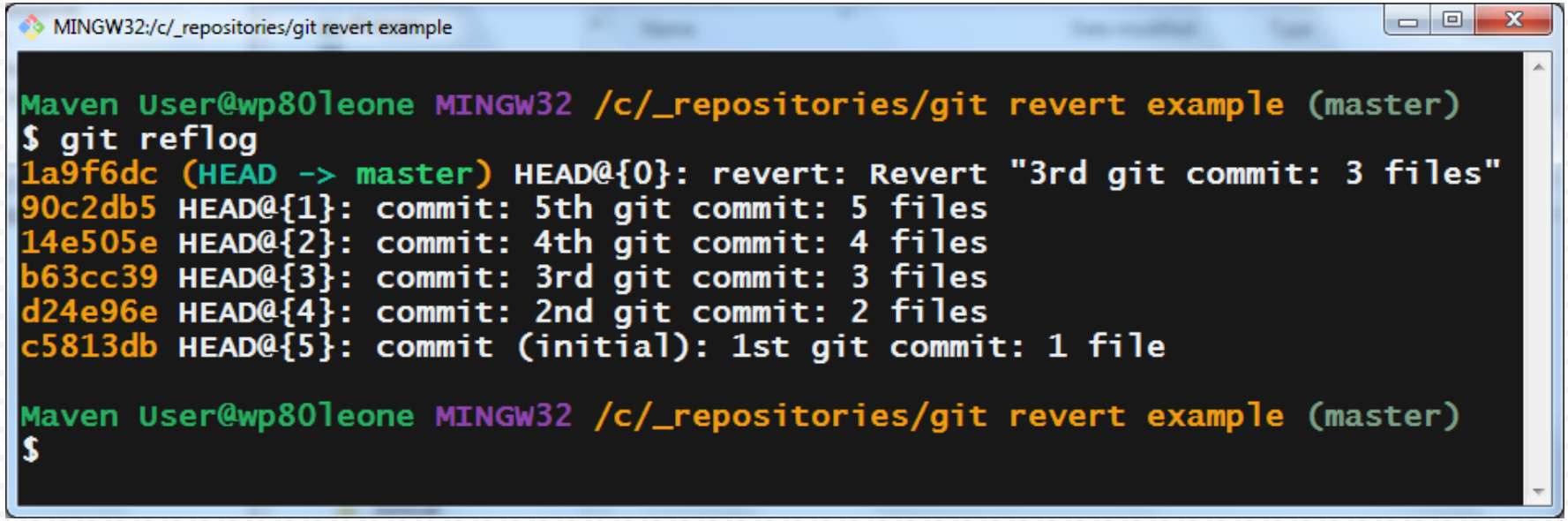
- Tracking changes
- Rolling back to prior versions
- Sharing code with fellow developers
- Isolated development
  - Why else?

Git Command Line

- All Git commands can be run from the command line
- Most GUI tools only allow limited subsets
- Administrators will usually use command line for advanced functionality
- **git init**
- **git status**
- **git add filename.ext**
- **git commit**

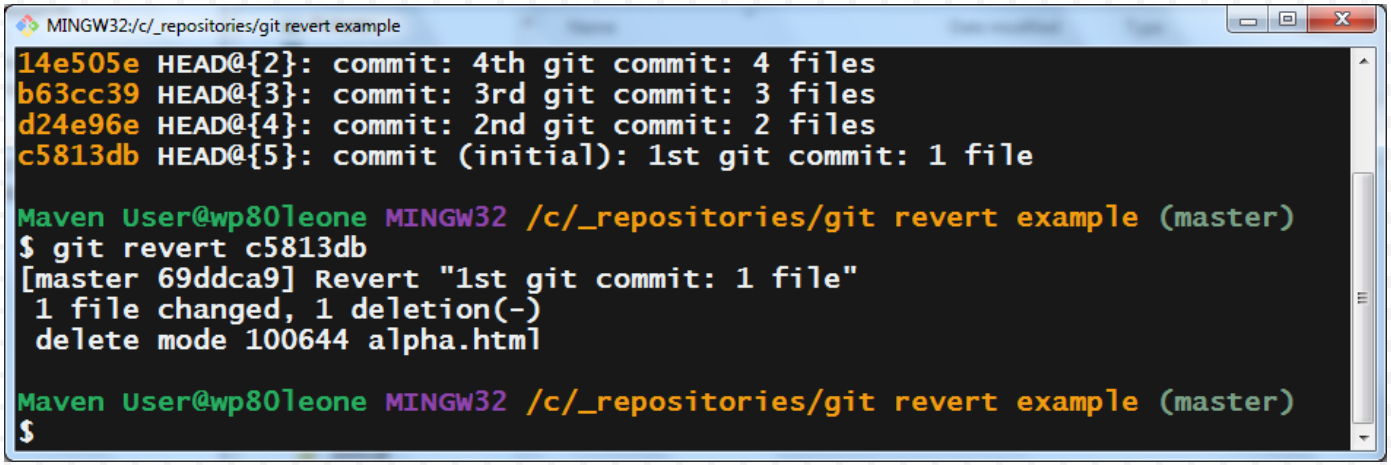
Looking at Commit History

- One of the benefits of using a version control system is the ability to view the commit history
- The `git reflog` command shows you a history of commits, along with commit messages



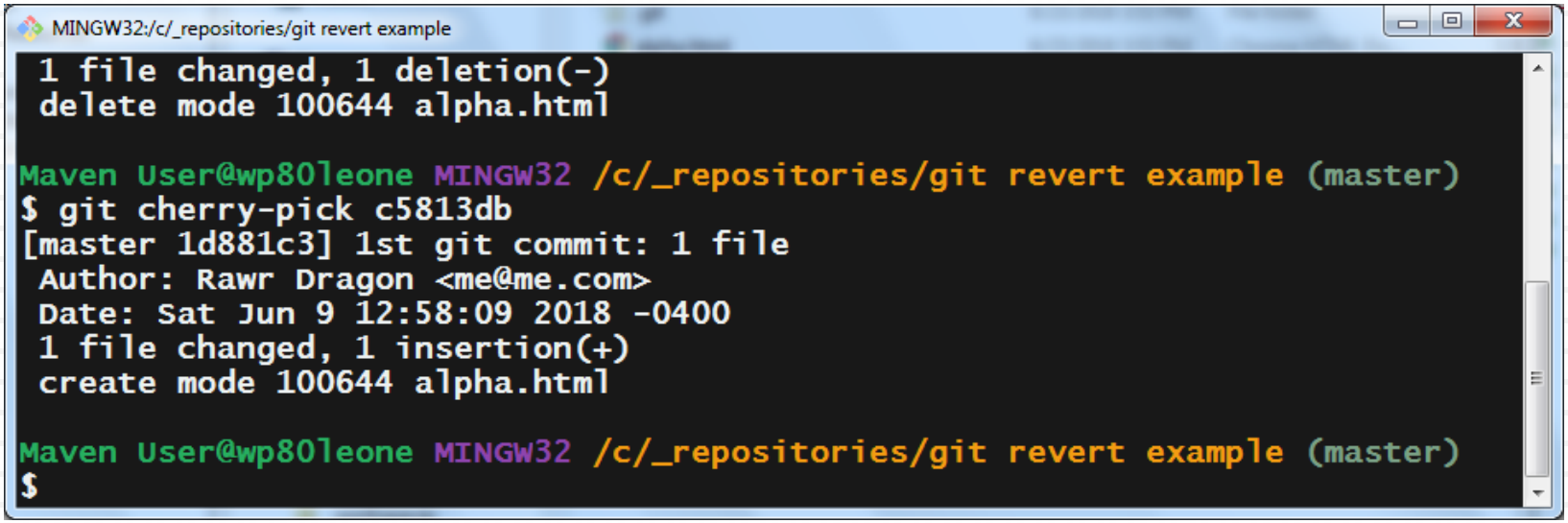
How to revert a commit

- If the changes added during a given commit were bad, the commit can be reverted
  - The command is `git revert`
  - This removes all of the changes of a past commit from the current code base



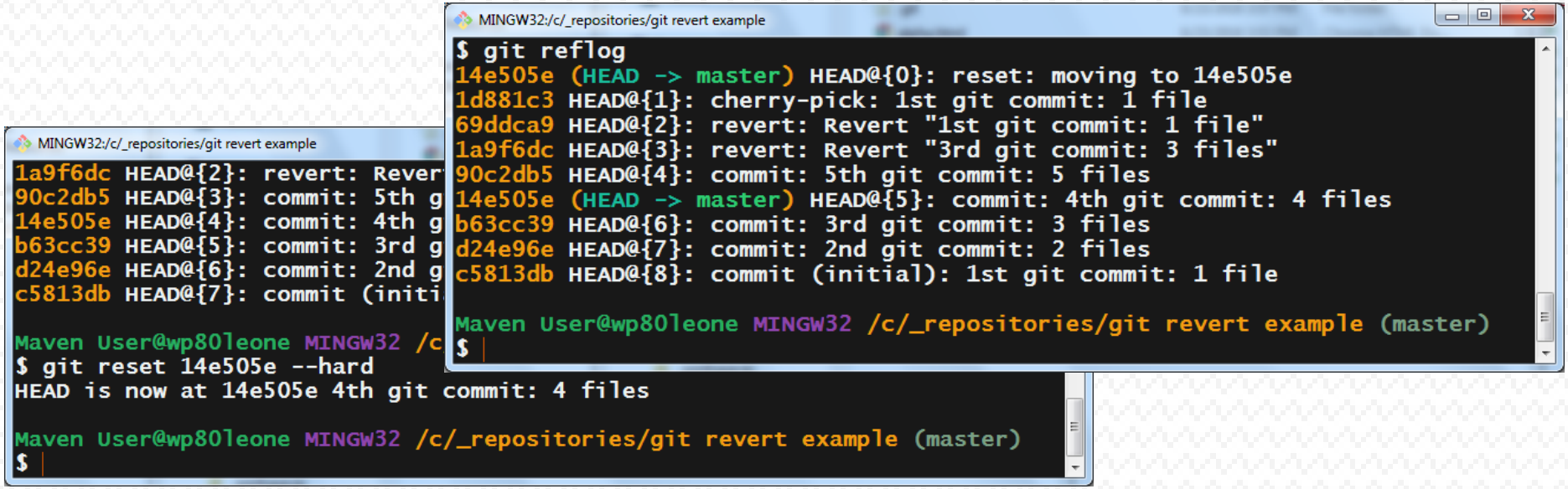
Cherry picking

- If you want to restore changes that were associated with a past commit, you can cherry-pic it
  - Command is git `cherry-pick`



Go back to an old commit

- You completely restore your entire working tree, index and repository head to a previous commit
  - The command is `git reset`
  - The --hard tag must be used to restore everything



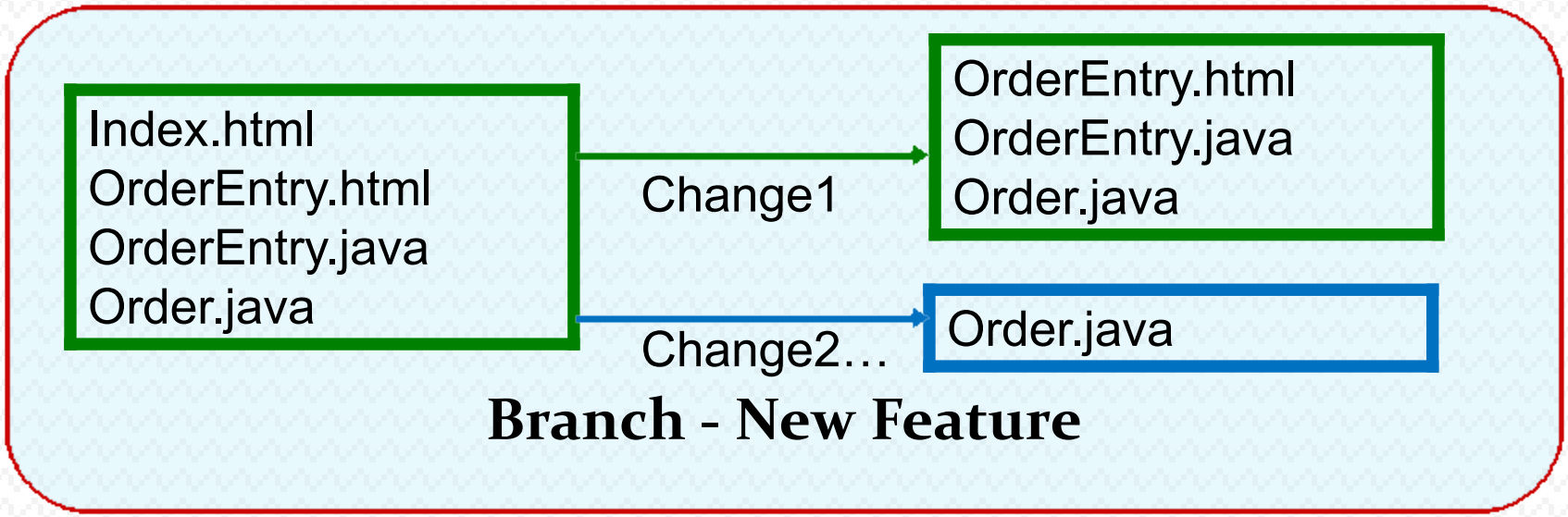
Summary

- Basic process of tracking changes with Git
  - Init
  - Stage
  - Commit
- Any questions?

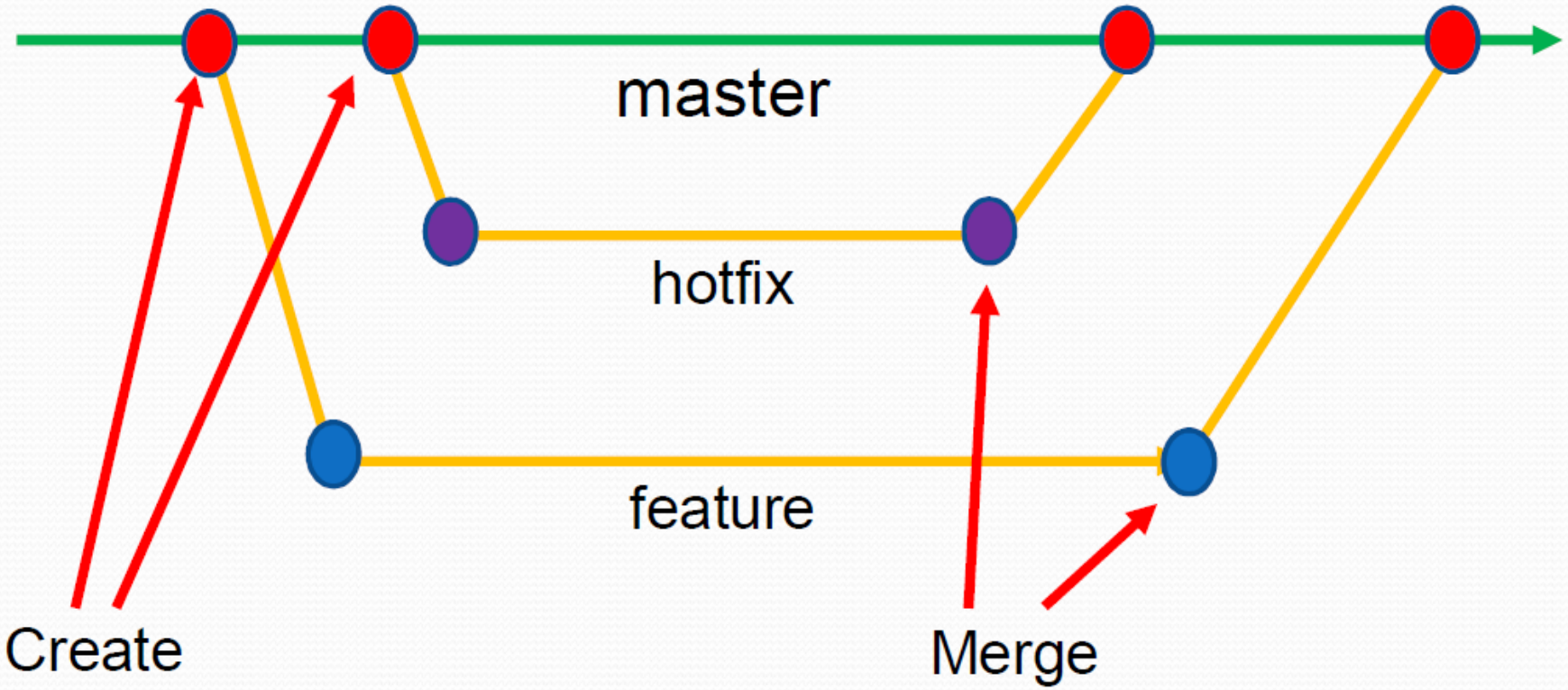
Branching

Why Branch?

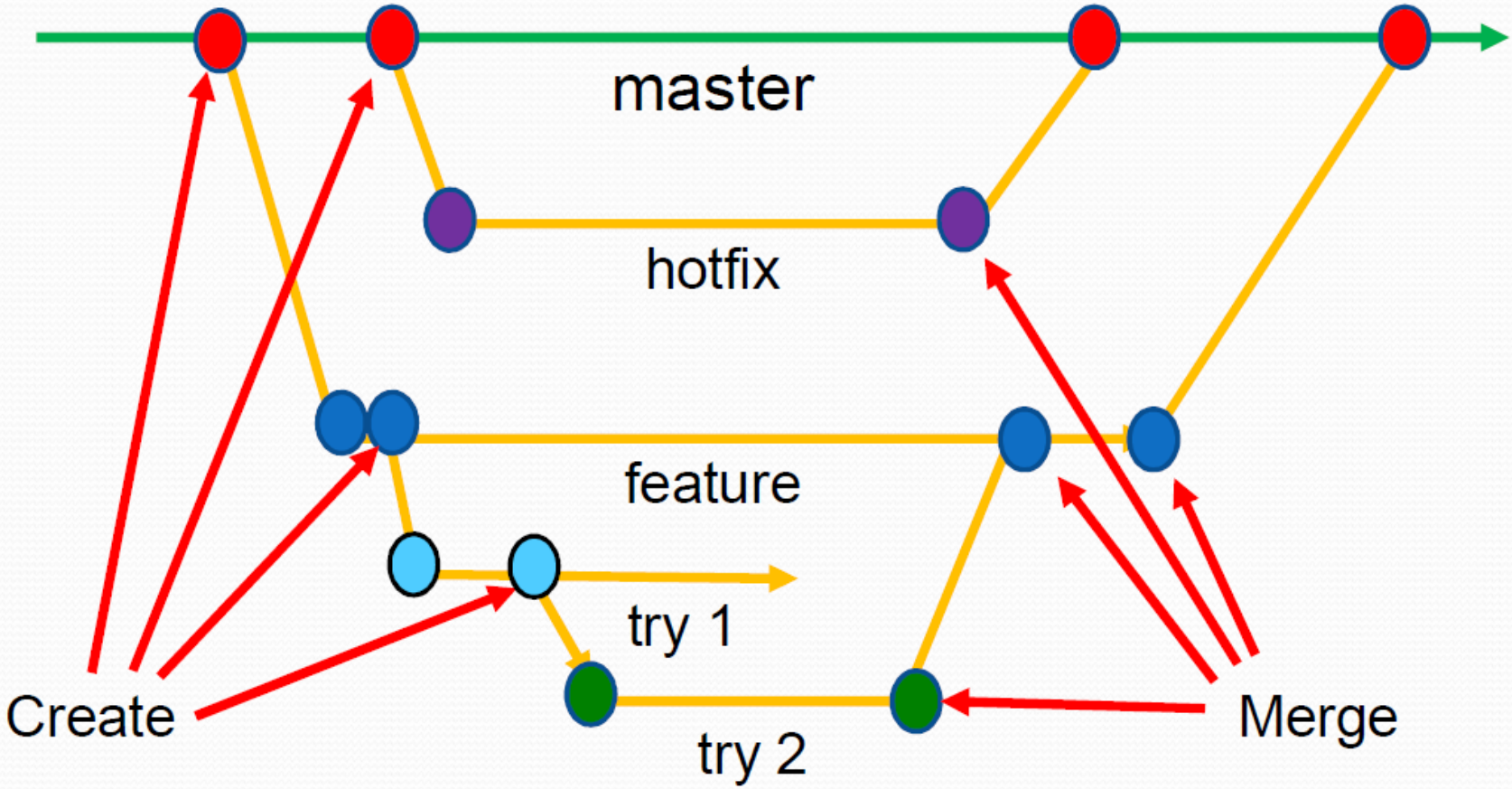
- Identify group of related changes on multiple files
- Maintain as a logical unit of work
- Push as a unit
- Once work incorporated in mainline - delete



Git Branch



Git Branch with a Twist



Commands

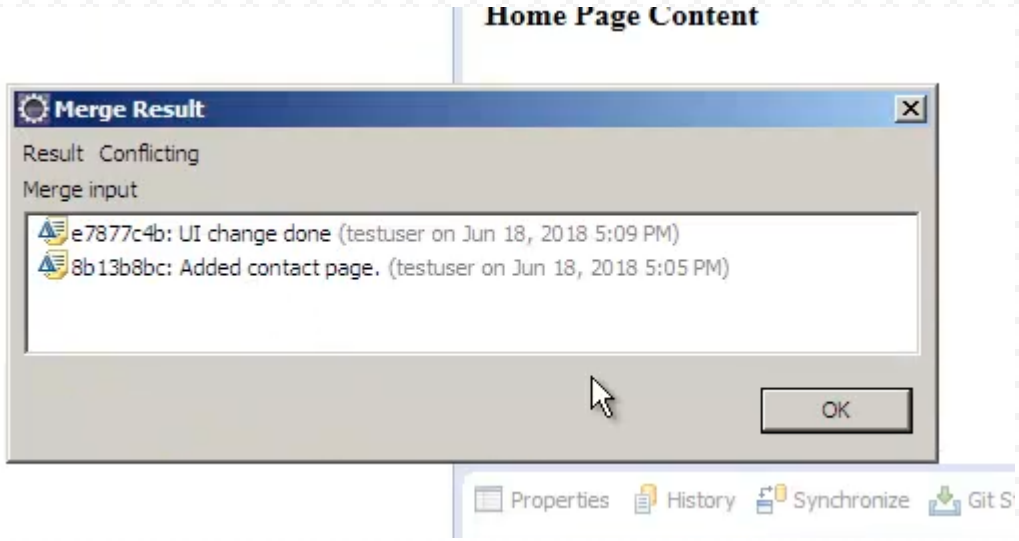
- **checkout** - bring a branch into workspace
- **branch** - create or delete a branch
- **merge** - merge a branch to another branch
- **rebase** - move the base of a change from the middle to the tip
- **status** - list merge status of changed files
  - **merged**
  - **unmerged**
  - Shows conflicting paths
  - Need to reconcile changes
- Commit the changed files to complete the merge

Basic Workflow - Local

- Check out the master branch
  - Updates the workspace with last committed changes
- Create a new branch to do some work
- Make changes to code
- Add the files to the staging area
- Commit the changes - provide the comment
- Merge the code into the master branch
- Delete the new branch

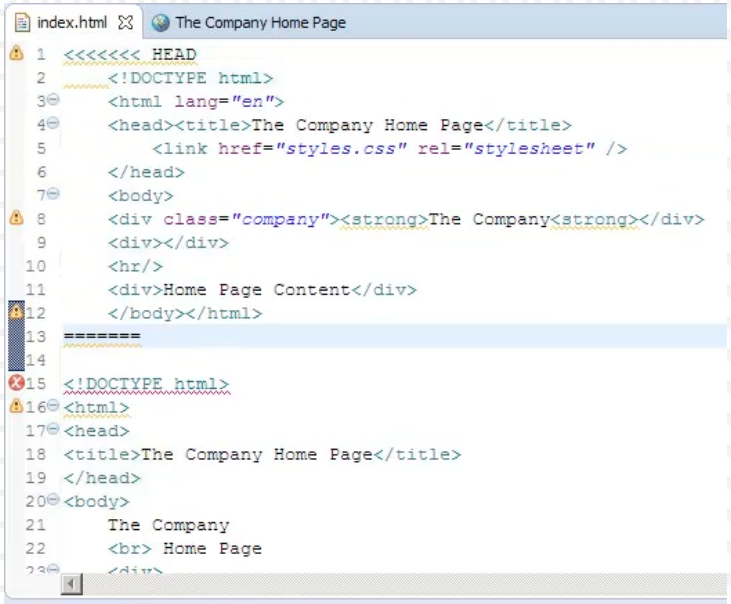
Problems with Branching

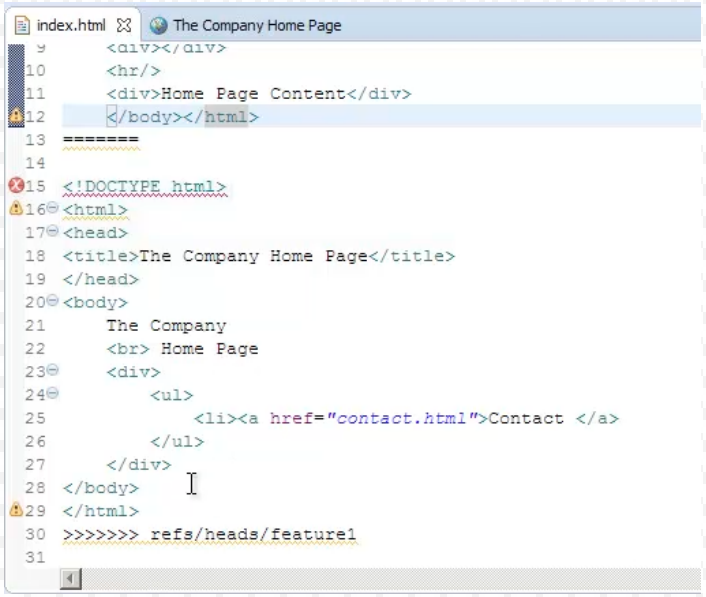
- When you branch, there is the looming issue of merging back into the branch
  - If two people have edited the same file, a conflict may arise



Resolving conflicts

- If a merge conflict appears, git will highlight all of the conflicting changes within the file
  - You must resolve the conflict, save your changes, make a new commit and merge back in





Summary

- Branching is cheap and easy
- Just a reference in the repository
- Use all the time - even daily
- Doesn't hurt to create a branch for every change

Distributed Git

Distributed Git Commands

- There are four basic Git commands for working with remote repositories:
  - `git clone`
  - `git pull`
  - `git push`
  - `git fetch`

Git Clone

- Git clone copies a remote repository to your local file-system
  - Creates and checks out an initial branch based on the remote repository's active branch
  - Creates remote tracking branches for all branches in the remote repository
- The git clone command initializes a git repository so there is no need for a git init command
- This is a one-off command that is only performed at the beginning of development

Git Push

- The git push command is used to move all of your local branches and commits to a remote repository
  - A push will only succeed when it is a fast-forward merge
  - If files have been added to the remote branch since it was checked out, those files must be pulled down
  - After pulling those files down, perform a commit and a push will succeed
  - This is not a merge conflict. It's a synchronization.

Git Pull

- The git pull command takes changes made to a remote branch and brings those changes into the local working directory
  - Creates a new commit when successful
  - Synchronizes the local branch with the remote branch
    - Actually a combination of fetch and merge
- Will fail if the merge part of the pull causes a conflict
  - All the more reason to work on isolated branches!

Git Fetch

- The git fetch command is used to update remote tracking branches
  - Pulls remote changes down to the local machine
  - Does not merge changes into the local working directory
- Required if a pull causes a conflict
  - Fetch to update your remote tracking branches
  - Merge your local code with the tracking branch
  - Commit and push!

Summary

- Push changes to remote repositories
- Pull changes from remote repositories
- Merge is sometimes required

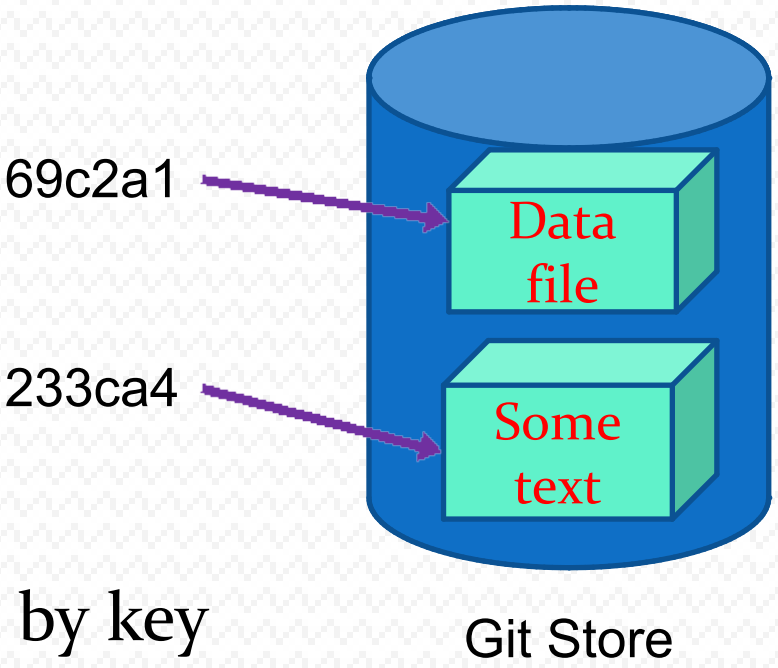
Git Internals

Git Directory Structure

- Directories
  - **hooks** - triggers run on specific events
  - **info** - global exclude like **.gitignore**
  - **objects** - directories of objects - 2 hex digit names
    - Each file under this is named the rest of the hash
  - **refs** - references to hashed objects
- Files
  - **config** - configuration settings per repository
  - **description** - used by **GitWeb** for display
  - **HEAD** - reference to the branch currently checked out
  - **index** - staging area information

Git File System

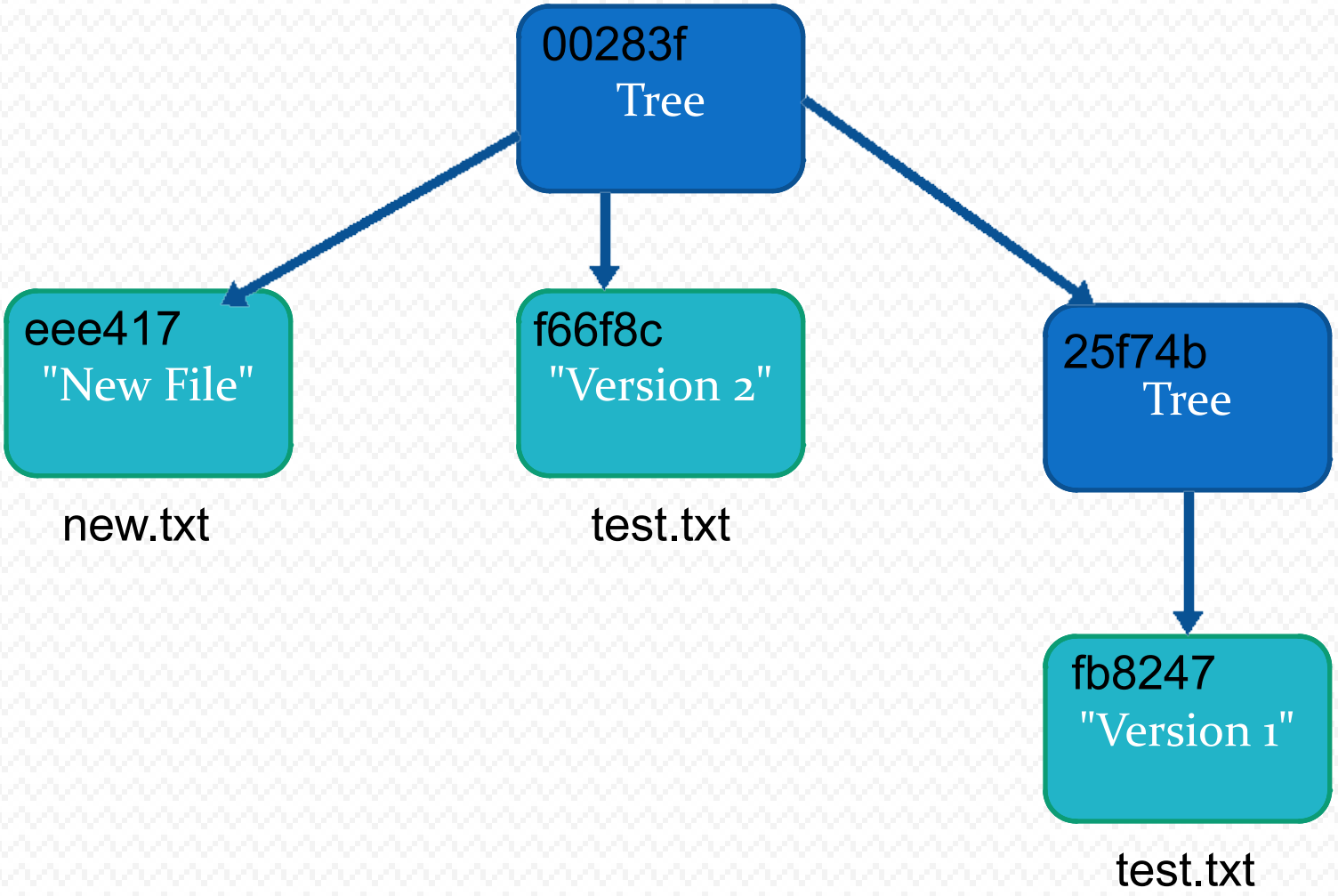
- Content-addressable file system masquerading as a VCS
- Stores key=value pairs
  - Insert content
  - Get a key (hash)
    - In the object directory
    - Packed in a zip
- Can store any content
- Can search/read the content by key



Objects types

- Blob - Binary Large Object
  - Anything that is not one of the below items
  - Source code
  - Built artifacts
- Tree
  - Like a folder structure
- Commit
  - Unique ID and metadata for a reference to a tree
- Tag
  - Unique ID pointing to metadata and a commit
  - Metadata include the tag name

Objects


00283f
Tree
eee417
"New File"
new.txt
f66f8c
"Version 2"
test.txt
25f74b
Tree
fb8247
"Version 1"
test.txt

Internal Plumbing Commands

- **hash-object** - creates the hashed object, returns hash
- **cat-file** - like cat but from a hashed object
- **update-index** can update the index area
  - Plumbing version of the staging area
  - Add index entries to the structure of the file system
- **write-tree** is used to write the staging area as a tree
- **read-tree** - reads the content of a tree written with write-tree back into the staging area
- **commit-tree** - used after building a set of content in staging area to move to the repository

The filter-branch command

- The filter-branch command can query content information
- The filter-branch command can change history
- The filter-branch command can remove content
- **Warnings:**
  - Never rewrite history on a public branch
  - Can lose history or content permanently
- Example:

```
git filter-branch --msg-filter 'cat && echo
"Acked-by: Some User someuser@somedomain.com"'
HEAD~5..HEAD
```

filter-branch

- Powerful tool for modifying the repository
- *"With great power comes great responsibility."*
  - Be very careful
  - Only use when REALLY SURE it is necessary
- Options:

  `--env-filter` committer and author information

  `--tree-filter` rewriting the tree and tree content

  `--commit-filter` rewrite the commit tree

  `--index-filter` rewrite the index and cache - not committed content

  `--parent-filter` rewrite the commit's parent list

filter-branch Example

Example **--env-filter**

```
git filter-branch --env-filter
        ' if test "$GIT_AUTHOR_EMAIL" = "auser@mydomain.com"
        then
                GIT_AUTHOR_EMAIL=buser@adomain.com
                export GIT_AUTHOR_EMAIL
        fi
        if test "$GIT_COMMITTER_EMAIL" = " auser@mydomain.com "
                then GIT_COMMITTER_EMAIL=buser@adomain.com
        export GIT_COMMITTER_EMAIL
        fi ' -- --all
```

The Growing Git Repo

- The repository grows with every new commit
  - Unless requested, nothing is ever removed or packed
- Everything is saved
  - All new versions of all objects
  - All old versions of all objects
  - Each consumes space

Git Count

The **`git count`** command can be used to discover information about the size of your repository

- Git way: **`git count-objects -v`**
  - Returns the count of objects and size of objects
  - If packed, the size of packed objects
  - Returns objects not yet garbage collected

GIT Pruning

- One way to prune is to run **`git fsck --unreachable`**
  - Gathers all unreferenced objects
  - Removes unreferenced objects from repository
- There's a better way, though
  - Use **`git gc`** instead

Before and After git gc

BEFORE

```
.git/objects/00/2083faeb22b1375592b9a9b95a8e657f392066
.git/objects/04/7b4f7407d61383124bf20c54dfa4ce297faff7
.git/objects/25/f74b173d389e86594e06d86561864ee65be6e6
.git/objects/4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects/56/389227d7ec48ed48014a1f985890ab339a045b
.git/objects/5d/3c4b4521e52a9f4d83839a81ff7dba9133e684
.git/objects/7b/57bd29ea8afbdeb9bac64cf7074f4b531492a8
.git/objects/9d/12eb662df7bd686ea6ca7adf2692030fc50e04
.git/objects/ba/6be2935effd4cb0d92b49ede0e10904b24b341
.git/objects/de/5b2ff5dd8bc2f44118d07e4206df2fd483045f
.git/objects/ee/e417f1b97a31f35ceb5f0e7d0bae6e78c5290a
.git/objects/f4/4f8c6b9a98205af69015113a3b7fdda0962e5b
.git/objects/fb/8247c7b27ae4cad9e7e3e66ba95126658ea7c2
```

AFTER

```
.git/objects/25/f74b173d389e86594e06d86561864ee65be6e6
.git/objects/4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects/5d/3c4b4521e52a9f4d83839a81ff7dba9133e684
.git/objects/7b/57bd29ea8afbdeb9bac64cf7074f4b531492a8
```

Summary

- Git is a file system masquerading as a VCS
- Plumbing commands can do the simple stuff
- Most commands string simple commands together
- Every version of everything done is available unless explicitly removed
- Git keeps everything in text files or a packed blob

Git Configuration

Agenda:

- Git Internals and Infrastructure
  - Repository size
  - File System Check
  - Pruning
  - Filter-branch
  - Garbage Collection
- **Global and Local Configuration**
  - Configuration
  - Customization
  - Finding Content
  - Debugging
  - Client Hooks and strategies
  - Server Hooks and strategies
- Git Collaboration
  - Rebase
  - Dry-runs
  - Configuring remotes
  - Refspecs
  - Archives (send & receive)
  - Patching
  - Sub-modules
  - Cherry-Picking

Git Configuration Scopes

- Git has three levels of configuration scope
  - System level set for the installation
  - Global level set for the user
  - Local set independently for each repo created

Git Configuration Locations

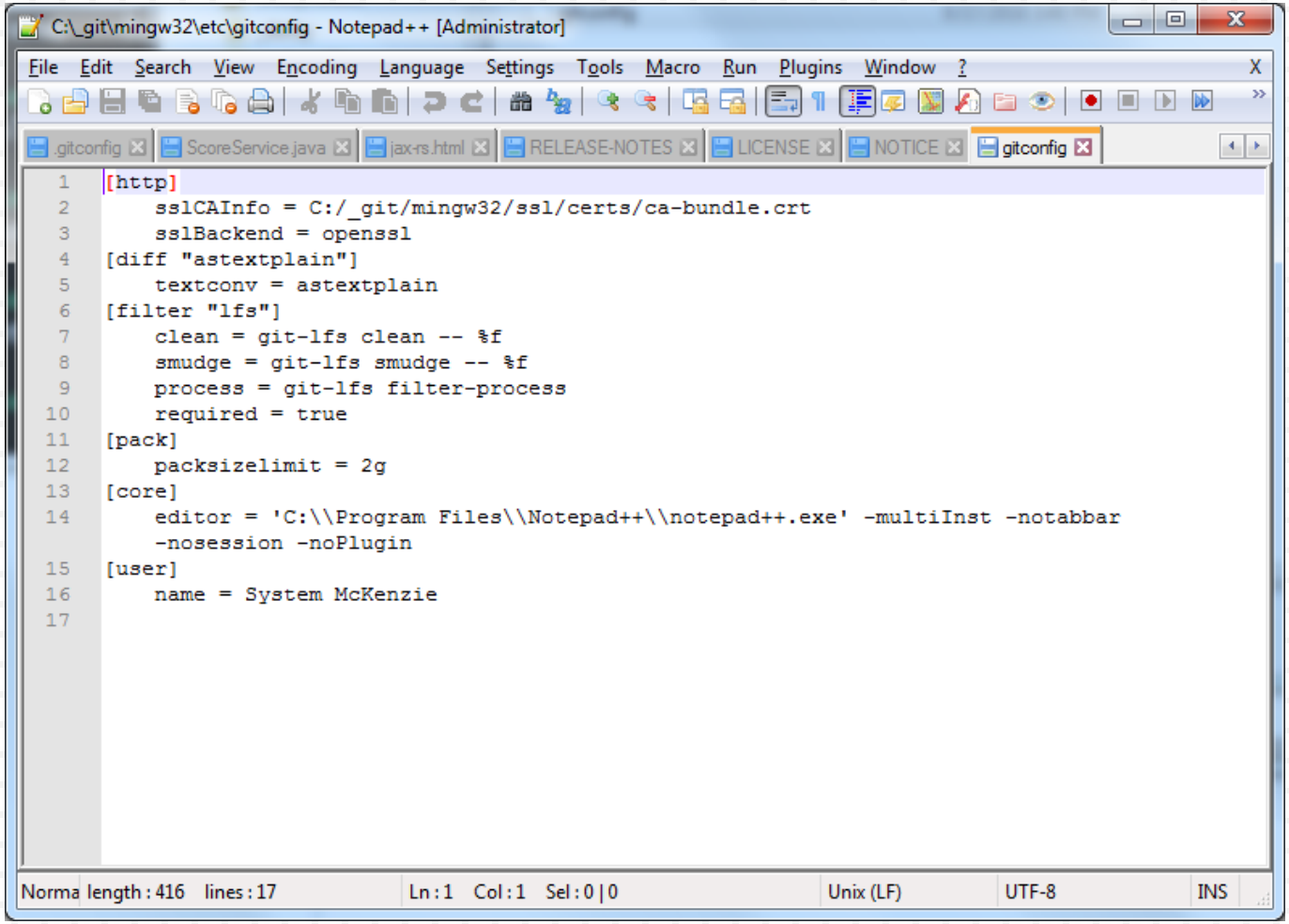
- First git command tries to set up an environment
- User configuration ~/**.gitconfig**
- New configuration location **$HOME/.config/git/config**
- System configuration **/etc/.gitconfig**
  - Windows or Mac Git Install folder/etc/gitconfig
  - Windows C:\ProgramData\Git\config
- Repository Specific **.git/config**
  - Set these options after installation

```
git config --global user.name "John Doe"
```

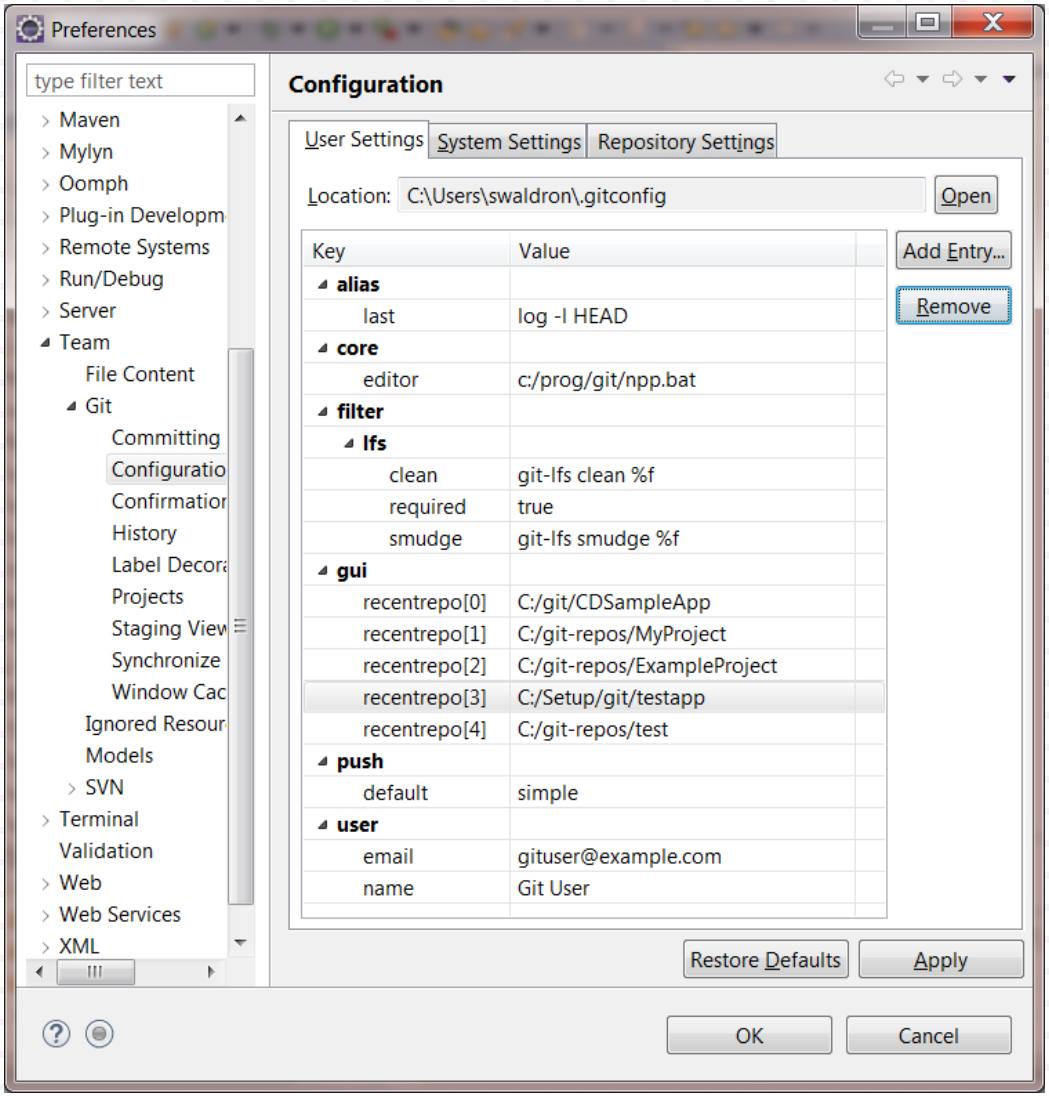
Config File Syntax

- Section names in brackets are case-insensitive
- Subsection names are case-sensitive
- Subsections are optional
- All variables must be in a section
  - Alphanumeric characters, underscore _ and dot .
- [section subsection]
  - Variables - name=value pairs and must be in a section
- Names without a value are true
- Names use the same conventions as section

Sample System Config



Simplified Config in Eclipse



Common Settings

- **EOL** - line endings - Windows vs. everything else
  - Other EOL settings **safecrlf** and **autocrlf**
- **Bare** - repository does not have a working tree
  - Cannot be used for development
- Color settings are true, false, never or always
  - Can be global or per project
- Merge strategy settings
- Hooks
  - Client-side - user convenience
  - Server-side - enforce policy

Poll

1. What is the command to change settings?
   A. config
   B. edit
   C. configure

2. Project settings are:
   A. -- global
   B. -- local

Additional Settings

- Core **commit.template** can point to a default commit

  ```
  commit.template $HOME/.gitmessage.txt
  ```

- For signing content

  ```
  user.signingkey $HOME/.gpg/my.key
  ```

- External **diff** tool
  - Perforce Visual Merge tool
  - Kdiff3
  - Meld
- Server side
  - **receive.denyNonFastForward** - defaults to true

Controlling Text/Binary Options

- Git also uses a file **.gitattributes**
- Pattern setting pairs
  - ***.java** text
  - ***.c** text
  - ***.gif** binary
- Diffing a file that is defined binary is true/false
- Word can have special support
  - ***.doc diff=word**
  - In **config diff.word.textconv catdoc**
  - Word documents, when encountered, are converted to text for comparison

Filtering

- Filtering is configured within config files
  - **Smudge** - filter file to working directory
  - **Clean** - filter file to staging area

- Example:

```
filter.indent.clean indent
```

  - Indents the file moving from working to staging

- Could use a program to substitute **$Date$** or similar

Client Hooks

- Workflow hooks
  - **pre-commit**
  - **prepare-commit-msg**
  - **commit-msg**
  - **post-commit**
- Email hooks
  - **applypatch-msg**
  - **pre-applypatch**
  - **post-applypatch**
- Additional client hooks
  - **pre-rebase**

Server Hooks

- Triggered on the server to process workflows
  - **pre-receive** - runs before the receive with list of refs
  - **post-receive** - runs after receive completed and lists what was pushed
  - **update** - runs for each branch before updating

Searching Git

- Git is a file system apart from the OS file system
- Need to be able to search within it
- **Grep** is the tool
- Has similar syntax to **grep**

```
-e or -G(extended) or -P (perl) -F (fixed string)
-v (invert)
```

- Example:

```
git grep -e 'MAX_LEN'  --and \( -e 'Entity' -e
'Table' \)
# output the count of lines
git grep -c  -e 'public' --and -e 'final'
```

Debugging

- **Blame** - information about committer of a li
- Can track a specific line through the commit history

  Example: `git blame -L 50,60 test.txt`

  Output: `"Git User <gituser@email.com>"`
- **Bisect** - uses a binary search to find commit introducing a bug
- Example - start and set good and bad versions:

```
git bisect start
git bisect bad # current version is bad
git bisect good v1.0 # last known good
version
```

Poll

Which of these are debugging tools?

A. blame
B. bisect
C. merge
D. rebase

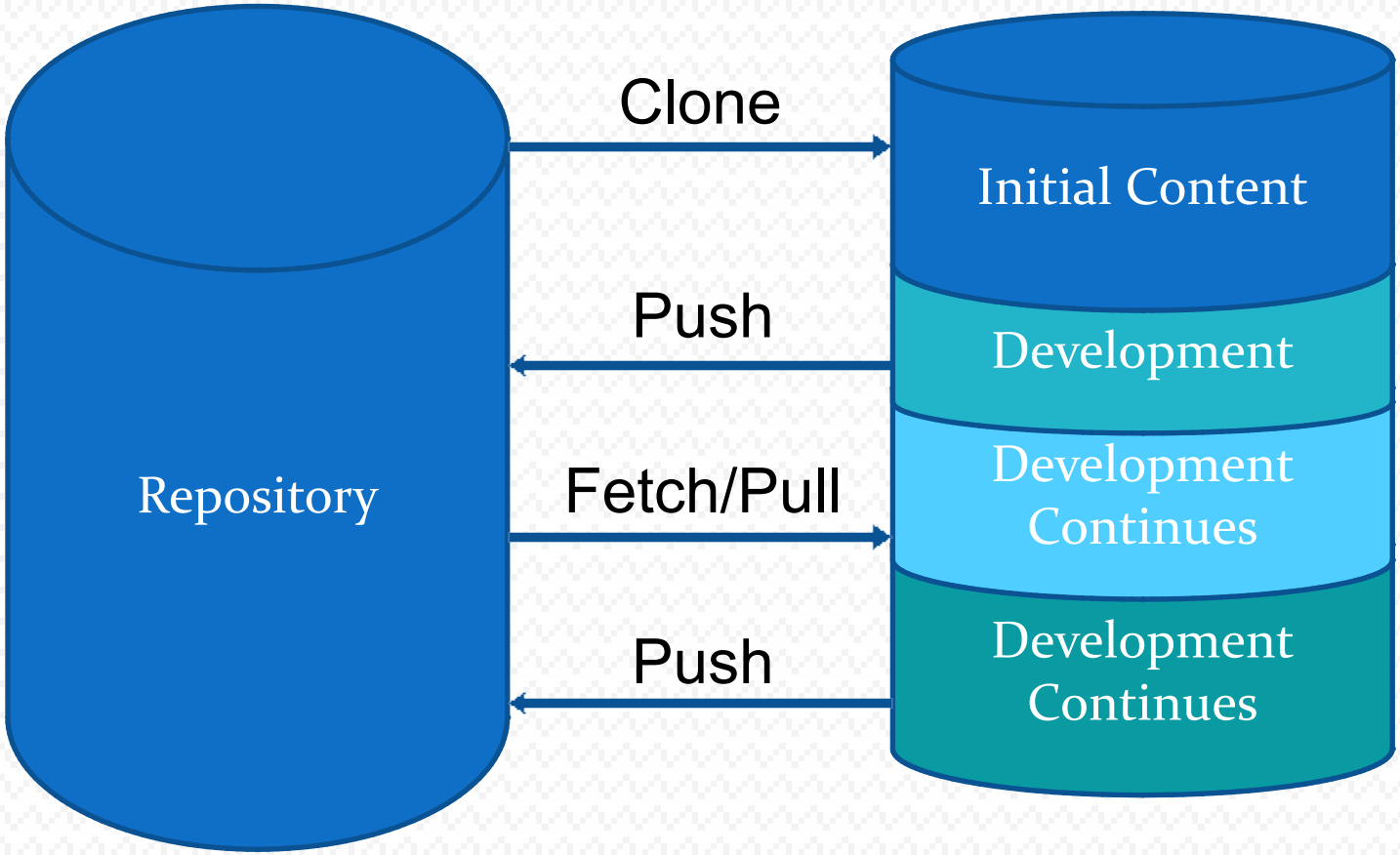
Summary

- Recovering versions is as simple as pointing a reference
- Customizing Git can be done using any editor
- Command line **git config** can set any setting
- Global settings apply to all projects
- Local settings apply to a project

This slide has been intentionally left blank.

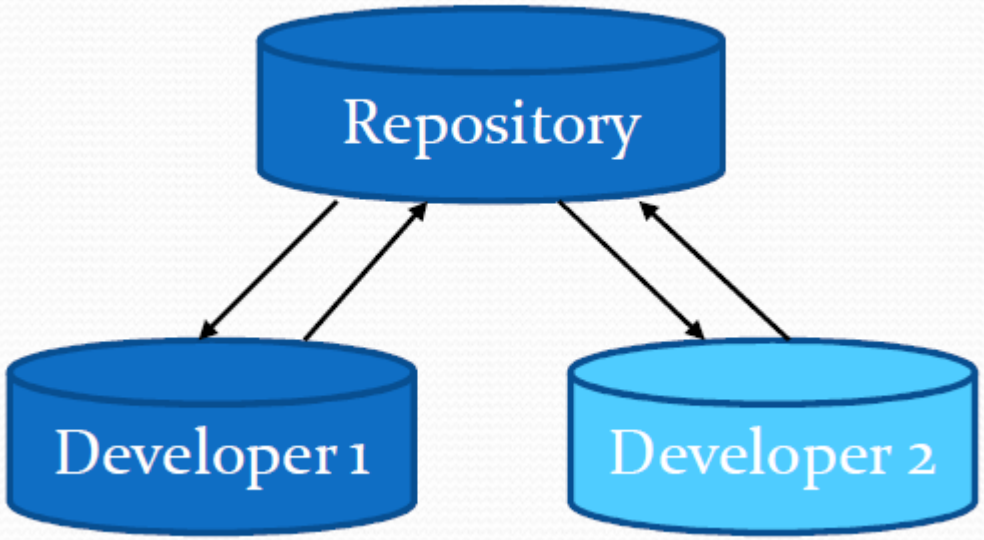
Git Collaboration

Workflow


Repository
Clone
Push
Fetch/Pull
Push
Initial Content
Development
Development Continues
Development Continues

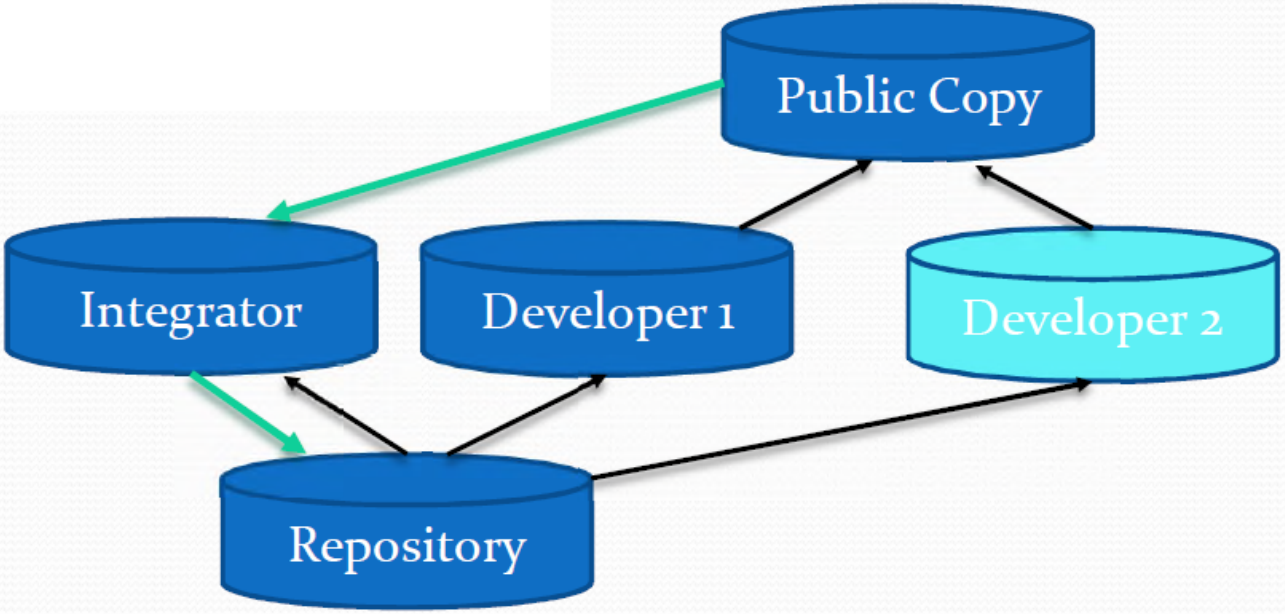
Centralized Workflow

- Everyone contributes equally
- Everyone merges before pushing
- Works like CVS or SVN



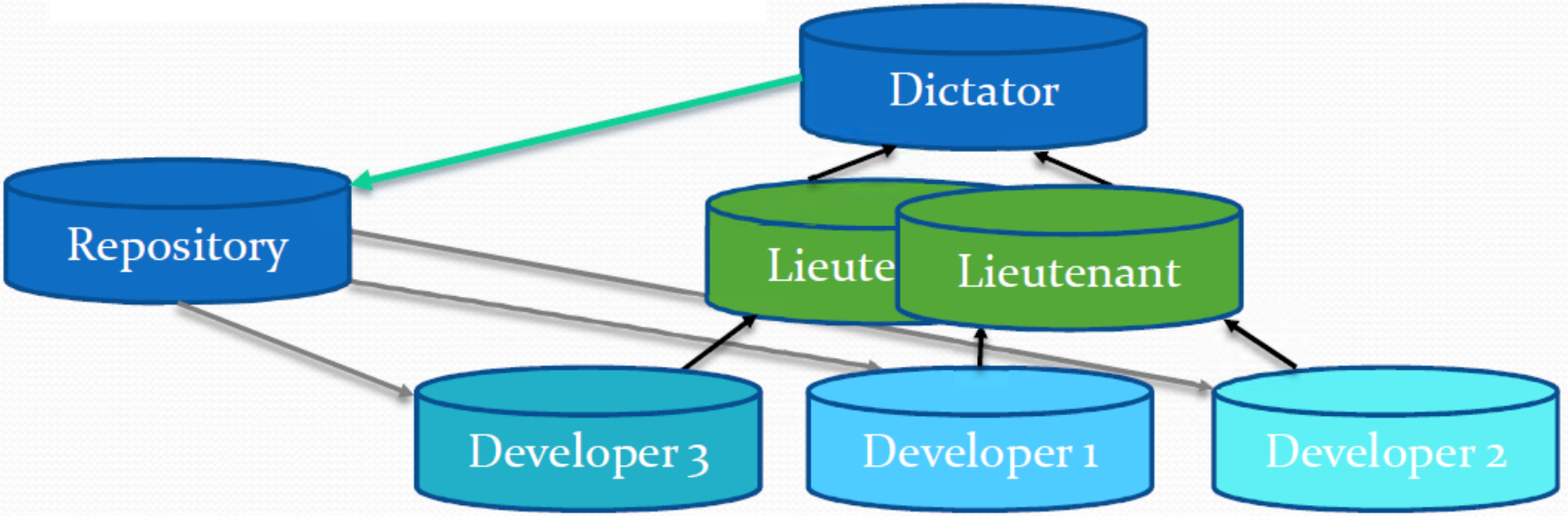
Integration Manager Workflow

- Clone from repository
- Developers push to public copy
- Developer sends email to integrator
- Integrator pulls from public
- Integrator pushes to repository



Benevolent Dictator Workflow

- Developers commit to topic branch - rebase on top of their master
- Lieutenants merge the developers' topic branches to their master branch
- The dictator merges the lieutenants' master branches into the dictator's master branch
- The dictator pushes their master to the reference repository so the other developers can rebase on it

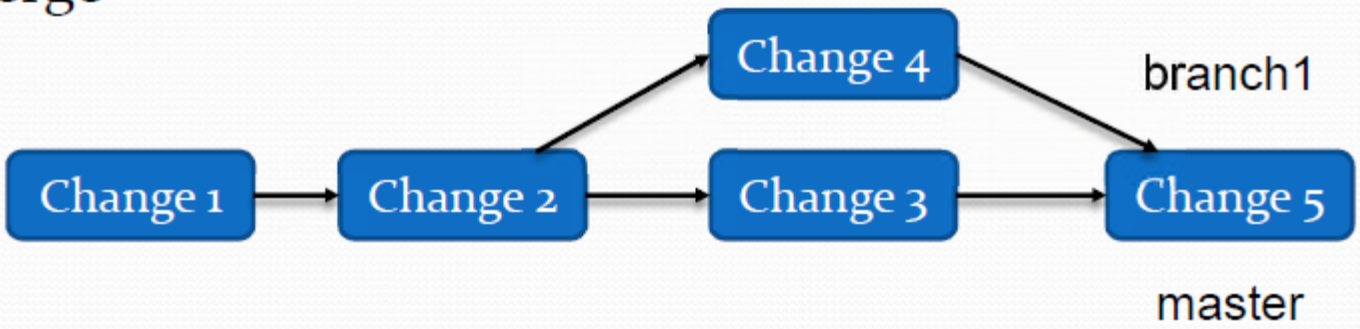


Rebase

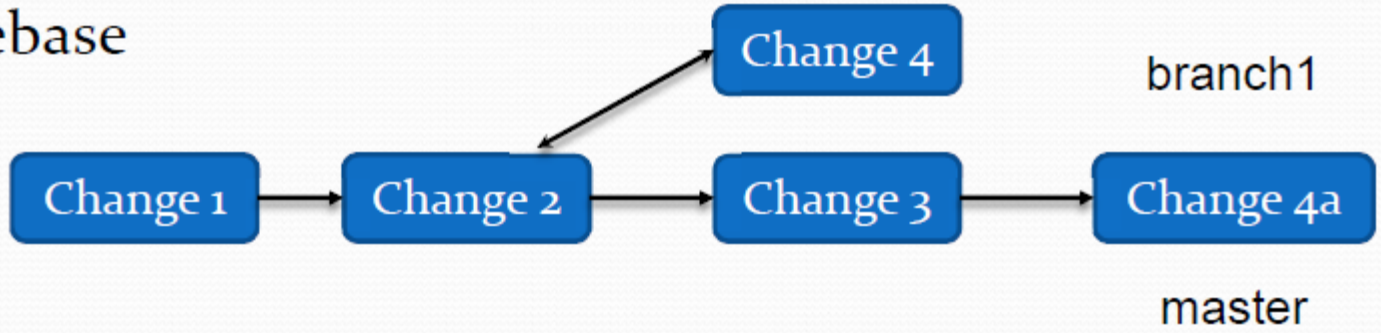
- `git rebase branch otherbranch`
  - Options can specify merge strategies
  - Can also do interactive **-i**
- Apply changes from one branch into another
- Same results as merge – different history
- Different from merge – not three-way
  - **Rolls** changes back to the common ancestor
  - Applies all the diffs from source branch to target branch
- Next, perform fast-forward merge to complete the operation
- Don't rebase commits outside your repository

Rebase Changes

- Merge



- Rebase



Interactive Rebase

- Pops up an editor with list of commits
- Move commit order to desired order
- Change pick to edit allows for editing files at this point
- **`git rebase --continue`** when finished editing
- Useful when changing the history to be more logical
- Add **exec** steps to test intermediate changes

Dry Runs

- Many Git operations have **--dry run** option
- Doesn't actually perform the operation
- Reports on what would happen
- Reports what will be committed or not and tracked or untracked
- Use **--reset** to go back to the beginning
- Report can be the *short*, *long* or *porcelain* format

How to Set Up Collaboration

- Create a central repository
  - GitHub – log in and create a project
  - Gerrit – log in and create a project
  - GitLab – log in and create a project
  - File system based
    - Open a command window
    - Change to the repository location – create if needed
    - Execute **git init -- bare #** create a bare repository for sharing
- Obtain the URL from repository owner
- Clone the repository into the local files system
- Determine which branch to switch to
- Begin development

Clone Submodules

- Separate repository logically linked to current repository
- Adding a submodule creates or adds a line to **.gitmodules**
  - When cloned, initially empty
    - If you know there are submodules in a repository clone with **--recursive**

Multiple Remotes

- Tell Git how to match a local repository reference to a remote repository
- Added using **git remote add**
- Takes a fully qualified URL to the repository

  `gituser@somerep.com:schacon/simple.git`
- Pushes and pulls will reference the remote
- **Refspecs** can be defined in the **config**
  - Reference the URL
  - Fetch references **local:remote**
  - Push references **local:remote**

Share Patches with Others

- Exchange differences via email using a diff format
  - One per file
  - One big file
- Can be text or binary differences
- Designed to allow developers without access to repository to distribute changes
- Produced using **format-patch**
- Applied using **am** or **apply**
- Can use patience algorithm to find **diffs**
- Diffs can be lots of smaller deltas or one single delta
- Produced in UNIX mailbox format

Delivering Archived Code

- Archiving is like export of a version
- Options to control format and content
- Default attributes come from **.gitattributes**
- Default is **tar** format
- Specify the path to archive and output file name
- Can specify remote instead of local for the source
- Examples:

```
git archive --prefix=v1.0/ -o proj_v1.0.tar.gz v1.0
git archive --format=tar.gz HEAD >proj.tar.gz
```

Poll

Which gets the latest content and merges if necessary?

A. fetch
B. push
C. pull
D. rebase

Poll

What command is used to create a diff file to be sent via email?

A. archive
B. format-patch
C. apply
D. am

Summary

- Several ways to move changes from branch to branch
- Rebase is an alternative to merge that can create a different commit history
- Using a **dry run** is very helpful to determine the likelihood of success

Course Wrap-up

Course Evaluations

- If you haven't already completed the course evaluation, please give us feedback regarding your impressions of this workshop:
  - Instructor
  - Materials
  - Equipment and facilities
  - or Virtual Experience

- We appreciate your feedback. Thank you!