# Introduction to git

Lorne Whiteway
lorne.whiteway@star.ucl.ac.uk
Thanks to Tomas James for assistance

Astrophysics Group
Department of Physics and Astronomy
University College London

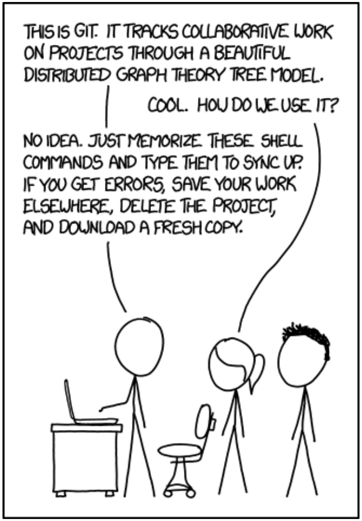
17 November 2017

## Where to find this presentation

Find the presentation at `https://tinyurl.com/y8pr4mvq`.

On this page click on ‘Download’ to get a copy of the presentation.

## https://xkcd.com/1597/


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

## Purpose of presentation

- I don't want to teach you how to use git.
- Rather I want to illustrate (part of) git's 'internal model' and to define certain key git terminology so that you will be better prepared to teach yourself git.
- My examples assume you are calling git from the command-line. Friendlier interfaces to git exist - but you still need to know the underlying model to use them effectively.

## git isn't optimal

- The internal model is complicated.
- The interface is inconsistent.
- The documentation is suboptimal.
- Several key ideas have been given misleading names.
- It uses a 'distributed' model whereas what you usually want is a 'client/server' model. So you tend to be 'fighting against the paradigm'...
- See more at `https://stevebennett.me/2012/02/24/10-things-i-hate-about-git/`.

## Source control

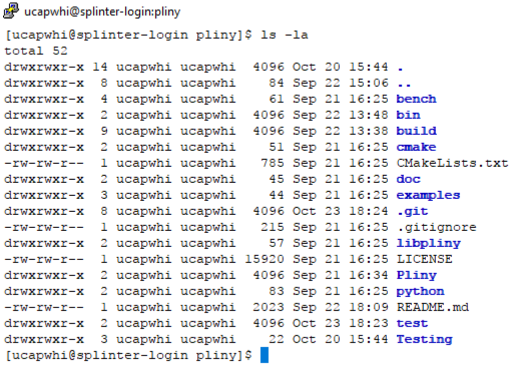
- Source control is software to ‘keep track of’ (i.e. store) successive versions as we edit a collection of *source* files (computer code, LaTeX documents, etc.)
- Works better on source files that are text, not binary. Intermediate files are usually not kept track of. Output files might be - your choice.
- Any serious project should be under source control.

## Working directory and repository

- You need a *working directory* and a *repository*.
- The working directory and its subdirectories contain the actual files that you are editing.
- The repository is some sort of database containing all previous versions.
- One model would be to put the repository on the Internet or Intranet where everyone can see it...

## Location of git repositiory

- … But in git the repository is **next to** the working directory, in a hidden subdirectory (called .git) of the top-level working directory.



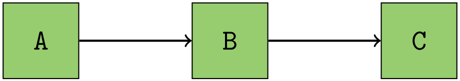
ucapwhi@splinter-login:pliny

```
[ucapwhi@splinter-login pliny]$ ls -la
total 52
drwxrwxr-x 14 ucapwhi ucapwhi  4096 Oct 20 15:44 .
drwxrwxr-x  8 ucapwhi ucapwhi    84 Sep 22 15:06 ..
drwxrwxr-x  4 ucapwhi ucapwhi    61 Sep 21 16:25 bench
drwxrwxr-x  2 ucapwhi ucapwhi  4096 Sep 22 13:48 bin
drwxrwxr-x  9 ucapwhi ucapwhi  4096 Sep 22 13:38 build
drwxrwxr-x  2 ucapwhi ucapwhi    51 Sep 21 16:25 cmake
-rw-rw-r--  1 ucapwhi ucapwhi   785 Sep 21 16:25 CMakeLists.txt
drwxrwxr-x  2 ucapwhi ucapwhi    45 Sep 21 16:25 doc
drwxrwxr-x  3 ucapwhi ucapwhi    44 Sep 21 16:25 examples
drwxrwxr-x  8 ucapwhi ucapwhi  4096 Oct 23 18:24 .git
-rw-rw-r--  1 ucapwhi ucapwhi   215 Sep 21 16:25 .gitignore
drwxrwxr-x  2 ucapwhi ucapwhi    57 Sep 21 16:25 libpliny
-rw-rw-r--  1 ucapwhi ucapwhi 15920 Sep 21 16:25 LICENSE
drwxrwxr-x  2 ucapwhi ucapwhi  4096 Sep 21 16:34 Pliny
drwxrwxr-x  2 ucapwhi ucapwhi    83 Sep 21 16:25 python
-rw-rw-r--  1 ucapwhi ucapwhi  2023 Sep 22 18:09 README.md
drwxrwxr-x  2 ucapwhi ucapwhi  4096 Oct 23 18:23 test
drwxrwxr-x  3 ucapwhi ucapwhi    22 Oct 20 15:44 Testing
[ucapwhi@splinter-login pliny]$
```

## This has consequences...

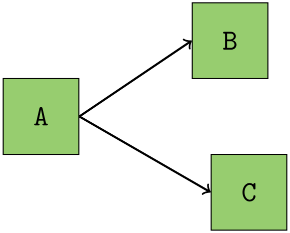
- You therefore need to have a git repository next to your working directory on your local directory (where you are doing the actual editing).
- But typically you will also need one on the internet (for backup and for collaboration and sharing).
- So you will typically be dealing with **two** git repositories (and dealing with the issues of keeping them in synch.
- The upside is that you can still do version control even if you are not connected to the Internet.

## Example repository content



- This repository contains three successive versions of the files in the working directory (and its subdirectories).
- Each version is represented here as a *node* (in green).
- An initial set of files (version A) was comitted to the repository; the files were then edited and the new file set (version B) was comitted; the files were then edited and comitted a third time (version C).

## Another example



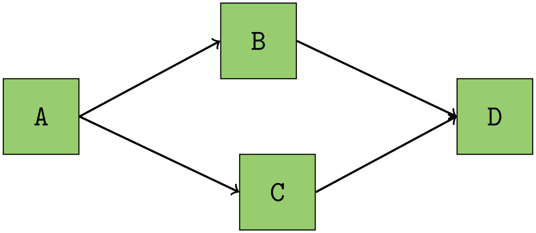
- Here we committed A...
- Then we edited A (to form B) and committed B...
- Then we went back to A, made a perhaps different set of edits (to form C) and committed C.

## What do the arrows actually stand for?

$$\begin{tikzcd}
 & \boxed{B} \\
\boxed{A} \arrow[ur] \arrow[dr] & \\
 & \boxed{C}
\end{tikzcd}$$

- An arrow respects time (pointing from an earlier version to a later version), and indicates that a node was derived from an earlier node by editing.
- Q: There exists a set of edits that would take you from B to C, so why not show that arrow as well? A: It's an *itinerary* (showing the route we took), not a *map* of all possible routes.

## Merging



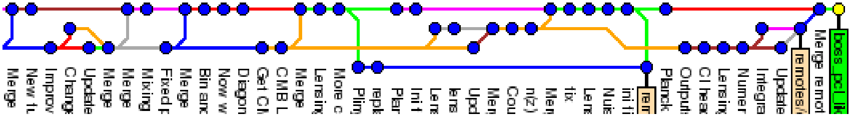
- Here we combined the ‘A to B’ edits and the ‘A to C’ edits (to form D), which was committed.
- More on such *merging* later.

## DAG

- Hence we get a graph (in the mathematical sense of nodes plus edges).
- The graph is not a *tree* (because of merging).
- It is *directed* (edges have arrows) and *acyclic* (cycles would break causality), so we have a *DAG* (directed acyclic graph).
- Nodes have *parents* and *children*, and hence *ancestors* and *descendents*.
- One node - the initial *root* node - has no parents. All other nodes have one or two parents (or more in special cases). Thus the graph is connected, and any two nodes have ancestors in common.
- *Graph theory* is an interesting part of mathematics - but alas not useful here.

## Seeing the DAG

- Run `gitk --all` to see the DAG (lots of other information as well).



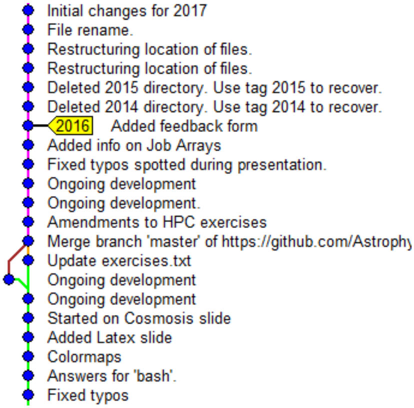
## What's in a node?

At least:

- Pointers to enough information to reconstruct the working directory as of that instant.
- Administrative information: who made the commit, a 'commit message', etc.
- A permanent node name (SHA1 format - 40 hex digits e.g. c2d2ea34cec13a0956488f2b919861fccad8a448). You can abbreviate this to an initial substring (provided that it is long enough to be unique).

## Pointers to nodes - tags

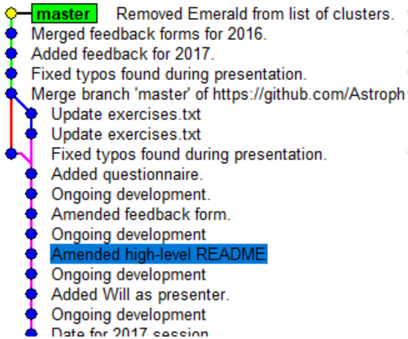
- A *tag* is a label that is attached to a node.



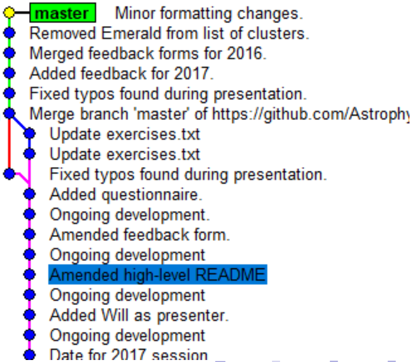
```
Initial changes for 2017
File rename.
Restructuring location of files.
Restructuring location of files.
Deleted 2015 directory. Use tag 2015 to recover.
Deleted 2014 directory. Use tag 2014 to recover.
2016   Added feedback form
Added info on Job Arrays
Fixed typos spotted during presentation.
Ongoing development
Ongoing development.
Amendments to HPC exercises
Merge branch 'master' of https://github.com/Astrophy
Update exercises.txt
Ongoing development
Ongoing development
Started on Cosmosis slide
Added Latex slide
Colormaps
Answers for 'bash'.
Fixed typos
```

## Pointers to nodes - branches (1)

- A *branch* is a moveable label that is attached to a node.
- Here ‘moveable’ means that if you commit an edit to that node then the branch label moves to the child node.



```
master  Removed Emerald from list of clusters.
Merged feedback forms for 2016.
Added feedback for 2017.
Fixed typos found during presentation.
Merge branch 'master' of https://github.com/Astroph
  Update exercises.txt
  Update exercises.txt
  Fixed typos found during presentation.
  Added questionnaire.
  Ongoing development.
  Amended feedback form.
  Ongoing development
  Amended high-level README
  Ongoing development
  Added Will as presenter.
  Ongoing development
  Date for 2017 session
```



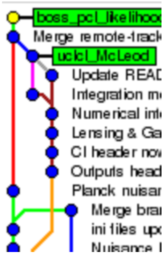
```
master  Minor formatting changes.
Removed Emerald from list of clusters.
Merged feedback forms for 2016.
Added feedback for 2017.
Fixed typos found during presentation.
Merge branch 'master' of https://github.com/Astrophy
  Update exercises.txt
  Update exercises.txt
  Fixed typos found during presentation.
  Added questionnaire.
  Ongoing development.
  Amended feedback form.
  Ongoing development
  Amended high-level README
  Ongoing development
  Added Will as presenter.
  Ongoing development
  Date for 2017 session
```

## Pointers to nodes - branches (2)

- Because it moves with the commits, the branch is often thought of as a ‘development line’ (imagine not just the node that the branch points to, but also all the past and future nodes that the branch has pointed to or will point to).
- In some contexts (e.g. cloning) a branch is used to refer not only to the pointed-to node but also all its ancestors.

## Pointers to nodes - branches (3)

- A branch can point to any node in the DAG - not just to a terminal node (= node with no descendents).
- So ‘branch’ is not a perfect metaphor; better is ‘pointer to a place where growth may occur’ (which may be on the trunk).

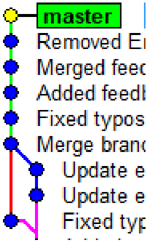


## Pointers to nodes - branches (4)

- When git creates a new repository it creates a branch called ‘master’ that points to the root node.
- House rules often insist that the node pointed to by ‘master’ be ‘good’ (i.e. stable, tested, usable); development is then done in other branches that are merged into master following testing and agreement.
- But this is not forced by git. You don’t even need to have a master branch.

## Pointers to nodes - HEAD (1)

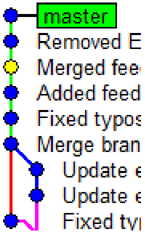
- The special pointer HEAD points to the node that corresponds to the files in the working directory. This can be any node in the DAG.
- Use `git checkout <nodename>` to set where HEAD points.
- In gitk the HEAD node is shown in yellow.



```
Removed E
Merged feec
Added feedl
Fixed typos
Merge branc
Update e
Update e
Fixed typ
```

## Pointers to nodes - HEAD (2) - Detached HEAD

- Usually HEAD points to the same node as one of the branches. But it doesn't have to; if it doesn't then we say we have a 'detached HEAD'.



## Name that node

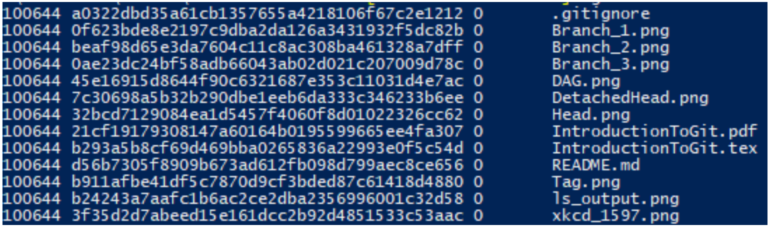
- Refer to a node using its (abbreviated) SHA1 name e.g. ‘c2d2ea34’.
- Or use a pointer name (a tag name, a branch name, or the name ‘HEAD’).
- Can append ^ to mean ‘parent of’, ^^ to mean ‘grandparent’, etc.
- Example: `git checkout master^^`.
- Distinguish multiple parents via ^1 or ^2.
- Alternatively e.g. ~4 means the same as ^^^^.

## DAG manipulation

- Much (but not all) of your interaction with a git repository involves maintaining and amending the DAG and the pointers to nodes in the DAG.

## The index

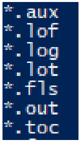
- In addition to the DAG, a git repository also contains an *index*: a list of the files in the working directory that git knows about (i.e. that git is 'tracking').
- The index contains other information that makes it fast to answer the question 'has this file changed since a version of it was last put into the repository?'



```
100644 a0322dbd35a61cb1357655a4218106f67c2e1212 0       .gitignore
100644 0f623bde8e2197c9dba2da126a3431932f5dc82b 0       Branch_1.png
100644 beaf98d65e3da7604c11c8ac308ba461328a7dff 0       Branch_2.png
100644 0ae23dc24bf58adb66043ab02d021c207009d78c 0       Branch_3.png
100644 45e16915d8644f90c6321687e353c11031d4e7ac 0       DAG.png
100644 7c30698a5b32b290dbe1eeb6da333c346233b6ee 0       DetachedHead.png
100644 32bcd7129084ea1d5457f4060f8d01022326cc62 0       Head.png
100644 21cf19179308147a60164b0195599665ee4fa307 0       IntroductionToGit.pdf
100644 b293a5b8cf69d469bba0265836a22993e0f5c54d 0       IntroductionToGit.tex
100644 d56b7305f8909b673ad612fb098d799aec8ce656 0       README.md
100644 b911afbe41df5c7870d9cf3bded87c61418d4880 0       Tag.png
100644 b24243a7aafc1b6ac2ce2dba2356996001c32d58 0       ls_output.png
100644 3f35d2d7abeed15e161dcc2b92d4851533c53aac 0       xkcd_1597.png
```

## .gitignore

- `git status` will warn you about files that **are** in the working directory but **aren't** in the index (so that git isn't paying attention to them).
- This gets boring for intermediate files that you never want git to track. So you can list such 'files always to be ignored by git' in a special file called .gitignore.
- So you probably want .gitignore to be one of the files that git is tracking.



```
*.aux
*.lof
*.log
*.lot
*.fls
*.out
*.toc
```

## So three possibilities...

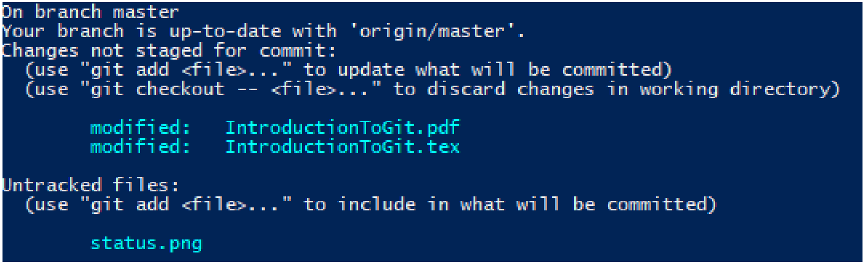
- The index lists the files that git is tracking. (Use `git ls-files --stage` to view the index.)
- The special file .gitignore lists the files that git knows not to track.
- `git status` will warn you about files that are not in either category.
- Use `git add <filename>` to add a file to the index.

## Three types of git operations

1. Managing the relation between a repository and its associated working directory.
2. Repository maintenance.
3. Managing the relation between two repositories (e.g. between your local repository and a copy that is on the Internet).

## Repository ↔ working directory (1)

- See a description of the relationship between the repository and the working directory: `git status`.



```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   IntroductionToGit.pdf
        modified:   IntroductionToGit.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        status.png
```

## Repository $\leftrightarrow$ working directory (2)

- Tell the repository's index about a new file in the working directory that you want the repository to pay attention to: `git add <filename>`.
- Check-in the (changed) files that are in the working directory, thereby creating a new node: `git commit -a`. TODO: WHAT EXACTLY DEFINES THE PARENT OF THIS NEW NODE?
- Using the `-a` option with `commit` minimizes your interaction with the index. There are alternatives that are more work but that give you more control.

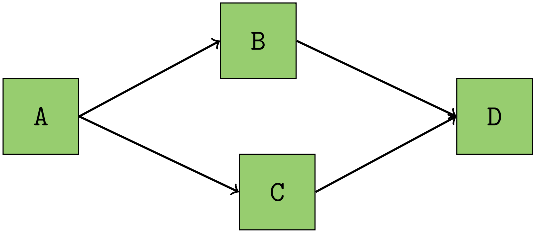
## Repository ↔ working directory (3)

- Use the `-m` option to `git commit` to provide a 'message' that will be associated with the commit.
- Example:
  `git commit -a -m 'Fixed divide-by-zero error.'`
- Or omit the `-m` in which case a text editor window will open and you can type a longer message.
- Writing good commit messages itself is an art; see for example https://medium.com/@steveamaza/how-to-write-a-propergit-commit-message-e028865e5791

## Repository $\leftrightarrow$ working directory (4)

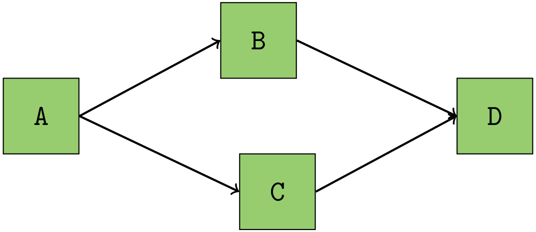
- Make the working directory be equal to the contents of a specific node: `git checkout <nodename>`.
- This fails if there are uncommited changes in the working directory.

## Repository maintenance (1)



- Merge two nodes (here, B and C) to form a new node D with both B and C as its parents.

## Merging (1)



- Merging requires B and C to have a common ancestor (A in the above example). In git this will always be the case.
- Form the union of the 'A to B' changes and the 'A to C' changes. Then apply these united changes to A; this gives D.

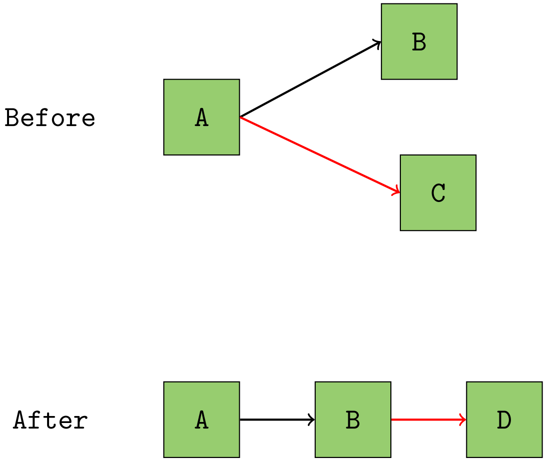
## Merging (2)

- ‘Changes’ include: adding files, deleting files, and amending files (by adding lines, deleting lines, or amending lines).
- Perhaps the ‘A to B’ changes and the ‘A to C’ changes are in conflict. For example, both sets of changes might amend a certain line, but in different ways.
- In this case the merge will fail, and you will have to manually edit D to choose one set of changes or the other (they will both be present in D, with special characters inserted to let you see what is going on). Specialised text editors can be helpful here.
- At merge time the ‘level of granularity’ is the line (if I amend the start of a line and you amend the end of the same line, then the merge will still fail.) Note that binary files can’t be merged.

## Rebase (1)

- *Rebase* is similar to merge, in that it allows two separate development streams to be brought together.
- However, the mental picture with rebase is ‘move the A to C arrow so that it starts from B instead of from A’.
- Alternatively can think of this as being a ‘cut and paste’ to move part of the DAG from one place to another.

## Rebase (2)



## Rebase (3)

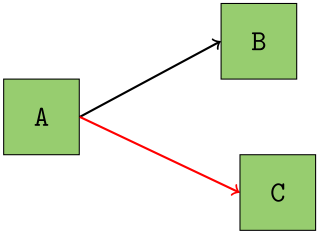
- Rebase yields a cleaner DAG than merge does - this is a possible argument in favour of rebase.
- Rebase ‘rewrites history’ (in the above example, node C will disappear) - this is a possible argument against rebase.
- You will need to decide which to use.

## Cherry-pick (1)

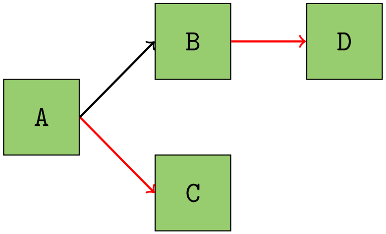
- *Cherry-pick* is similar to rebase, except that it is a ‘copy and paste’ instead of a ‘cut and paste’.

## Cherry-pick (2)

Before



After



## Pointer maintenance

- Tags and branches can be created, moved and deleted.
- Recall that a branch is just a pointer to a node. So moving a branch doesn't involve any rearrangement of the DAG - it means simply changing the node to which the branch points.

## Node cleanup

- If a node is not ‘named’ (i.e. pointed to by a tag or a branch), and if none of its descendents are named either, then git considers it unneeded and will delete it the next time it does ‘automatic cleanup’.
- So to prune an unneeded ‘spur’ simply delete the branch (i.e. the pointer to the last node in the spur). The nodes in the spur will then eventually be deleted.

## Interactions between two repositories (1)

- Use `git clone` to make a copy of a repository.
- The new cloned repository remembers the location of the repository from which it was cloned (i.e. the *remote* repository).
- You can tell the cloned repository a name to use to refer to the remote repository. By default this name is 'origin'.

## Standard working procedure:

1. Create a repository on the Internet;
2. Clone it to make a local copy;
3. Use the commands described above to add and commit to the local repository;
4. Use commands to be described below to keep the local and remote repositories in sync.

## Internet hosting for repositories

- Github (`https://github.com/`) and BitBucket (`https://bitbucket.org/`) are websites for hosting git repositories.
- They allow various levels of read access and write access to these repositories.

## Interactions between two repositories (2)

- Use `git push` to make the remote repository look like the local repository.
- The full syntax is `git push <remote> <branch>`.
- Thus for example you might need to say `git push origin master`.

## Interactions between two repositories (3)

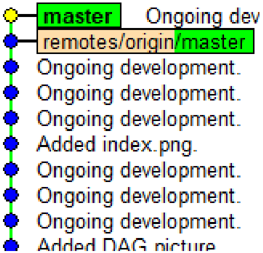
- Use `git pull` to make the local repository look like the remote repository.
- This would be necessary e.g. if someone else had made contributions to the remote repository since you cloned it.

## Interactions between two repositories (4)

- Of course, the local and remote repositories may have diverged since the clone in inconsistent ways.
- In this case the necessary merging will fail and manual intevention will be needed.
- TODO: can I say more here?

## Remote tracking branches

- A cloned repository will have two types of branches: not only normal local branches but also *remote tracking branches* that mirror what is going on in the remote repository.



## Fetch and merge equals pull

- Use `git fetch` to update a remote tracking branch (to get changes from the remote repository); **this won't affect any local branches**.
- You can then use `git merge` to merge the remote tracking branch into the local branch.
- These two operations (`git fetch` followed by `git merge`) are the same as a `git pull`; by doing the job in to stages you can pause after the fetch to inspect what changes you have brought down from the remote repository.

## Github forking and pull requests

- *Forking* is a GitHub innovation to help people contribute to projects to which they don't have write access.
- It uses a 'three repository' model.
- Begin by cloning repository R1 (to which you don't have write access) to get a new repository R2 (to which you do have write access) **that is also on GitHub** - this is called forking.
- Then develop in R2 as usual (this will invloves you cloning R2 to get a local copy R3).
- Once your work is pushed to R2 you email the owners of R1 to ask them to pull your changes from R2 to R1 (*pull request*).
- If all contributors are 'trusted contributors' then this level of complexity is probably not needed.

## Workflows with git

- Many different workflows are possible with git.
- You will need to choose - it's a matter of project objective, project size, team size, team working style, taste, etc..
- Several are discussed here: `https://www.atlassian.com/git/tutorials/comparing-workflows`.