Git in and Out

Why the ins?

- Git is a very simple System at the core
- Understanding the core makes us better understand how to use it
- The low level git commands handle the core
- The high level Git commands implement specific workflows with the object store
  - Useful to control versioning
  - Useful to control cooperating developers
- Conceivably, those same low level commands could be combined differently

Why the outs?

- High level commands implement very useful versión control paradigm
- These are the commands you Will use 99% of the time
  - But only 20% of them
  - The rest you will need to use very less often
- WE ARE NOT GOING TO PRESENT ALL THE “OUTS”
  - Just the 20%
  - You can understand the rest if you understand this content.
    - Reading their docs, of course.
      - Hey, you could even write your own scripts using just the “ins”

The Object Store

The core repo basis

Object Store: Key-value store

- Base Core concept: you MUST understand it
  - The rest can be derived from the structure and techniques used in the store
- An Object is an IMMUTABLE array of bytes
- An object has a unique ID
  - Sha1 produces unique Ids
    - Unique: as Sha1 is a cryptographic HASH function
    - ID: an object is immutable ➔ its content does not change ➔ its Sha1 does not change
- Ids of objects are used to index them

Key-value object store

- Basic command to produce the Sha1 of an object stored in a file
  - git hash-object <file>
    - Returns the hash value of the file's content (an array of bytes encoding some information)
  - You can also provide the object through stdin:
    - echo "hola mundo" | git hash-object –stdin
      - a762db79411f6e7bda2cd280c8ae805eebd780ba
      - Hex digit string representation of the hash value
- Basic command to store an object in the object store
  - git hash-object <file> -w
  - Objects are stored as files, whose names "are" their IDs

Key-value object store

- Reading an object's content
  - git cat-file 644cb46af691d3a74d2d1e3fd202ff164378d4
  - ➔ hola mundo
- Removing an object from the store:
  - … no command for that…
  - Because…
    - Virtually, git's object store contains ALL possible objects
    - In actuality, we will see that unused objects are garbage collected
    - Need a way to determine which objects are "used"/"relevant"

Aside: object store structure

- Objects are stored as files
- Every object file's name is its ID minus the first two hex digits
  - E.g. the name of the "hola mundo" object file is
    - 62db79411f6e7bda2cd280c8ae805eebd780ba
- An object file is stored in a folder whose name is the first two digits of the object ID
  - E.g. the name of the folder where file 62db79411f6e7bda2cd280c8ae805eebd780ba is stored is
    - a7
- All object folders are stored within a folder, the objects folder

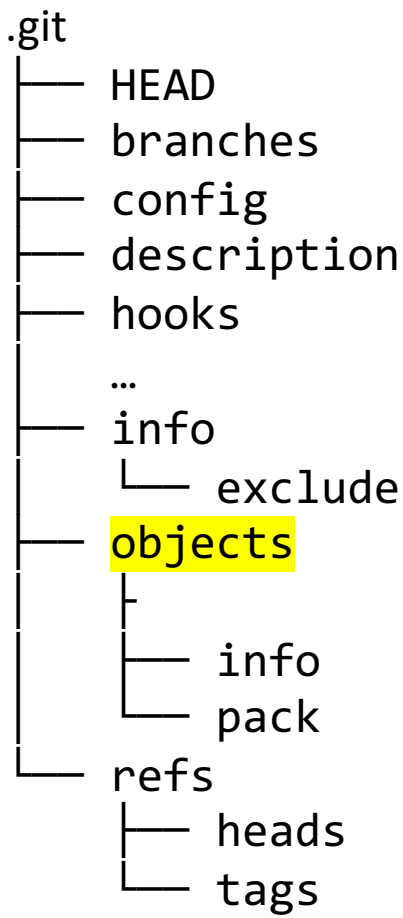
Object store structure

- echo “hola mundo” | git hash-object –stdin –w
  - a762db79411f6e7bda2cd280c8ae805eebd780ba
- echo “hola mundo cruel” | git hash-object –stdin –w
  - 30644cb46af691d3a74d2d1e3fd202ff164378d4
- echo “adios mundo cruel” | git hash-object –stdin –w
  - 5389c17b66fa6c03b871ebe50a705b42b1c8440d

```
objects
├── 30
│   └── 644cb46af691d3a74d2d1e3fd202ff164378d4
├── 53
│   └── 89c17b66fa6c03b871ebe50a705b42b1c8440d
├── a7
│   └── 62db79411f6e7bda2cd280c8ae805eebd780ba
```

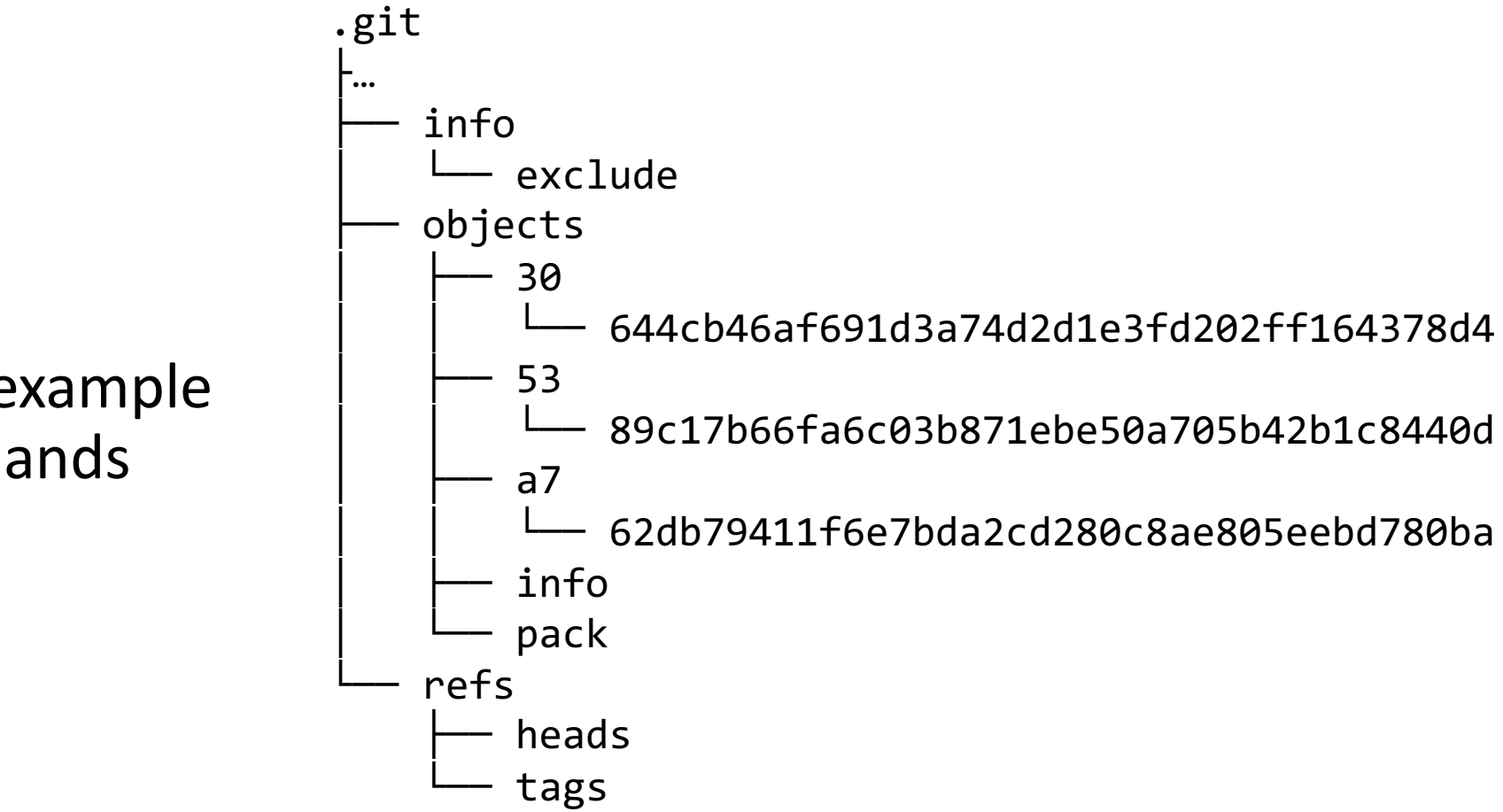
A Git Repo

- Go to an empty folder
- Run: `git init`
- This creates a `.git` folder with these contents
  - There we find our *objects* folder
    - Initially empty
    - It contains the key-value store with the objects of the repo
    - All git hash-object –w créate objects within the *objects* folder as explained



A Git Repo example

- After running our example hash-object commands



Exercises

- Write a script that implements git cat-file <hash>
- Write a script that implements the unimplemented feature in git
  - delete-file <hash> , which deletes the object whose hash is given

What does a Git Repo represent?

- A History of changes to a set of files organized hierarchicaly within folders
  - Each “step” of the history is referred to as a “commit”
  - Each step is an immutable representation of the state of the Repo at some point in the past
- How is this history represented/stored
  - Using the object store
  - Using several kinds of objects
    - blob
    - tree
    - Commit

Blob

- Uninterpreted arrays of bytes
  - We have already seen them in our examples…
    - *Uninterpreted* is the key Word
      - Git does not inspect the contents of a blob.
- Git uses them to store the files we edit in our repo
  - Code Files
  - Arbitrary text files
  - Even binary files

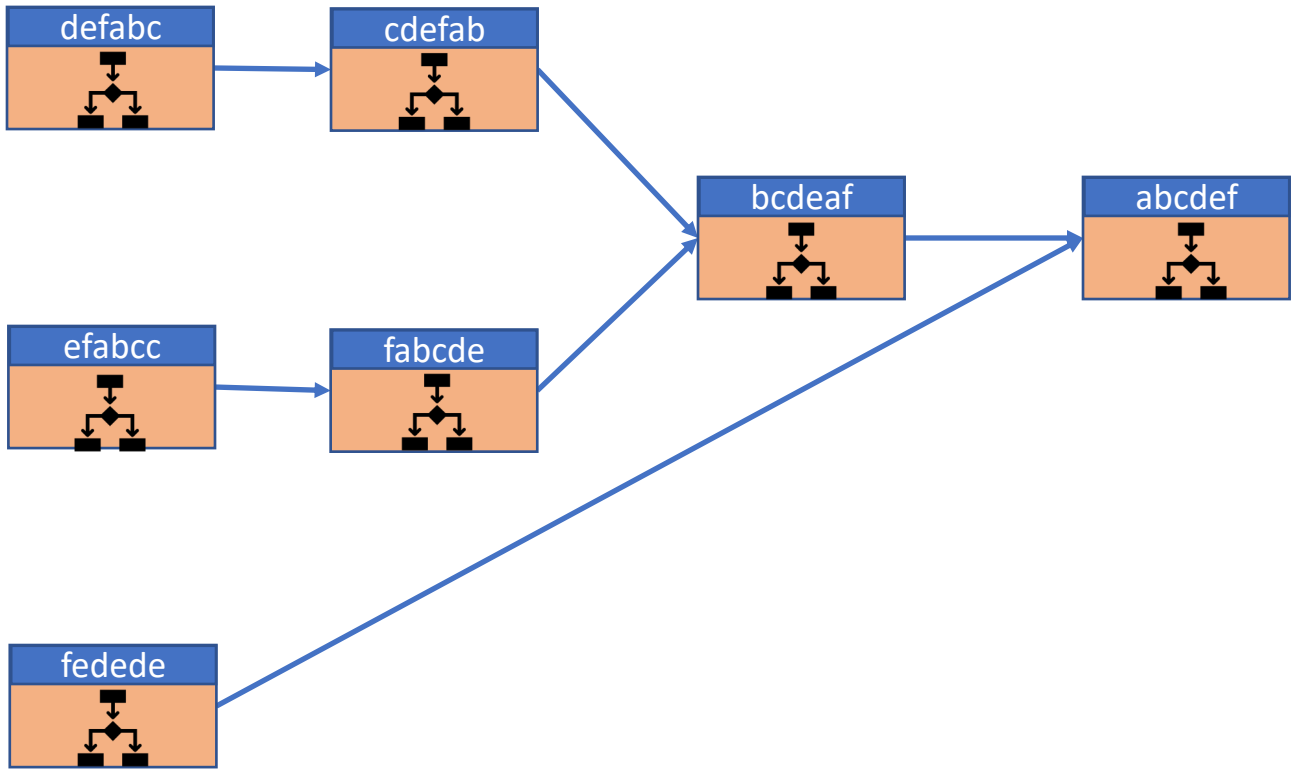
Tree

- Structured object representing a folder
- It has the following structure
  - <hash_1> name_1
  - <hash_2> name_2
  - …
  - <hash_n> name_n
- Each name is the name of either a file or a folder within the folder represented by the *tree*.
  - When *namei* is the name of a folder, <*hashi*> is the id of the tree object representing its contents
  - When *namei* is the name of a file, <*hashi*> is the id of a blob object with the contents of the file
- Trees are used to represent hierarchical file system structures
  - Like the directory containing all files of a project

Commit

- A commit is a structured object representing a snapshot of a hierarchy of files/directories within a history of other snapshots
  - Snapshot === immutable “picture”
- It mainly contains descriptive metadata (comments, descriptions, authors) and “pointers”
  - To the parent commit
    - History line…
    - Forms a linked list of commits
    - It can have more tan one (e.g., merges)
  - To the tree object representing the snapshot (immutable) directory structure
- Commit DAG through the parent pointers

Possible commit DAG in a repo



Arrows represent parent relationships

References

Basic working element

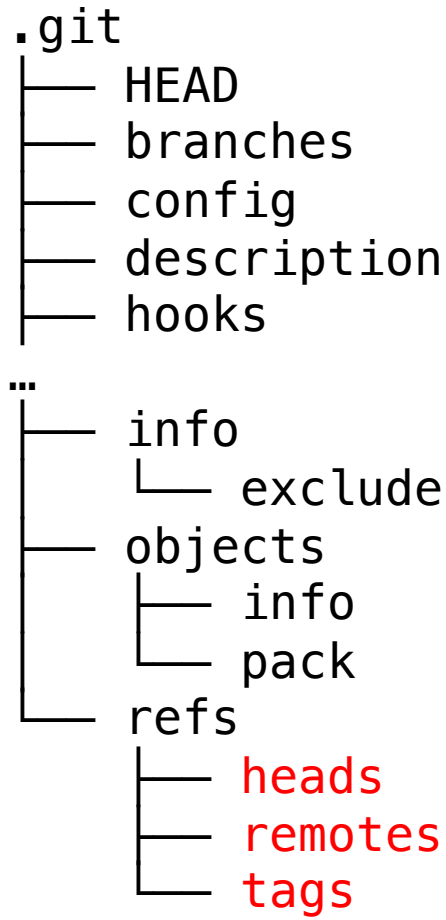
References

- Day to day use of a git repo implies working mainly with commit objects
- We can refer to commit objects using their IDs (the SHA1 digests)
  - This is highly cumbersome
  - Also error-prone
  - Additionally: how do we identify the “current” snapshot, i.e., commit?
- To help referring to relevant commits git implements commit references (or just *refs*)

Refs

- Three *normal* kinds:
  - *Heads* (aka *branches*)
  - Remotes
  - Tags
- Represented as text files
  - The name of the file is the name of the ref
  - The content of the file is the HEX representation of the commit object ID (a 40 digit SHA1)
- Files stored within their own directories
- Special refs:
  - The main being HEAD
    - In .git/HEAD
    - Contains either a commit SHA1
    - …or “ref: <refpath>”… a pointer to another ref

Refs directories



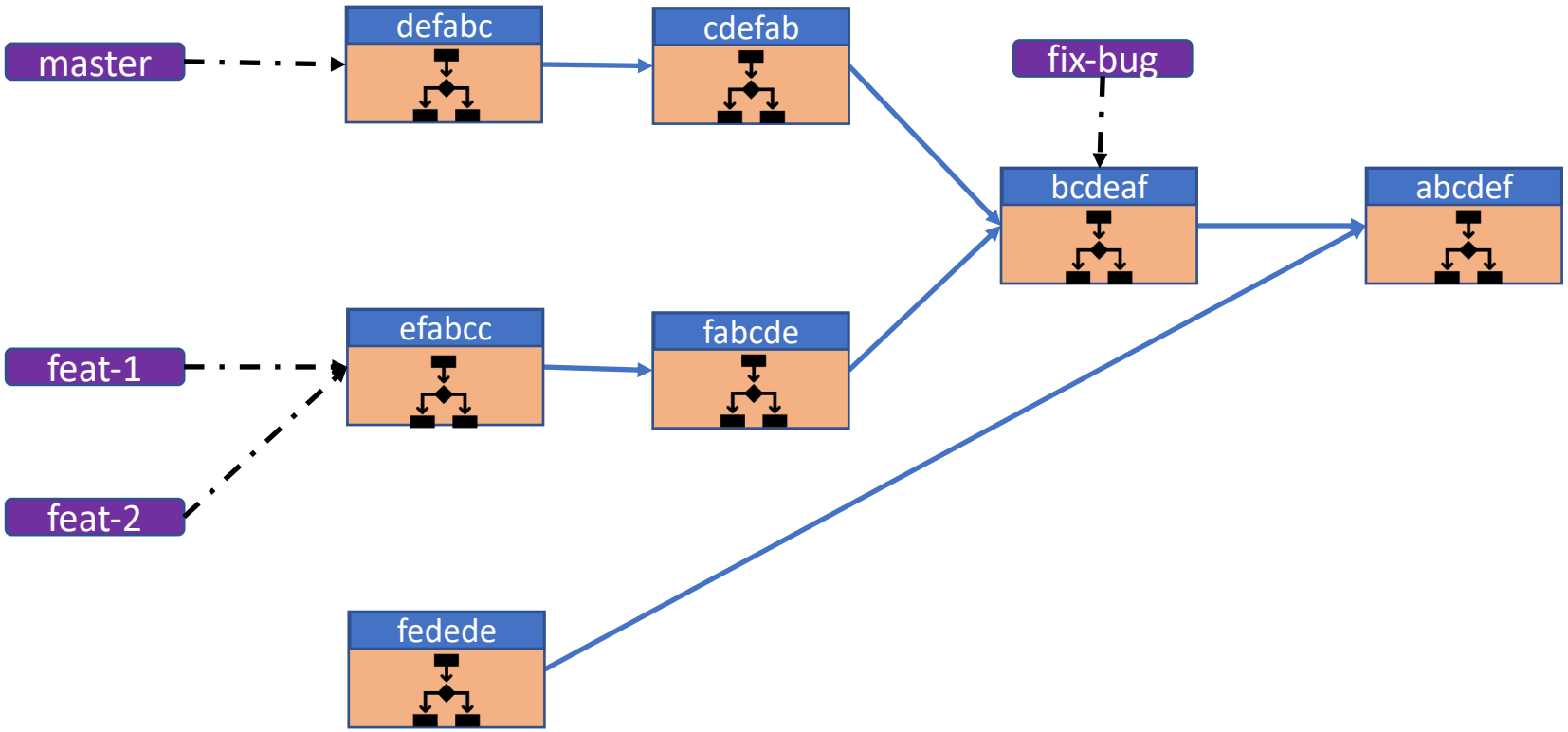
Ref expresions

- Ref file names
  - Paths to avoid ambiguity
- Relative Refs
  - Used to traverse the history from a commit
- Let R be a ref expression (refers to a commit)
  - R~1 is its parent commit
  - R~2 is its grandparent commit
  - …
  - If R has more than one parent
    - R^1 is its first parent commit
    - R^2 is its second parent commit
    - …
    - R^1^1 is the first parent of its first parent commit
    - …

Branches

- Branches are sub-dags of the repo commit DAG
- They are built by ***head*** refs, pointing to the “top” commit in the Branch
  - The rest of the commits in the sub-dag are obtained following the *parent* links
  - There is no commit in a Branch pointing to the top commit through its parent link
- One commit can belong to multiple branches
- The *top* commit of a Branch can be the parent of other commits out of the branch
- Branches are mutable: they change as new commits are added
  - To track history

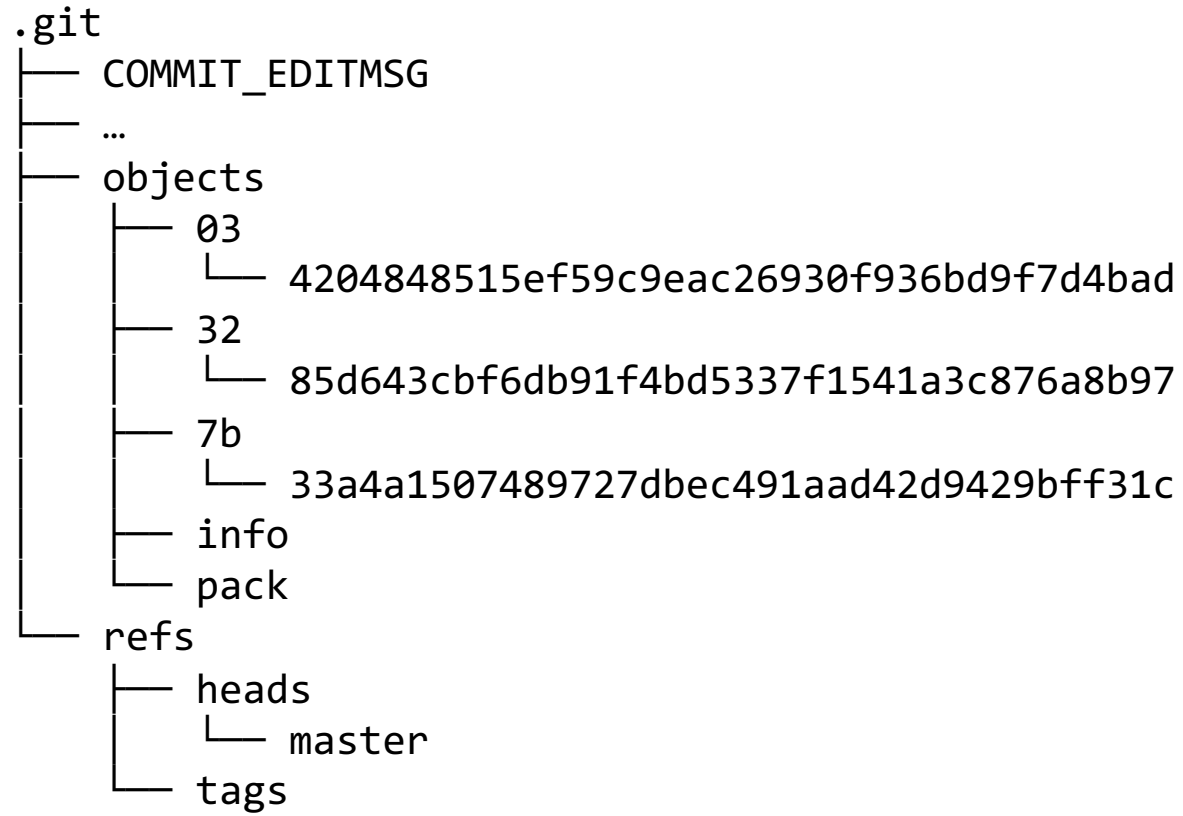
Branches


master
defabc
cdefab
fix-bug
bcdeaf
abcdef
feat-1
efabcc
fabcde
feat-2
fedede

How are branches stored?

- Within .git/refs/heads
  - Each file is a branch name
  - Each file contains a commit object ID
    - The “top” commit of the Branch represented by the file
- cat .git/refs/heads

  034204848515ef59c9eac26930f936bd9f7d4bad



Tags

- Similar to branches
  - But stored within .git/refs/tags
- They focus on representing a particular commit
  - Specifically, the snapshot the commit represents
- Used to identify/name specific, relevant versions of the snapshotted folder.
- They are IMMUTABLE a tag always identifies the same commit
  - No commands exist to support workflows altering a TAG

Remotes

- A Remote is another repo somewhat related to this one
- A Remote ref represents a branch of the remote repo whose objects have been brought into this repo
  - The objects reachable from the commit identified by the branch
- Remote Refs are stored within folders with the name of the remote repos. E.g. for the **origin** remote:
  - .git/refs/remotes/**origin**/master
- Multiple remotes can be tracked with these references
  - This is used by the syncing mechanisms with remote repos

Working with a repo

The three trees

The HEAD

- As mentioned it is a special Ref
- Represents the “current” commit on top of which we are working on a Repo
  - Thus, it represents the “current" **tree** snapshot of our repo
    - In reality, it intends to represent a particular **branch** of the history
      - The previous commits reachable from the parent pointers starting from HEAD
- SO, IN SUM, IT REPRESENTS A TREE of files/folders
  - The last one we checked-in and ON TOP OF WHICH WE ARE WORKING

The WORKING DIRECTORY

- If we are working on top of the HEAD tree…
- … where are the files/folders that we can actually manipulate?
- ➔ in the WORKING DIRECTORY

- Representation on the file System of the tree of a commit
  - We modify the files/folders in this working directory
  - Starting from the tree of a commit
    - The HEAD
  - Finally creating a new commit and substituting it as the new HEAD
    - After linking it with the previous HEAD as its parent
  - NOTE: This is the common WORKFLOW
- Typically in the folder containing the .git folder

Forming Commits: The INDEX

- Two trees so far:
  - The COMMIT tree, pointed to by HEAD, kept in the object store
  - The WORKING DIRECTORY tree, on the file System
- We use a third tree representation: the INDEX
  - Stored within .git/index (not a text file, though)
  - Actual files kept in the object store
- The INDEX represents a tree as a list of PATHS to files
  - Each item contains the SHA1 of the file blob
    - Plus some extra metadata, not relevant to our main discussion
- When ready to make a new commit, the tree of this commit is the tree represented in the INDEX
  - If equal to the HEAD's tree, no new commit is created

Working with the INDEX: Normal flow

- Initially all three TREES represent the same folder/file structure
  - And all files are captured in the object store as blobs, with their SHA1 ID
- Then
  - We alter some file in the WORKING DIRECTORY
  - We modify the INDEX to include this modification in the next commit
  - We repeat the above as many times as we consider adequate to form the new commit
  - We actually create the new commit, whose tree will be the one represented in the INDEX

Main commands on the INDEX: add

- Command to add a modified file in the WORKING DIRECTORY to the INDEX:
  - `git add <path-to-file>*`
- It creates the blob objects into the object store
- `<path-to-file>` can have globs
  - E.g. `git add *.py`
- If `<path-to-file>` is a directory, it includes all of its subfiles
  - E.g. `git add .`
    - Makes the INDEX equal the Working Directory

Main Commands on the INDEX: checkout

- Command to set all three TREES to the same value
  - `git checkout <Ref>`
  - The HEAD becomes equal to Ref
    - If Ref is not a branch, the HEAD is “detached”: contains the ID of the Ref
    - If Ref is a branch, HEAD =
      - ref: <Ref>
        - E.d. ref: refs/heads/master
      - That is, it is itself a ref
  - The WORKING TREE contains the file-System representation of the HEAD’s tree
  - The INDEX contains the path-list representation of the HEAD’s tree
  - NOTE: this command should be issued when all three trees are equal
    - If it detects non-commited changesit fails, unless forced to proceed (-f)

Main Commands on the INDEX: reset

- Various versions
- `git reset --<mode> <Ref>`
  - All modes set HEAD to <Ref>
    - If HEAD is brnach, it actually sets the Branch to Ref too
  - --soft
    - Does nothing more
  - --mixed
    - Sets the INDEX to the <Ref> tree
  - --hard
    - Sets the INDEX and the WORKING DIRECTORY to the <Ref> tree

Main Commands on the INDEX: reset

- `git reset  <Ref> <paths>`
  - Set the INDEX for <paths> to their value in <Ref>
  - When <Ref> == HEAD, it is the opposite of git add <paths>
  - Default <Ref> is HEAD
- Ref can actually be anything that can resolve to a tree object
- This form does not touch the working directory

Main Commands on the INDEX: commit

- `git commit -m <commit message>`
- Creates a new commit object C
  - C's tree is the current INDEX tree
    - It may need to créate intermediate tree objects for subfolders containing modified or new files
  - If HEAD is detached
    - The new commit's parent is set to the HEAD's commit
    - The HEAD is set to the new commit
  - If HEAD points to a Branch B
    - The new commit's parent is set to B's commit
    - B is set to the new commit
- At the end, the HEAD's tree and the INDEX tree are the same
- In reality, the INDEX is not altered

Branching workflow

A basis for colaborative work

Branching: Simplistic workflow

- With Git we can implement several workflows
- Simplistic workflow, maybe valid for single developer
- Only the master Branch is used
- Each feature set Will receive a versión number
  - NOTE that we advise using semantic versioning
- Work procedes through several commits on the master branch
  - Edit →Add → Commit → Edit …
- When the feature set is properly reached
  - A tag Ref is created with the versión name (e.g., v1.0.0)

Branching: feature branches

- The master Branch tracks the current line of development
  - Well tested
  - Targetting a particular feature set
  - Each feature having its own complexity
- When a new feature needs to be implemented, a new branch is started
  - Work proceeds on this new Branch
  - The master branch is not disturbed by this work
  - When finished, the feature branch is TESTED
  - When tests pass the feature branch is merged into the master branch
  - A tag is created to represent the new versión
    - Typically, with patch number update…
- When the master Branch reaches feature completeness, it is tagged with a versión number
- Then the cycle repeats again

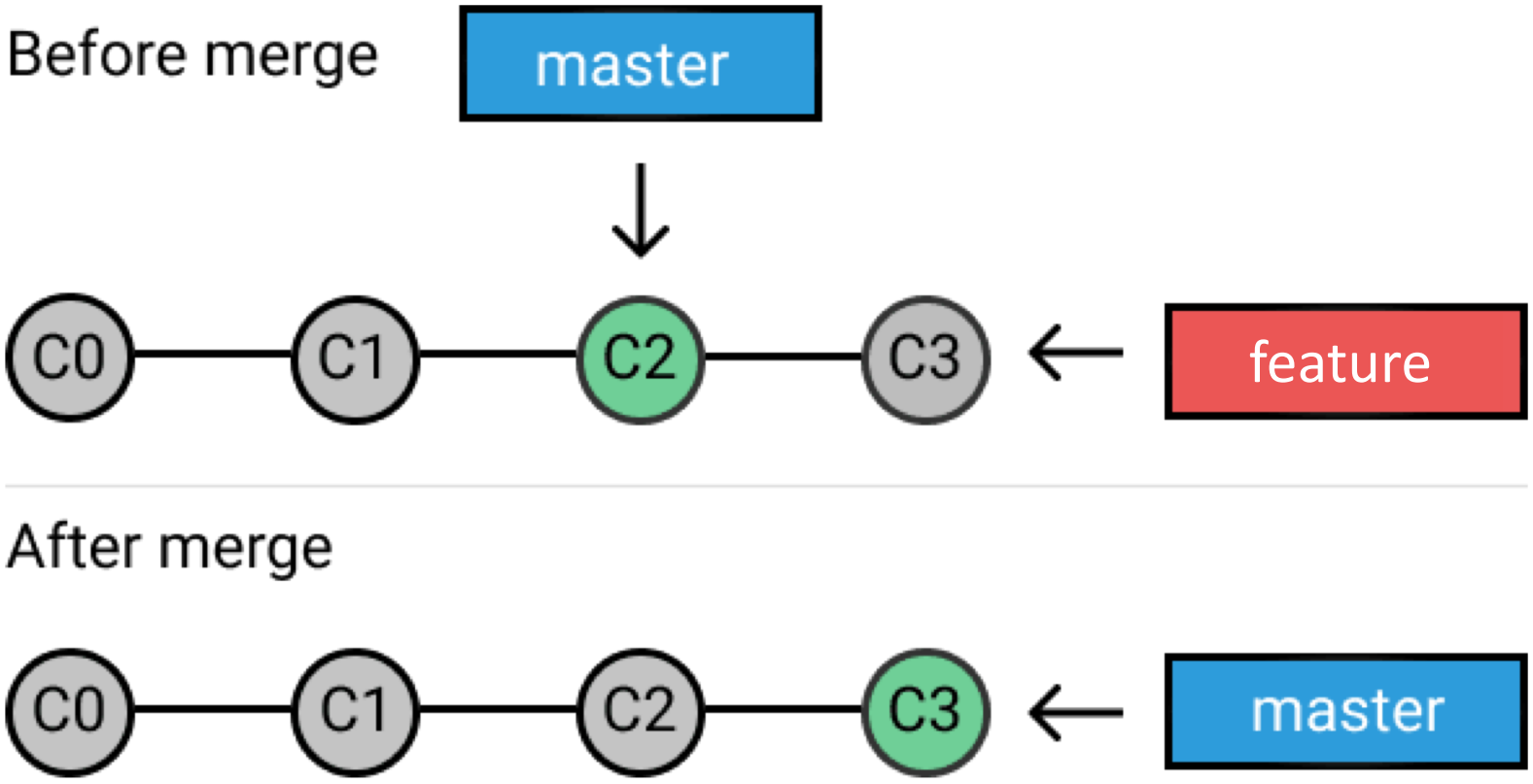
Branching: bug fix branches

- Very similar to feature branches
  - But motivated by a defect discovered after releasing a new versión
  - Generally with some urgency/higher priority over new features
  - Generally, proceeding in parallel to new feature branches
- When a bug needs fixing, a new Branch is started
  - Work proceeds on this new Branch
  - The master Branch is not disturbed meanwhile
  - When finished, the bugfix Branch is TESTED
  - When tests pass the bugfix Branch is merged into the master Branch
  - A tag is created to represent the new versión
    - Typically, with patch number update… representing the bug fix wrt the versión released

Branching: merging

- In both branching scenarios, we need to merge branches
  - In our scenarios, some branch onto the master branch
  - But could be arbitrarily complex: any branch onto any other branch
- Assume you want to merge branch ***feature*** onto branch ***master***
  1. `git checkout master`
  2. `git merge feature`
- Two possibilities for the merge
  - Fast-Forward
  - Three-Way

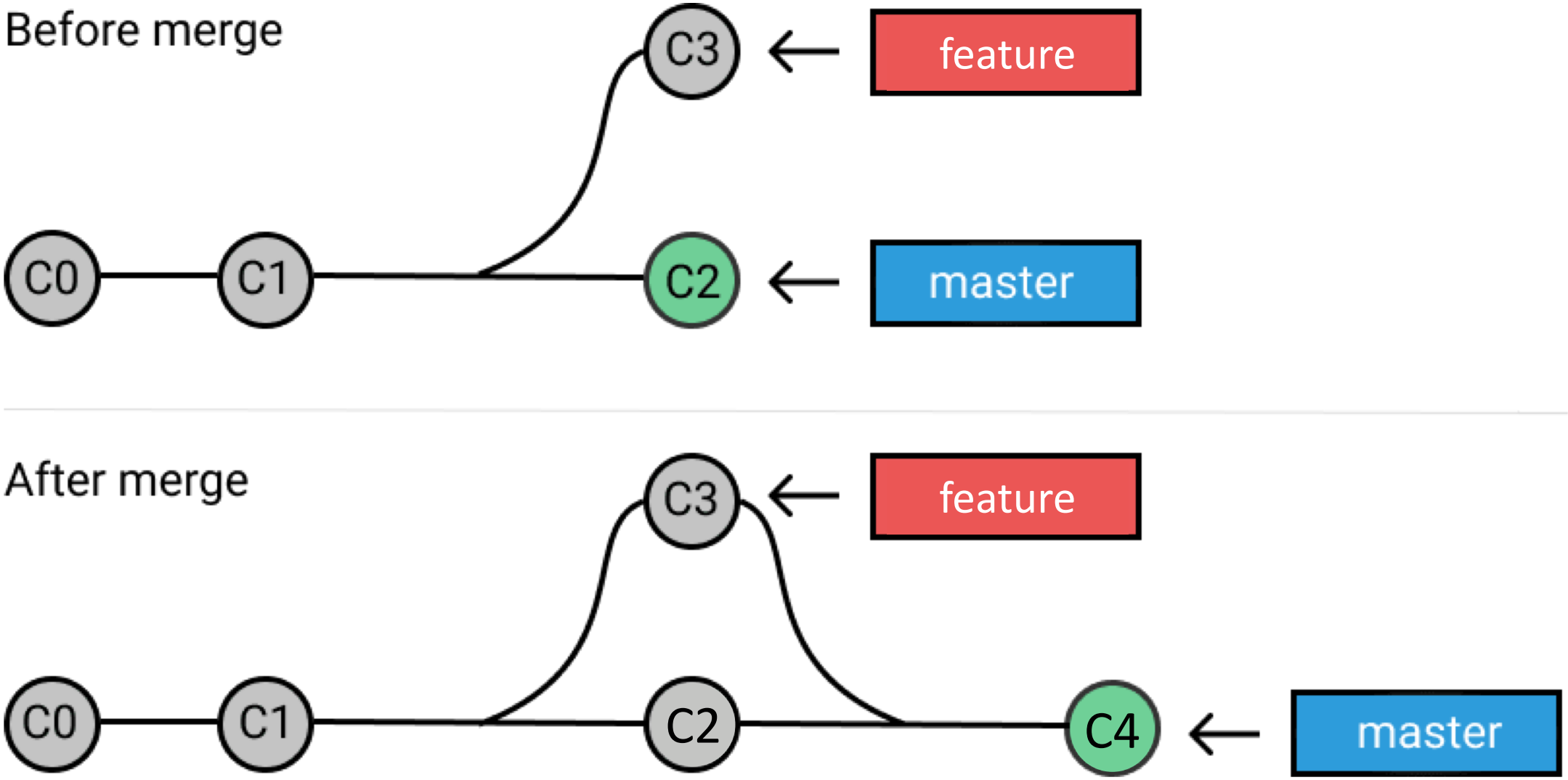
Branching: Fast-Forward merges



Branching: Fast-Forward merges

- No conflicts between master and feature
  - Master is an ancestor of feature
  - No divergence allowed to proceed this way
  - No new commit is created
  - No additional actions need to be performed by the developer

Branching: Three-Way merges

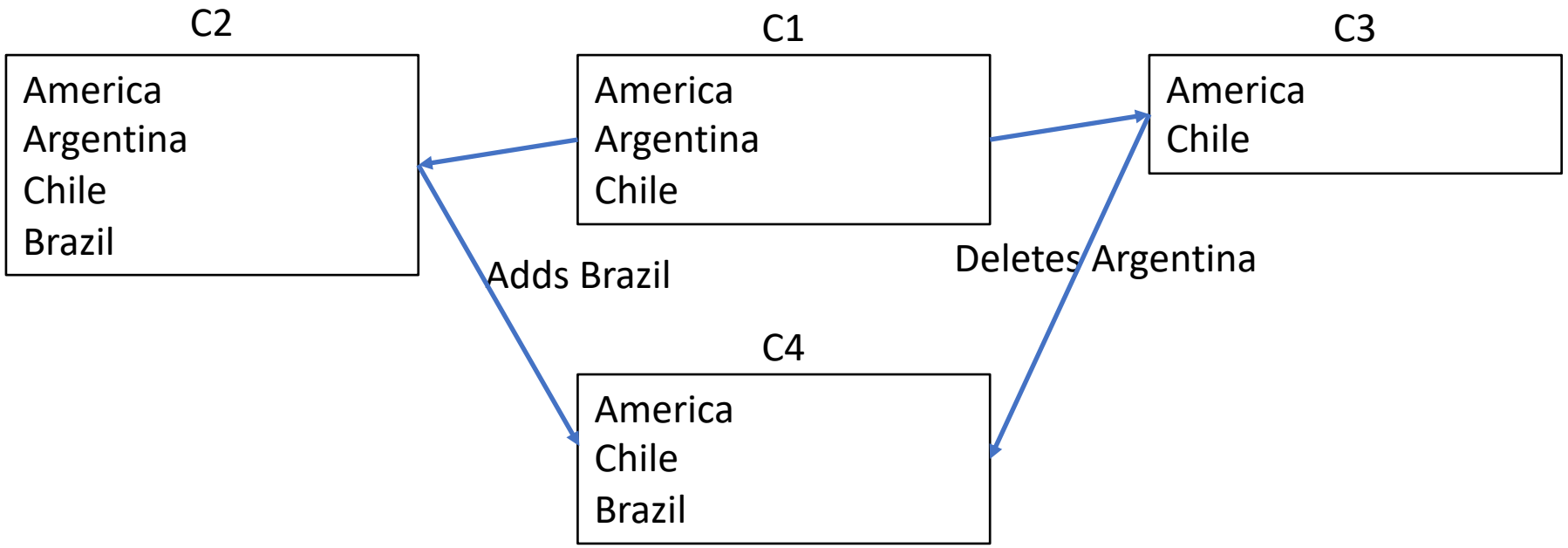


Branching: Three-Way merges

- A new commit is created
  - It has two parent commits
    - The first from the merged-into branch (C2 at master in our example)
    - The second from the merged branch (C3 at feature in our example)
- GIT performs calculations to find possible merge conflicts
  - Applies Three-way-merge algorithm
    - For each file, F, in C3
      - Git may find conflicts with F in C2
        - It may be able to resolve them
        - OR it may need the developer to resolve the conflict
    - If the developer had to resolve no conflicts, GIT creates C4 with the merged files
    - Else, the developer has to create the commit in the standard way
      - Using git add, to build the index
      - Using git commit, to finally create the commit

Three-way merge algorithm

- E.g.: feature vs master branches that have diverged in file F
- Uses common ancestor C1:F, and compares C2:F and C3:F to it
  - Figures out changes
  - If no conflict, it merges

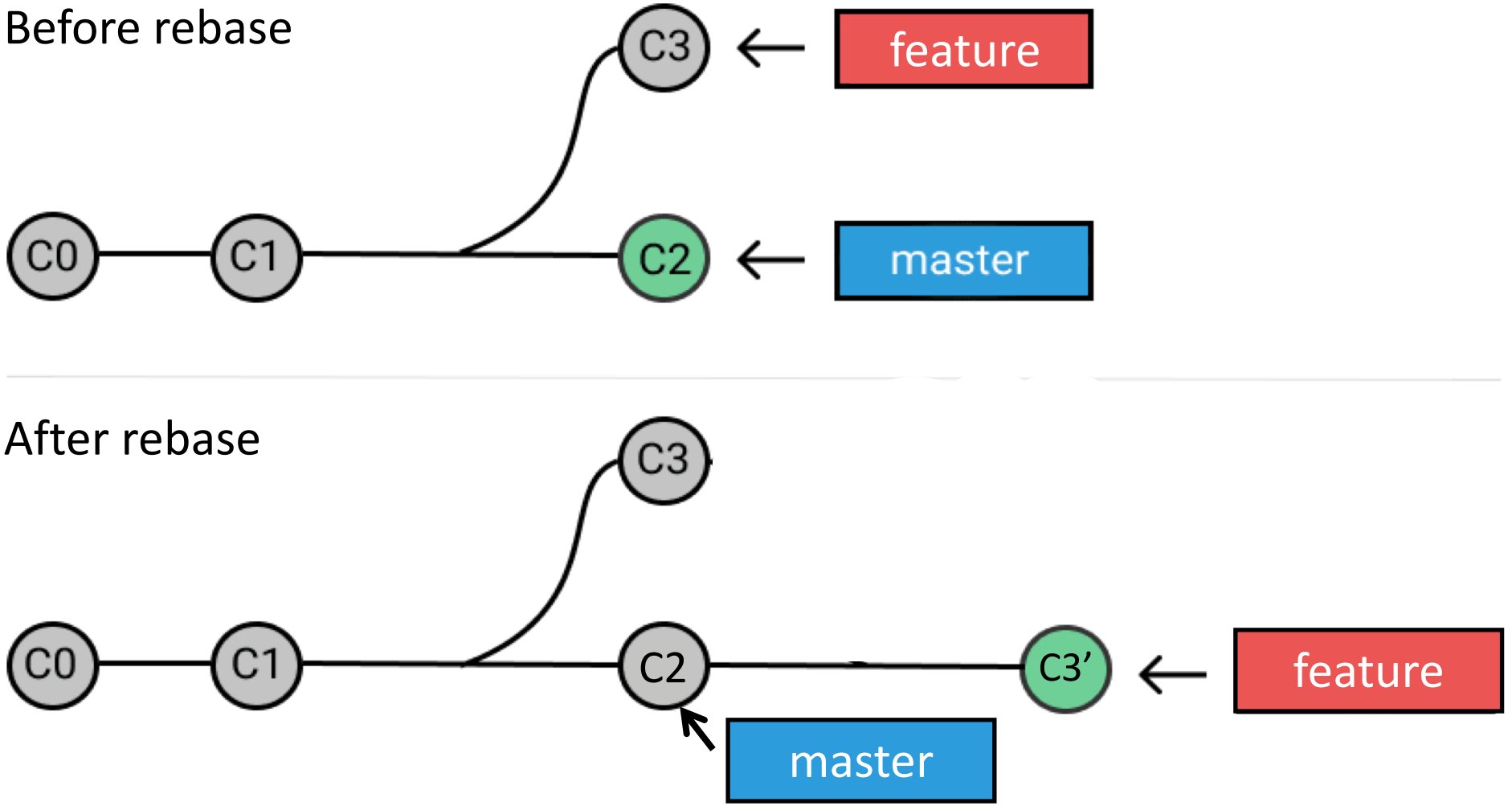


Merge Conflicts

Branching: Rebasing

- Merging creates complex DAGs
  - New commit with two parents
- When done frequently, it can lead to complex histories
  - Difficult to follow
- Rebasing helps keep a linear history
  - Avoids complexity of DAG

Branching: Simple rebase



Branching: Rebasing

- Rebasing is another technique to merge two divergent branches
  - By rewriting history
- In our example, we rebase C3 from `feature` to be parented by C2 from `master`
  - Actually, not C3, but a new commit, C3', applying the same changes to C2 as C3 does to C1
- Commands:
  - `git checkout feature`
  - `git rebase master`
- We may need to resolve conflicts, as with three-way merges

Branching: Commit squashing

- Rebasing helps keep histories clean
- Often we make lots of silly/simple commits
  - They contribute nothing to understand what is going on
  - They just keep track of ongoing work, that cannot be completed in one seating (for example)
- Before we publish our commit we may want to “clean up” the history
  - We can use commit squashing.
  - Several techniques. E.g. If we want to squash the last 3 commits
    - `git reset --soft HEAD~3`
    - `git commit -m “my new squashed commit”`

Scenario1: From 0. Repo Initialization

- Create a folder where you want to place your work
  - E.g., ”project”
    - `mkdir project`
      - This will become the WORKING DIRECTORY of the repo, which is empty
- Go into that folder:
  - `cd project`
- Create and initialize the git repo
  - `git init`
    - Creates the repo within the `.git` subfolder of `project`
    - NOTE: the .git subfolder itself is not part of the WORKING DIRECTORY
  - The HEAD’s contents are
    - `ref: refs/heads/master`
    - But the folder .git/refs/heads is empty
    - So HEAD is pointing to no commit: The HEAD tree is empty
  - The INDEX tree is empty, like the WORKING DIRECTORY

Scenario 1: From 0. Content creation

- Create a file and commit it
  - `touch README`
  - `git **add** README`
    - This creates a blob object with the contents of README
    - Tracks this object in the **INDEX**
  - `git **commit** –m “first commit”`
    - Creates tree objects for the subfolders that appear in the index (NONE)
      - Containing any changed files
    - Creates a tree object for the contents of the INDEX tree
      - Pointing to files (README)
      - Pointing to subfolder trees created (NONE)
    - Creates a commit object pointing to the top level tree object just created
      - With no parent, as the master points nowhere
      - Creates the master file for the master branch and points it to the commit
        - `.git/refs/heads/master`

Scenario 2: Existing content

- We have a “project” folder with our files
  - It will be the WORKING DIRECTORY of our repo
- `cd project`
  - `git **init**`
    - As in scenario 1, this creates the actual git repo within the .git subfolder of project
  - `git **add .**`
    - Tracks all contents of the working directory in the INDEX
      - Creates all blob objects for all files in the working directory
      - INDEX tree == WORKING DIRECTORY tree
  - `git **commit** –m “first commit”`
    - Creates tree objects for the subfolders that appear in the index
      - All fileas are new
    - Creates a tree object for the contents of the INDEX tree
      - Pointing to top level files
      - Pointing to tree objects created for top level subfolders
    - Creates a commit object pointing to the top level tree object just created
      - With no parent, as the master points nowhere
      - Creates the master file for the master branch and points it to the commit
        - `.git/refs/heads/master`
    - All three trees are equal

Scenario 3: Remote content in github

- We have a github repo
  - ***username***
  - ***myprojectname***
- We want to work with it in our computer
- `git clone` `git@github.com:username/myprojectname.git`
  - Uses ssh protocol to access the server
    - Need to register public ssh key in the server if repo is private
- `git clone` `https://github.com/username/myprojectname.git`
  - Uses https protocol to access server
    - Can request interactively the password to access the repo if private
- In both cases
  - A folder "`myprojectname`" is created
  - `myprojectname` is the WORKING DIRECTORY
  - `myprojectname/.git` contains the actual git repo

Scenario 3: Remote content in gitlab

- We have a gitlab repo (or project in gitlab's lingo)
  - ***hierarchy*** (NOTE: gitlab allows a hierarchy of groups to contain projects)
    - ***group/subgroup1/subgroup2/…*** OR ***username***
  - ***myprojectname***
- We want to work with it in our computer
- `git clone git@gitlab.com:hierarchy/myprojectname.git`
  - Uses ssh protocol to access the server
    - Need to register public ssh key in the server if repo is private
- `git clone https://gitlab.com/hierarchy/myprojectname.git`
  - Uses https protocol to access server
    - Can request interactively the password to access the repo if private
- In both cases
  - A folder "`myprojectname`" is created
  - `myprojectname` is the WORKING DIRECTORY
  - `myprojectname/.git` contains the actual git repo

Scenario 3: Fake Remote content: folder

- We have a repo in our file system
  - Maybe in a file server
  - Within <path>/myprojectname folder
- We want to work with it in another folder
- `git clone <path>/myproject`
- A new folder "`myprojectname`" is created where the command is executed
- `myprojectname` is the WORKING DIRECTORY
- `myprojectname/.git` contains the actual git repo
  - A full copy of the <path>/myprojectname/.git folder