

# Data Structures and Algorithms

## Lecture 4 — GIT

Alan P. Sexton

University of Birmingham

Spring 2019

# Revision Control with GIT

We need to be able to manage multiple versions of software

- to be able to return to earlier versions
- to manage different versions for different platforms
- to develop features/extensions in a way that doesn't endanger production code
- to support multiple developers working simultaneously on the same project
- to track changes back to the programmers who made them
- to be able to safely and reliably distribute consistent source code snapshots

Revision Control Systems provide access to **snapshots** of your files

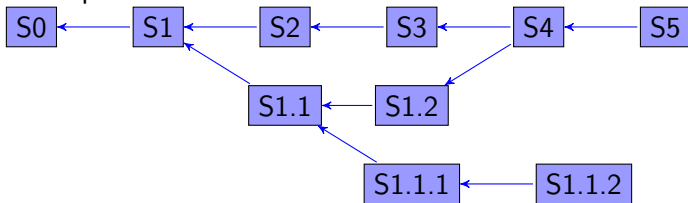
- A snapshot can be thought of as a (virtual) backup of all your source files in your project at some point in time
- The intention is that each snapshot should be
  - **consistent**: all the files together in the snapshot correspond to a version of the system at some stage of development
  - **recoverable**: if a modification of the project goes badly, one can always throw away the changes and return to a previous snapshot
  - **efficiently stored**: rather than each snapshot being stored completely independently, resulting in much wasted duplication, only one full version of the files are stored and the other snapshots are stored as change records that need to be applied to that version

# Snapshot Organization

- In a simple case, snapshots are organised into a sequence, reflecting a linear development of a project<sup>1</sup>:



- More commonly, there will be branches that diverge from the main sequence (possibly with sub-branches of their own), some of which will merge back into their parent sequence, for development of features or experimenting with possible development ideas



<sup>1</sup>arrows indicate which snapshots the current snapshot is based upon

Particular features that make GIT stand out from many other version control systems are:

- GIT deals in *system snapshots* rather than file versions
- GIT supports *Disconnected Development*
- GIT uses a *Distributed Model* rather than a *Centralised Model*

## Books:

- Scott Chacun and Ben Straub, *Pro Git*, APress, 2nd edition, July 29 2014. <https://git-scm.com/book/en/v2>
- Brent Laster, *Professional Git*, O'Reilly, 2017
- Mike McQuaid, *Git in Practice*, Manning, 2015
- Peter Savage, *Git in the Trenches*,  
<http://cbx33.github.io/gitt/>, Last accessed: Jan 2018

## Web links:

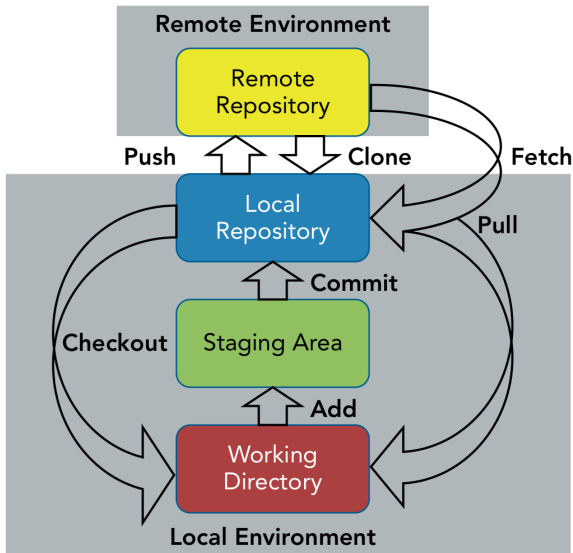
- <https://git-scm.com/>
- [https://bocoup.com/blog/  
git-workflow-walkthrough-feature-branches](https://bocoup.com/blog/git-workflow-walkthrough-feature-branches)
- [https://bocoup.com/blog/  
git-workflows-for-successful-deployment](https://bocoup.com/blog/git-workflows-for-successful-deployment)

# GIT Basics



- Remote Repository
  - A tree of snapshots (also called commits or versions) of a system
  - Cleverly implemented to minimise storage and maximise performance
  - Easiest case is a simple list of snapshots
  - Branching allows development of new features independently of the main trunk that can be merged in later
  - Accessible via a server over the internet to everyone with permissions
- Local Repository
  - A local copy of the full remote repository contents and history
  - Accessible to a single user on a single machine
- Staging Area (aka “Index” or “Cache”)
  - A place to collect sets of changes before they are committed together to make a new snapshot in the local repository
- Working Directory
  - a single snapshot that is being worked on

# GIT Commands



taken from: Brent Laster, *Professional GIT*, O'Reilly 2017

You can see what your current configuration settings are with:

```
git config -l
```

There are some basic configuration steps you need:

```
git config --global user.name "username"  
git config --global user.email "email_address"  
git config --core.editor "editor"
```

Replace username, email and editor with the appropriate values for you.

# Create your Remote Repository

- If you are using an existing remote repository (e.g. at gitlab, github etc,) you can skip this step, you just need the URL of the repository
- For some modules you may be given a specific GIT repository URL to use
- Otherwise go to <https://git.cs.bham.ac.uk/> and create an empty repository
  - **IMPORTANT: for a repository that you use for coursework, make sure you make it a PRIVATE repository unless your lecturer tells you otherwise**
- The url will be something like  
<https://git.cs.bham.ac.uk/username/myproject.git>

# Creating Your Local Repository, Method 1: Clone

If you are starting a new project, or getting a local copy of an existing remote repository:

```
git clone https://git.cs.bham.ac.uk/username/myproject.git
```

- This creates a directory called `myproject` with the contents of the remote repository
- If this is a new empty repository, then add a project description file: (use your normal text editor instead of “edit”)

```
cd myproject
edit README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

- The “-u” for this first push to set the default branch that future pulls and pushes will work with (more about branches later)
- `README.md` is a text file in which you can use Markdown formatting commands (c.f. <https://en.wikipedia.org/wiki/Markdown>)

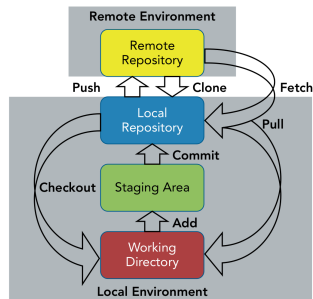
# Creating Your Local Repository, Method 2: Init

If you have an empty remote repository, and already have a local project directory with files in it, then:

```
cd existing_folder
git init
git remote add origin https://git.cs.bham.ac.uk/username/
    myproject.git
git add .
git commit
git push -u origin master
```

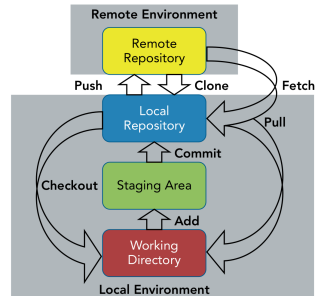
# git add: Getting files into GIT

- Adds new non-tracked files to the staging area for inclusion in the next commit and tracks them
- Adds tracked files to the staging area for inclusion in the next commit
- for example
  - `git add myfile.txt`
  - `git add mydirectory`
  - `git add *.java`
  - `git add .`



# git commit: Creating a new (local) version

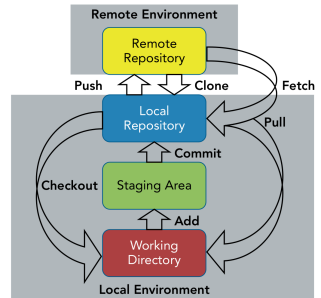
- Takes all the changes that have been added to the staging area and puts them into the local repository as a new snapshot
- `git commit -m "descriptive message"`
- Leave out the `-m "descriptive message"` and your editor will open to let you write the message
- For larger commits, the message should have:
  - a short, one line subject
    - this will appear in summary status messages and logs
  - an empty line
  - a more detailed body





# git push: Updating the remote repository from the local

- Takes all the changes that have been made to the local repository and tries to make them on the remote repository
- `git push`
- If no changes were made to the remote repository from a different local repository, this will work
- If other changes have been made to the remote from a different local since the last time this local and remote were synchronised, then you get a *merge conflict*



# Merge Conflicts

Note that any change in the remote repository's snapshot that was not seen by the local repository before the push is a merge conflict. Suppose that the remote contains files `x.txt` and `y.txt`:

- 1 local1 clones remote
- 2 local2 clones remote, updates `x.txt` and pushes successfully
- 3 local1 updates `y.txt` and pushes
- 4 Push is rejected because local1 has not seen the updated `x.txt`

This is even though local1 has not changed any file that local2 has changed or vice versa

# Dealing with Merge Conflicts

The remote **rejects** pushes that have merge conflicts. Solution:

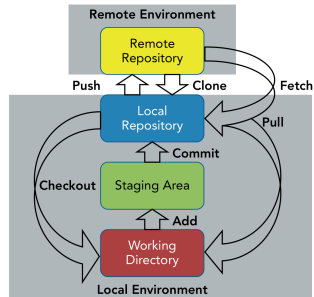
- Pull to update the local with all changes from the remote
- This overwrites individual files in the working directory with new versions if those files have no conflicts
- For files that have conflicts, a merged version of the files are created which will have one or more sections as follows:

```
this line is the same in both versions
<<<<<< HEAD
this line is in the remote version only
=====
These lines are in
the local version only
>>>>>>
this line is the same in both versions
```

- The user must correct these sections to whatever he/she considers the appropriate merge result
- All fixed files should be added and then a commit made
- Finally the push can be re-tried

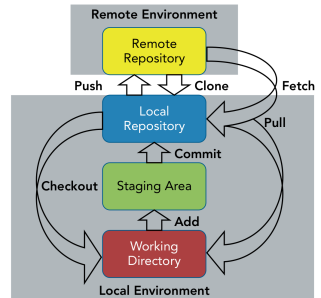
# git pull: Updating the local repository and working directory from the remote

- Takes all the changes that have been made to the remote repository and updates the local repository with them, merging changes into the working directory
- `git pull`
- This is the standard way to update your local system from the remote repository.
- To minimise problems you should stage working directory changes and commit to the local repository before calling pull.
- If there are merge conflicts, the corresponding files in the working directory will require fixing as described above.

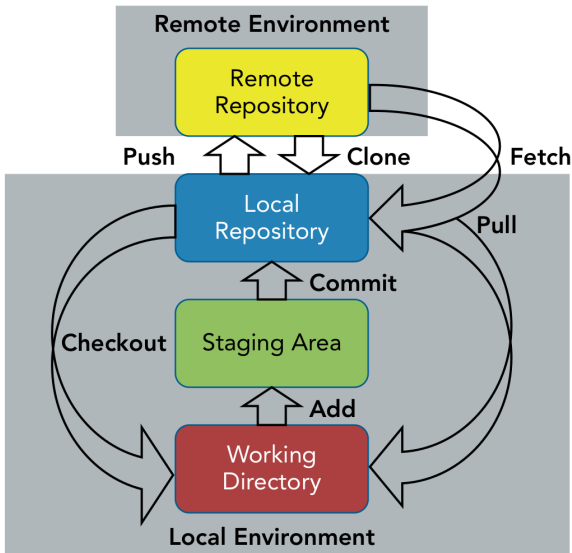


# git fetch, git merge and git checkout

- git pull is actually composed of two commands:
  - 1 git fetch, which copies the changes in the remote repository into the local repository, but does not modify the working directory
  - 2 git merge, which copies out the appropriate parts of the local repository onto the working directory, merging where possible and creating the special failed merge files where necessary
- git checkout replaces the contents of the working directory with chosen snapshots of the local repository history. This is mostly used for dealing with *Branches*, an important topic that we will discuss later



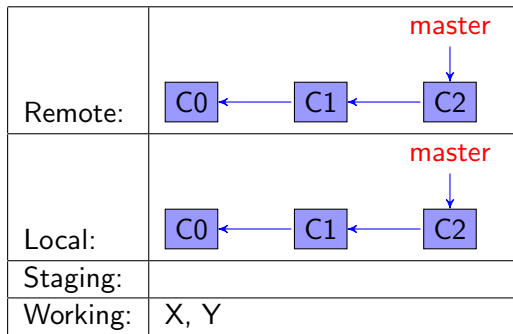
# Summary So Far: GIT Commands



taken from: Brent Laster, *Professional GIT*, O'Reilly 2017

## Basic GIT Examples

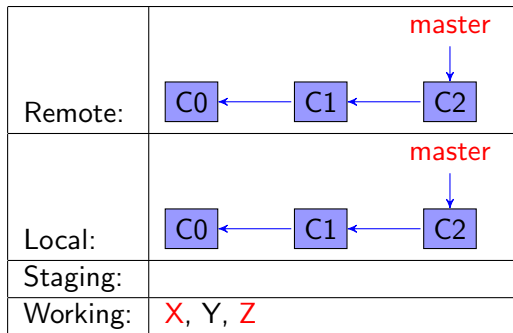
# Making Changes on One Machine



## 1. Initial State

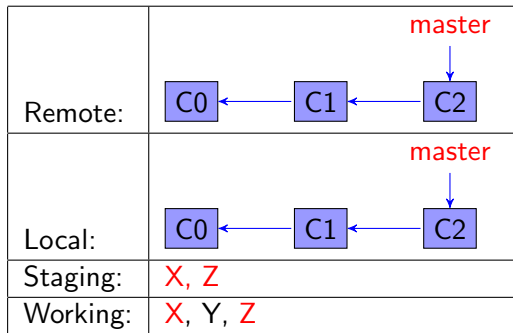


# Making Changes on One Machine



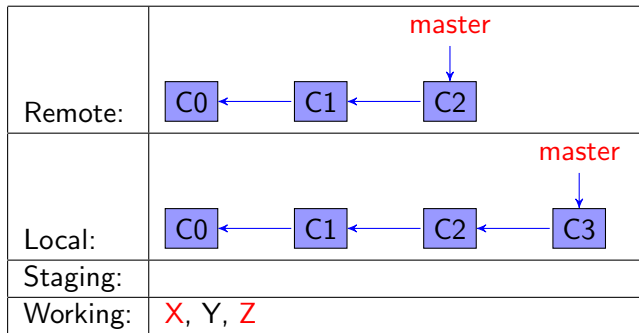
2. After modifying X and creating Z

# Making Changes on One Machine



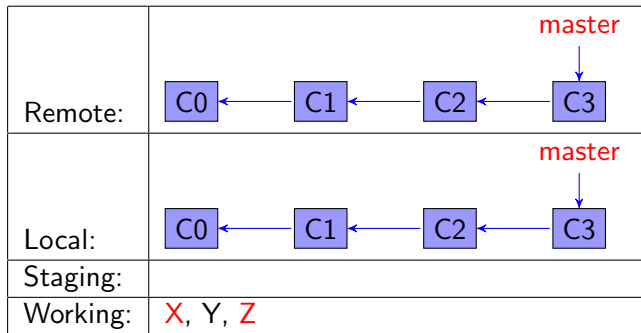
3. After: `git add X Z`

# Making Changes on One Machine



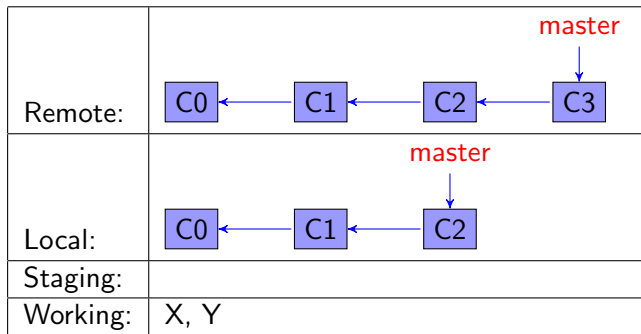
4. After: `git commit -m "updated files"`

# Making Changes on One Machine



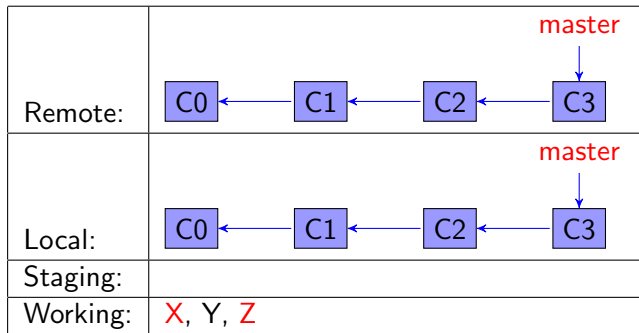
5. After: `git push`

# Getting Changes on a Different Machine



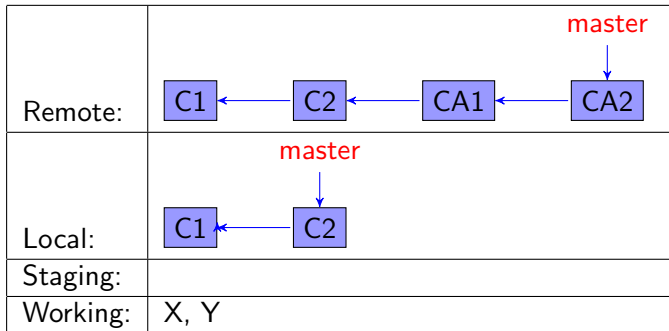
## 1. Initial State

# Getting Changes on a Different Machine



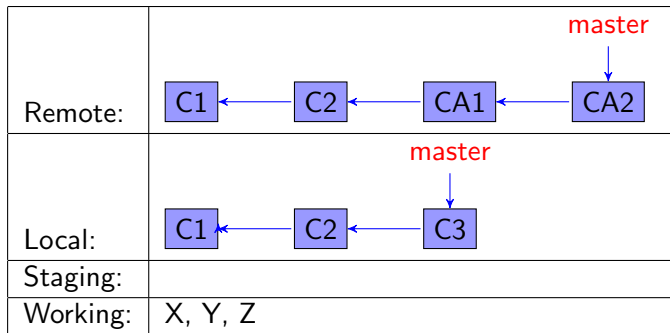
2. After `git pull`

# Merge Conflicts



1. Initial State: On remote, **X** has been modified and **W** added

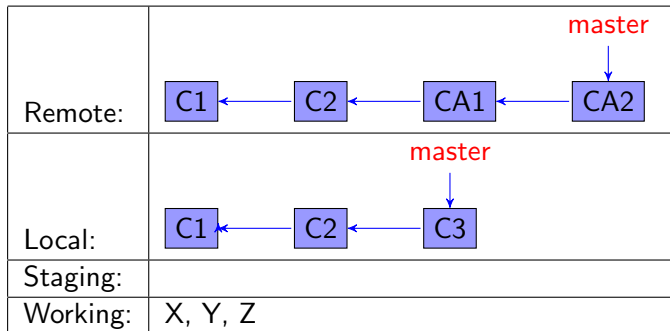
# Merge Conflicts



2. After modifying X, creating Z, adding and committing



# Merge Conflicts



## 3. Try `git push`: Fails with error message:

To <https://git.cs.bham.ac.uk/aps/myproject.git>

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'https://git.cs.bham.ac.uk/aps/myproject.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push -help' for details.

# Merge Conflicts, ...try git pull

```
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://git.cs.bham.ac.uk/aps/myproject
   69b71d5..dae30e0  master    -> origin/master
First, rewinding head to replay your work on top of it...
Applying: Modified X, added Z in myproject1
... <INFO ABOUT DIFFERENCES> ...
Falling back to patching base and 3-way merge...
Auto-merging X.txt
CONFLICT (content): Merge conflict in X.txt
Failed to merge in the changes.
Patch failed at 0001 Modified X, added Z in myproject1
The copy of the patch that failed is found in:
    /home/aps/repos-git/bham/myproject1/.git/rebase-apply/patch
```

When you have resolved this problem, run "git rebase --continue".  
If you prefer to skip this patch, run "git rebase --skip" instead.  
To check out the original branch and stop rebasing, run "git rebase --abort".

# Merge Conflicts, ...git status

```
rebase in progress; onto dae30e0
You are currently rebasing branch 'master' on 'dae30e0'.
(fix conflicts and then run "git rebase --continue")
(use "git rebase --skip" to skip this patch)
(use "git rebase --abort" to check out the original branch)
```

Changes to be committed:

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file:   Z.txt
```

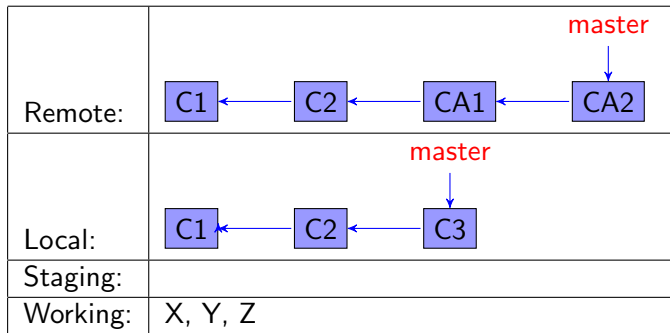
Unmerged paths:

```
(use "git reset HEAD <file>..." to unstage)
```

```
(use "git add <file>..." to mark resolution)
```

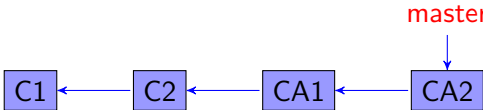
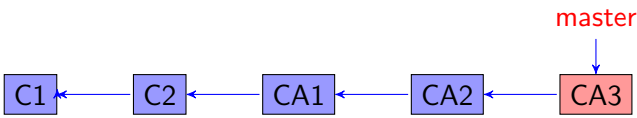
```
both modified: X.txt
```

# Merge Conflicts, ... Continued



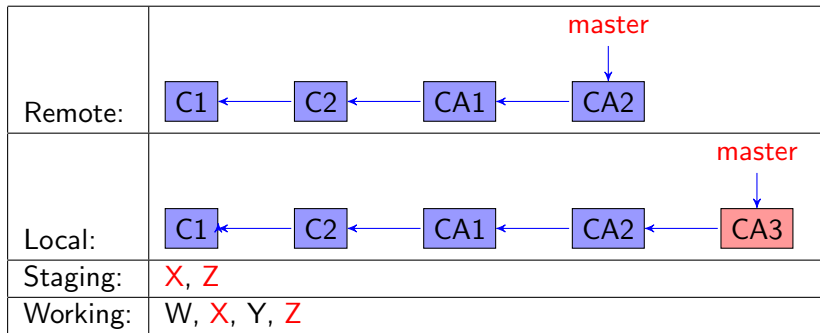
3. Current state after failed `git push`

# Merge Conflicts, ... Continued

Remote:	 <p>A horizontal sequence of four blue boxes labeled C1, C2, CA1, and CA2 from left to right. Blue arrows point from C2 to C1, CA1 to C2, and CA2 to CA1. A red arrow labeled 'master' points down to the CA2 box.</p>
Local:	 <p>A horizontal sequence of five boxes labeled C1, C2, CA1, CA2, and CA3 from left to right. C1, C2, CA1, and CA2 are blue, while CA3 is red. Blue arrows point from C2 to C1, CA1 to C2, CA2 to CA1, and CA3 to CA2. A red arrow labeled 'master' points down to the CA3 box.</p>
Staging:	Z
Working:	W, X, Y, Z

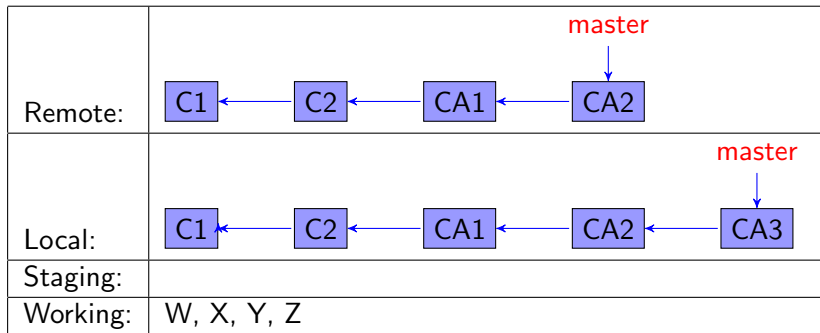
4. After `git pull`: X is in conflict

# Merge Conflicts, ... Continued



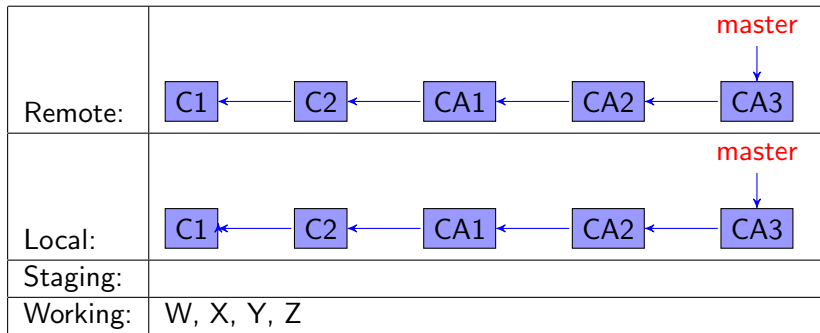
5. After correcting X and `git add X`

# Merge Conflicts, ... Continued



6. After `git rebase --continue`

# Merge Conflicts, ... Continued



7. After `git push`



# Merge Conflicts, Final Note ...

- The `git push` that failed required us to do a `git pull` to update our local repository, getting the new **W** and finding the changes to **X**, which we had already changed locally
- That `git pull` required us to fix the clash in **X**, then do the `git rebase --continue`
- If there had not been the clash in **X**, the `git pull` would have successfully merged the remote changes into our local repository and our working directory: no further rebases, adds or commits necessary.
- Lesson: Dealing with merge conflicts is trivial if they don't involve conflicting changes to the same file
- Resolving conflicts can be easier using a merge tool (I like `meld`, which provides a GUI to correct differences): call it with `git mergetool`. Try `git mergetool --tool-help` to see options, and you can set the default merge tool with `git config -global merge.tool=meld`

## Common Other Commands

All of these commands have many options: see the manuals or `git help` for details

- `gitk`: visual browsing of your local repository
- `git log`: shows, for each commit (i.e. snapshot) the SHA1 identifier, the author, the message and the date and time
  - `git log --pretty=oneline` gives a one line summary per snapshot
  - `git log --decorate` shows the reference names that are attached to each snapshot
  - `git log --graph` shows an ascii graph drawing of the commit history of the repository
- `git status`: shows
  - 1 which files are tracked/untracked
  - 2 what is in the staging area
  - 3 which files in the working directory are different from the latest version in the local repository.

- `git diff`: allows comparison of different versions of files
  - `git diff`: compares the working directory against the latest version in the local repository
  - `git diff x.java`: compares the working directory vs the local repository versions of `x.java`
  - `git diff 73a364e 6c829b2`: compares the two versions specified by their SHA1 identifiers
  - `git diff 73a364e HEAD`: compares the version specified with the latest version in the local repository

## git reset: for rolling back locally only

This command rolls back to an earlier version (leave out the SHA1 identifier to roll back to the latest committed version):

- `git reset --soft c724e12` Updates **only** the local repository to make the specified (old) version be the official latest (later versions are still there). Staging area and working directory are left unchanged
- `git reset --mixed c724e12` As above but updates the staging area to correspond to the new version
- `git reset --hard c724e12` As above but updates the staging area and working directory to correspond to the new version. This throws away any changes in the working directory that has not been added and committed!
- Note: later versions are still available from the local repository (using `git checkout`)
- **IMPORTANT:** don't reset to a version before one that has been pushed to the remote repository as it can cause serious difficulties with merge conflicts

This command is like a reset but works differently

- ① it gets an old version
- ② it calculates the changes that have to be added to the latest version to return the contents to the old version
- ③ then it commits those changes.

The difference from reset is that

- it simply adds a new commit rather than trying to change the history
- because the history is not changed, it is safe to revert to a commit before one that has already been pushed
- the history, however, does show the bad commits that are being thrown away

- **Don't add generated content**
  - Never add class files, formatted PDFs that can be generated from source files etc.
- Beware of large binary files
  - changes normally require entirely new copies to be added to the repository.
- Be careful with the frequency of commits and pushes
  - Wait too long between pushes and you may get large merge problems
  - Never commit/push non-working code to the main branch (trunk)
  - Typically commit/push when a consistent working change/feature has been made
  - Often commit/push at the end of a working session

- Single User Project:
  - Remote repository provides safety in case of local disk failure/stolen laptop etc.
  - Local repository on each machine: laptop, home, lab, . . .
  - Always add/commit/push when finished on one machine and pull before starting on a different one
- Small Multi User Project:
  - Single Remote Repository
  - All users push to the same remote repository
  - Deal with merge conflicts when they occur
- Large Multi User Project:
  - Each user has their own remote repository and work with their own repository as per Single User Project
  - When one user has changes that should be integrated into all copies, he/she generates a pull request for the other users
  - Other users pull the changes and can check/review changes before merging into their production systems
  - Truly distributed: no single central master repository



# Branching

# Branches

Problem: When making a significant change (e.g. a new feature), we don't want to work in the main trunk:

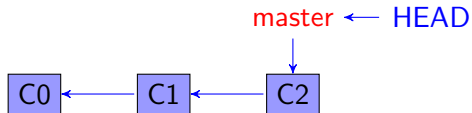
- Don't want broken, untested code to infect production code
- Don't want to stall other work on production code while waiting on the new feature

Solution: **Branches**

- Make a new branch
- Work in the new branch (including commits and pushes) without effecting trunk
- Incorporate updates from the trunk as desired
- Temporarily switch from the branch to the trunk if work is necessary on the trunk
- Switch back to work on the branch
- Finally merge the branch back into the trunk when feature is complete

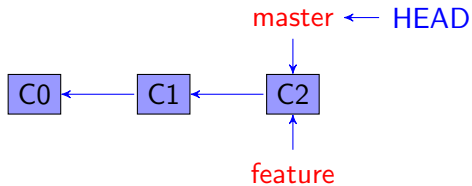
- A **branch** is a chain of snapshots (commits) that is identified by a branch name.
- The **master** branch is, by default, the main trunk
  - Nothing special about the name “master”, just convention
- **HEAD** is not a branch name, it is a **symbolic reference**, or pointer, that points at the latest snapshot that we are currently working on
  - It says which snapshot to base a new commit on
  - If working on the trunk, it normally points at **master**
  - If working on non-trunk branch, it normally points at the latest snapshot on that branch
  - It can be set to an old snapshot, for example to start a new branch from an old version

# Creating Branches



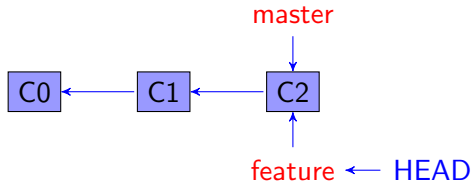
Initial state

# Creating Branches



After: `git branch feature`

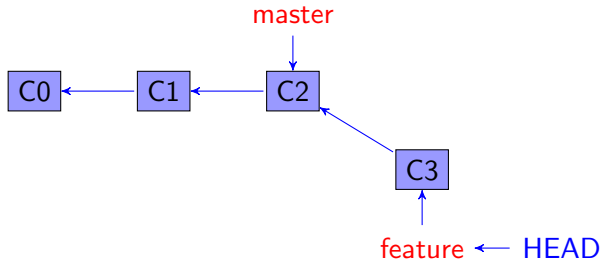
# Creating Branches



Switch to feature branch: `git checkout feature`

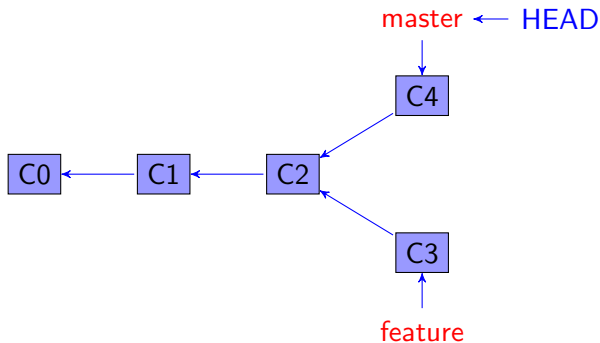
- This **REPLACES** current working directory contents with contents for feature branch
- Make sure you have committed changes on current branch **BEFORE** switching to a new one!

# Creating Branches



After making some changes and committing

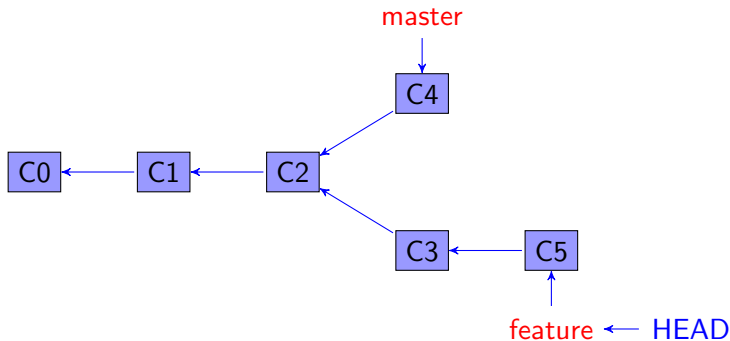
# Creating Branches



Work needed on main trunk: switch back to master (`git checkout master`), make changes and commit

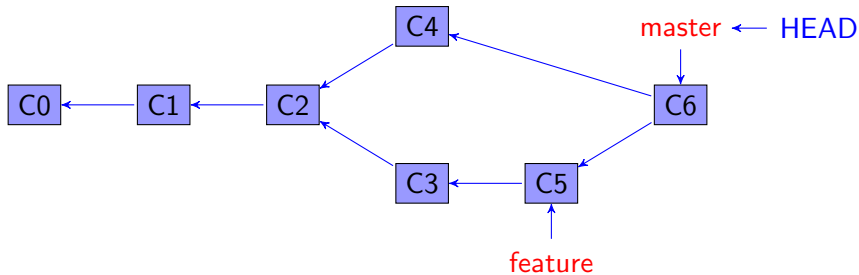


# Creating Branches



Switch back to feature (`git checkout feature`), make changes and commit

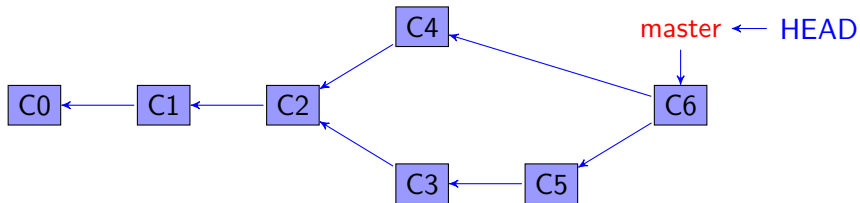
# Creating Branches



Feature is now finished: merge it back into the trunk:

- Switch back into the branch that you want to merge into:  
`git checkout master`
- Merge in the changes from feature:  
`git merge feature`

# Creating Branches



Finished with feature: delete the branch

- `git branch -d feature`

# Merge Conflicts while Merging Branches

At the point where you try to merge branches, you may get merge conflicts:

- In example above, if the **master** branch has commits (C4) with changes that conflict with changes in C3 and C5 of the **feature** branch
- Git gives you an error message and marks conflicts in files as we saw earlier
- Fix the problem files, add them to the staging area and commit

## Contributing to someone else's Project

# Contributing Patches

Suppose you want to contribute a change to a project in a public repository that you do not have write access to.

- The simplest way of doing so is to contribute a **patch**.
- A **patch** is a text file of a particular format that records all the changes to all files to apply a correction or update.
- Contributor clones the repository, makes a branch to develop the patch in, makes changes and commits, creates a patch file for his/her changes relative to the **master** branch of the repository, and sends the patch file to the repository owner.
- Repository owner reviews the patch, and, if happy with it, signs it off (after possibly making further changes) and applies it to his/her local repository and pushes it to the public remote repository.
- Contributor can switch back to the master branch, delete the branch that was used to develop the patch changes and pull the new version of master with the integrated patch.

# Patching in Detail: Contributor's Part

- Contributor: after cloning the repository, create a new branch:

```
git checkout -b my_patch
```

- Contributor: make the necessary changes, add them and commit:

```
git commit -m "fix for ..."
```

- Contributor: Create the patch file relative to the **master** branch and send to the Owner:

```
git format-patch master
```

This creates a patch file (name ending in ".patch"). Send this (**as an attachment!**) to the Owner.

- NOTE: It is important **NOT** to reformat code or reflow text lines in a patch or you will introduce large spurious patches that are very hard to review
- NOTE: Developing the patch in a branch:
  - avoids problems in identifying the correct snapshot to make the patch relative to and
  - ensures that the final integrated patch in the public repository will not conflict with the local patch changes by the contributor

# Patching in Detail, Owner's Part

- Owner: extract the patch from the email and check what it involves (without applying it!):  
`git apply --stat <filename>.patch`
- Owner: check whether the patch will cause any conflicts:  
`git apply --check <filename>.patch`
- Owner: Assuming no conflicts, apply the patch and sign it off:

```
git am --signoff < <filename>.patch
```

The sign-off means that the log file will contain a message identifying who accepted the patch. Also, this command completes the commit as well, but not the push to the public repository.

- Owner: Push the changes to the public repository:  
`git push`



# Patching in Detail: Owner's Part — Conflicts

If there are conflicts in applying the patch, the check will fail and there are a number of strategies for the owner to try:

- If the patch is small, they can be applied by hand
- Often the failure is due to whitespace problems — particularly end of line differences between Window, Macs and Linux. Ignoring whitespace in applying the patch may succeed:

```
git apply --reject --ignore-space-change --ignore-whitespace  
    <filename>.patch
```

The `--reject` option allows the patch to go ahead even if some of the patch changes cannot be applied: rejected patch parts are saved in `*.rej` files which can then be manually corrected.

- If the conflicts are too severe, use a 3 way merge and resolve as for any git merge problem: `git apply --3way <filename>.patch`
- NOTE: `git am` applies and **commits** a whole mailbox full of patch files. `git apply` only applies a single patch file to the current working directory: they then still have to be added and committed (and pushed!).

# Patching in Detail: Contributor's Cleanup

At this point, the public repository has the patches integrated onto the **master** branch, and we left the contributor on the `my_patch` branch. To clean up, the contributor needs to switch back the master, pull the latest version and delete the `my_patch` branch:

```
git checkout master
git pull
git branch -D my_patch
```

NOTE: the “-D” argument forces the delete of the branch even though it has not been pushed to the remote repository (which it cannot be as the contributor does not have write access to that repository).

# Pull Requests

There is another approach to contributing to someone else's project, which we will not study here in detail: **pull requests**. In brief:

- Contributor **forks** the Owner's public repository as his/her own public repository
  - Forking simply means cloning as a new public repository: standard repository providers like Github, Gitlab, Bitbucket etc. provide a quick and easy way to do this, but it can be done easily from the command line too.
  - Contributor makes the necessary changes, usually in a branch off the main trunk, commits and pushes to this new fork
  - Contributor generates a **pull request** on the Owner's public repository with respect to the head of the change branch on the Contributor's repository
  - The Owner reviews the pull request and merges it into his/her repository

The advantage of this approach is that multiple developers can maintain their own forks (i.e. public mirrors of the Owner's repository) and submit their changes to each other

GIT is a large and complex system which is very powerful and useful, but it is also not very user friendly:

- Conceptually simple operations often require quite complex sequences of commands with complex sets of options
- Documentation assumes detailed understanding of the underlying architecture, design and terminology of the system
- Very poor level of abstraction in the user interface
- There is a great deal more to learn about GIT, some of which may be critical when working on software development in industry.
- Only the basic essentials have been covered here!