git concepts simplified

Sitaram Chamarty, sitaramc@gmail.com

Adapted in slides deck by Olivier Berger from the HTML version at http://sitaramc.github.com/gcs/

- The title of this page used to be "git for computer scientists – my version", said title being inspired by a much older page at http ://eagain.net/articles/git-for-computer-scientists.
- Following a discussion on IRC where someone said I copied the idea but did not acknowledge the original, I first added the previous para, than sat down to re-read both looking for any similarities. I did not find anything significant except the title itself. (Here's a quick check you can do in a few seconds : click on the "chapter toc" above to see the structure of this one, then try and do a mental "diff" with the old version.)
- Anyway, as a result of this re-look I remembered why I actually wrote it : because I felt the older one was too abstract and terse for a real newbie. I also realised that my version is not really targeted at computer scientists specifically, and arguably never was.
- So I changed the title, carefully picking one that would not affect the URL because I know people have bookmarked it.

the 4 git object types

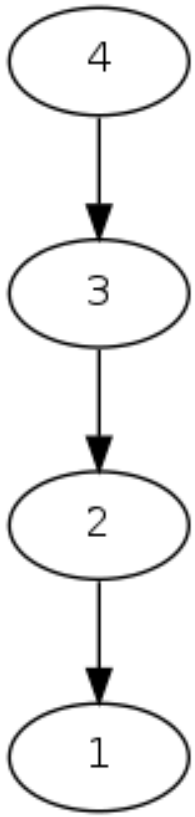
- Git keeps all its data inside a special directory called `.git` at the top level of your repository. Somewhere in there is what we will simply call the *object store* (if you're not comfortable with that phrase, pretend it's some sort of database).
- Git knows about 4 types of objects :
  - **blob** – each file that you add to the repo is turned into a blob object.
  - **tree** – each directory is turned into a tree object. Obviously, a tree object can contain other tree objects and blob objects, just like a directory can contain other directories and files.
  - **commit** – a commit is a snapshot of your working tree at a point in time, although it contains a lot of other information also.
  - **tag** – we will see this type a bit later.

what is a SHA

- A commit is uniquely identified by a 160-bit hex value (the 'SHA'). This is computed from the tree, plus the following pieces of information :
  - the SHA of the parent commit(s) – every commit except the very first one in the repo has at least one parent commit that the change is based upon.
  - the commit message – what you type in the editor when you commit
  - the author name/email/timestamp the committer name/email/timestamp
- (Actually, all 4 git objects types are identified by SHAs, but of course they're computed differently for each object type. However, the SHAs of the other object types are not relevant to this discussion).
- In the end, as I said, it's just a large, apparently random looking, number, which is actually a cryptographically-strong checksum. It's usually written out as 40 hex digits.
- Humans are not expected to remember this number. For the purposes of this discussion, think of it as something similar to a memory address returned by malloc().
- It is also GLOBALLY unique ! No commit in any repo anywhere in the world will have the same SHA. (It's not a mathematical impossibility, but just so extremely improbable that we take it as fact. If you didn't understand that, just take it on faith).
- An example SHA : `a30236028b7ddd65f01321af42f904479eaff549`

what is a repo

A repository ('repo') is a graph of commits.



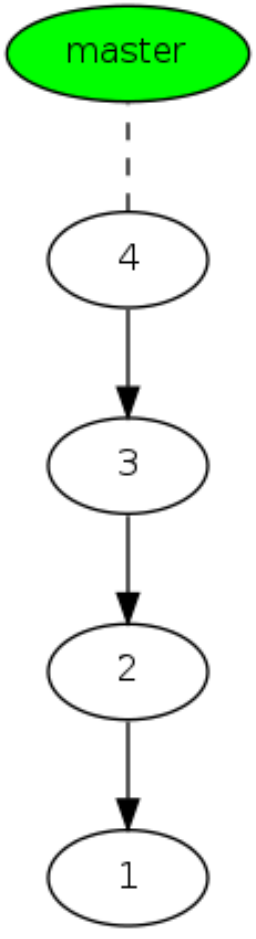
In our figures, we represent SHAs with numbers for convenience. We also represent time going upward (bottom to top).

why are the arrows backward in your pictures ?

- So why are the arrows pointing backward ?
- Well. . . every commit knows what its parent commit is (as described in the "what is a SHA" section above). But it can't know what it's child commits are – they haven't been made yet !
- Therefore a repo is like a single linked list. It cannot be a double linked list – this is because any change to the contents would change the SHA !

branch

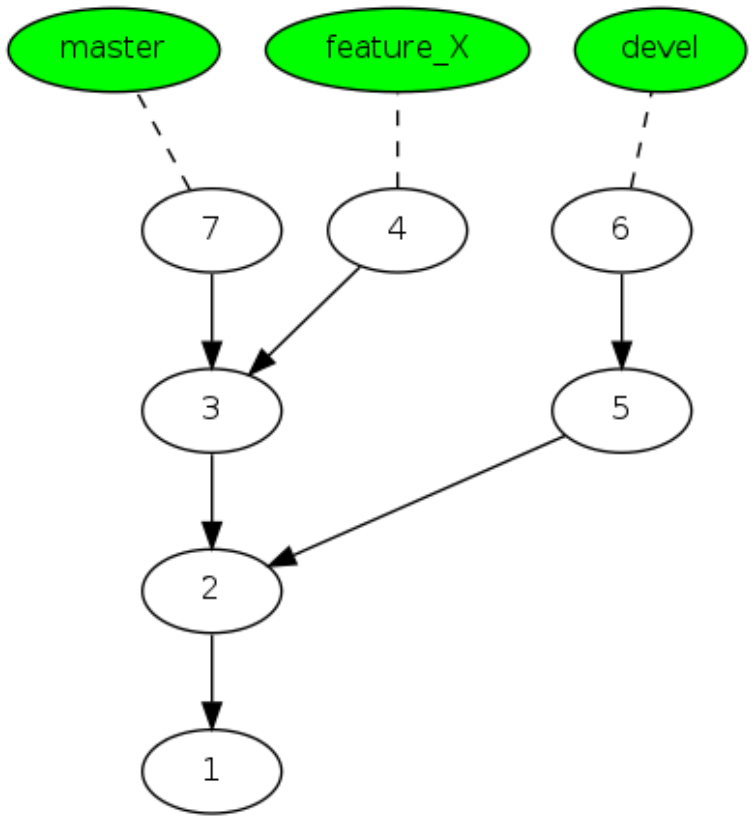
- Traditionally, the top of a linked list has a name.
- That name is a BRANCH name.
- We show branch names in green circles.



more than one branch

(a.k.a “more than one child commit”)

- Remember we said a repo is a GRAPH ?
- Specifically, more than one child node may be pointing at the same parent node.
- In this case, each ‘leaf node’ is a branch, and will have a name.

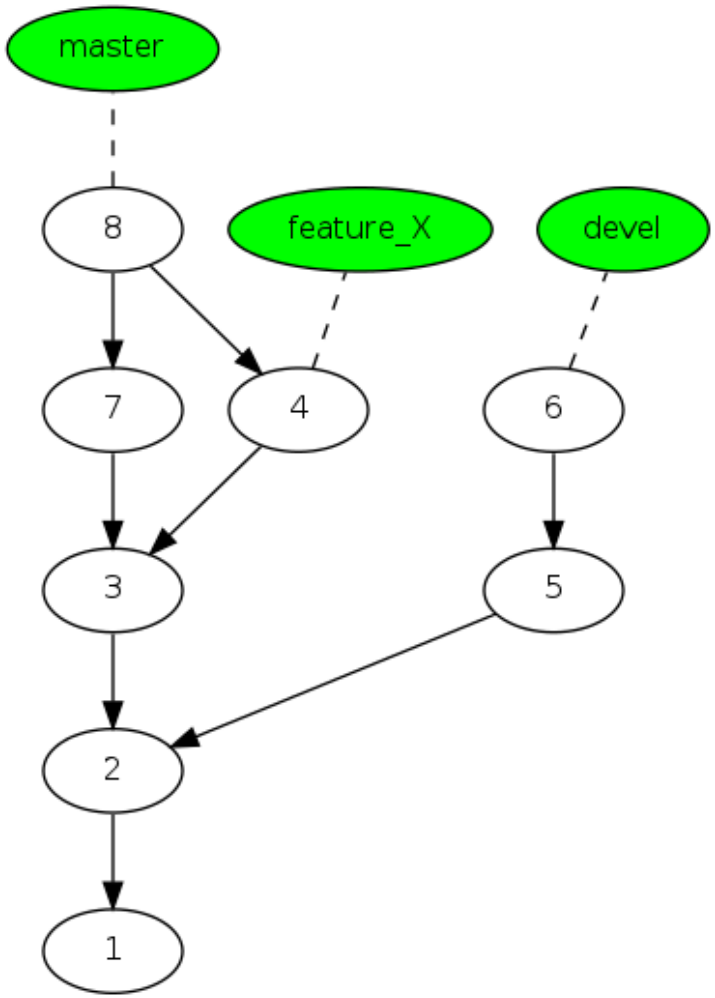


more than one parent commit (1/2)

- Well we can't keep creating more branches without eventually merging them back.
- So let's say "feature X" is now tested enough to be merged into the main branch, so you

```
git merge feature_X
```

- Here's what you get :



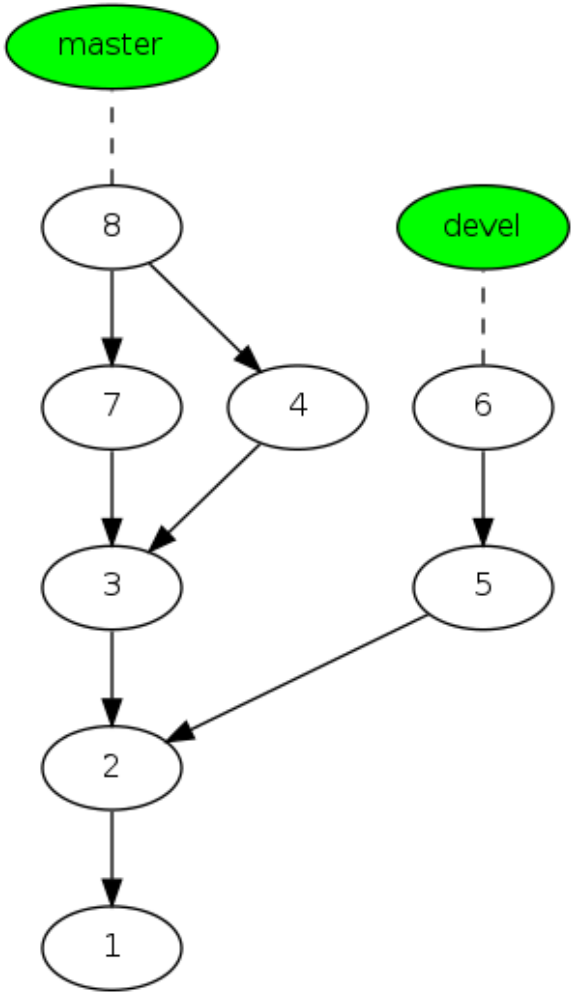
Notice that commit 8 now has 2 parents, showing that it is a "merge commit".

more than one parent commit (2/2)

- At this point, it's quite common to delete the feature branch, especially if you anticipate no more "large" changes.
- So you can run

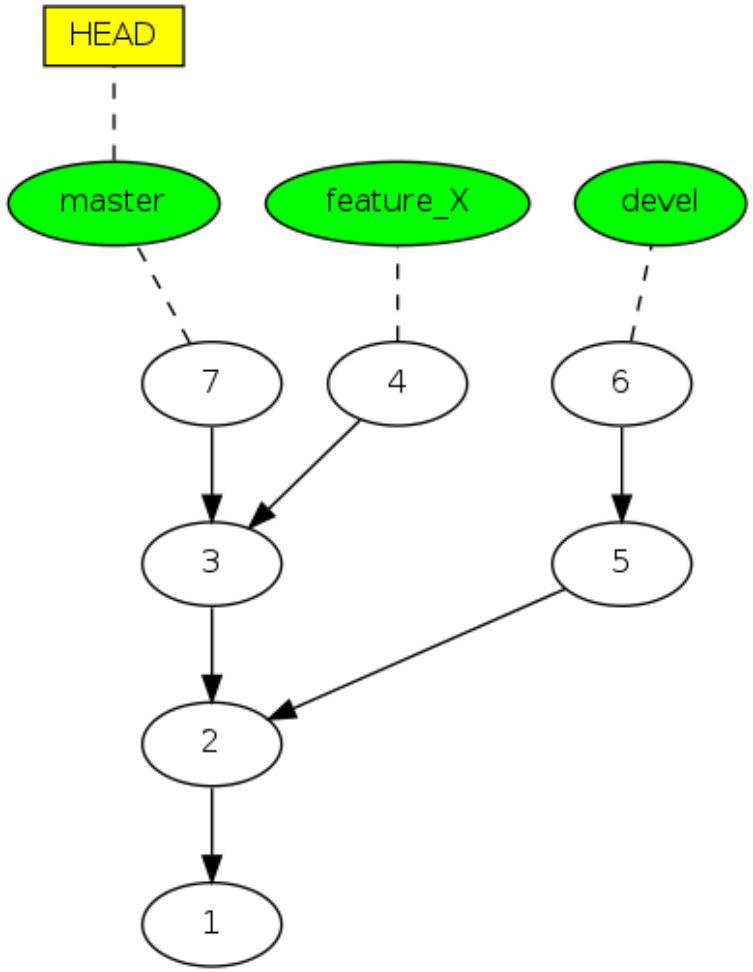
```
git branch -d feature_X
```

- which gives you this :



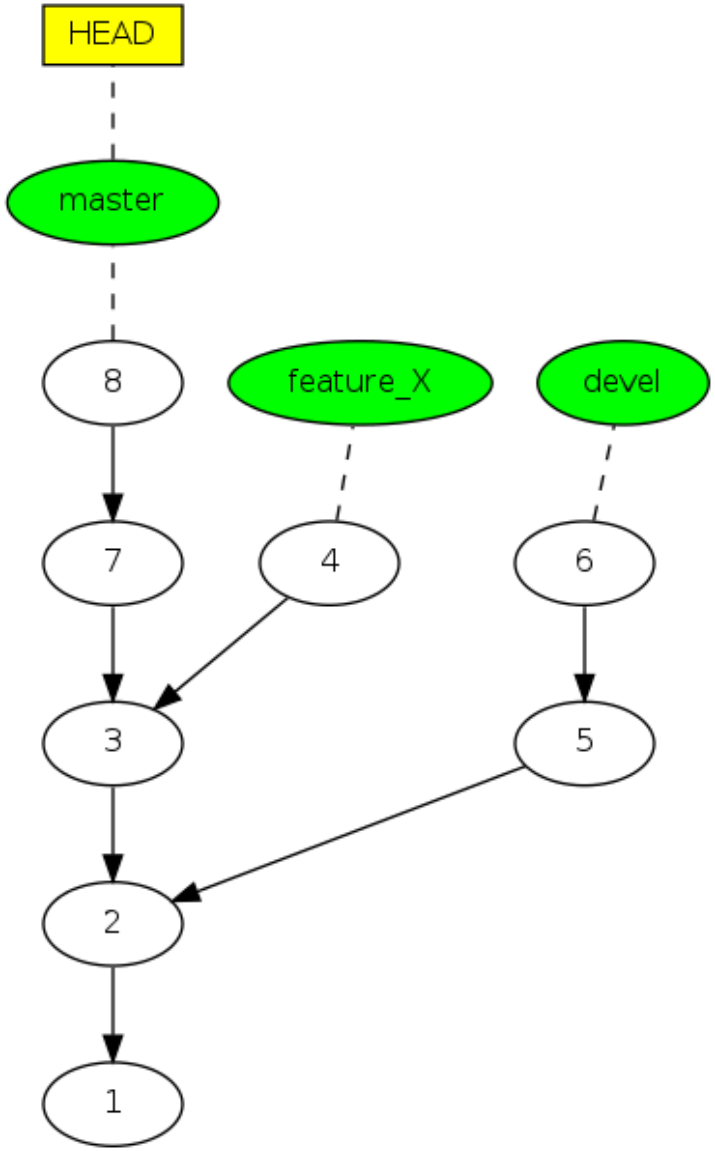
current branch/checked out branch

- There is a notion of a ‘currently checked out’ branch.
- This is denoted by a special ref called HEAD.
- HEAD is a *symbolic* ref, which points to the ‘current branch’.



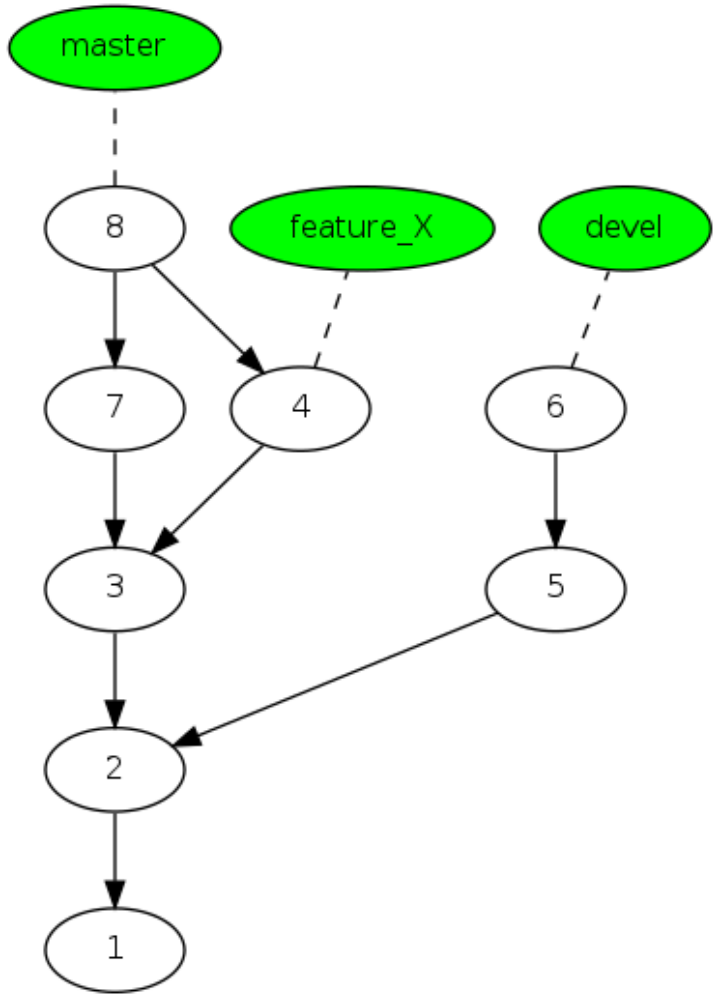
committing

- When you make a new commit, the current branch moves.
- Technically, whatever branch HEAD is pointing to will move.



naming non-leaf nodes (1/2)

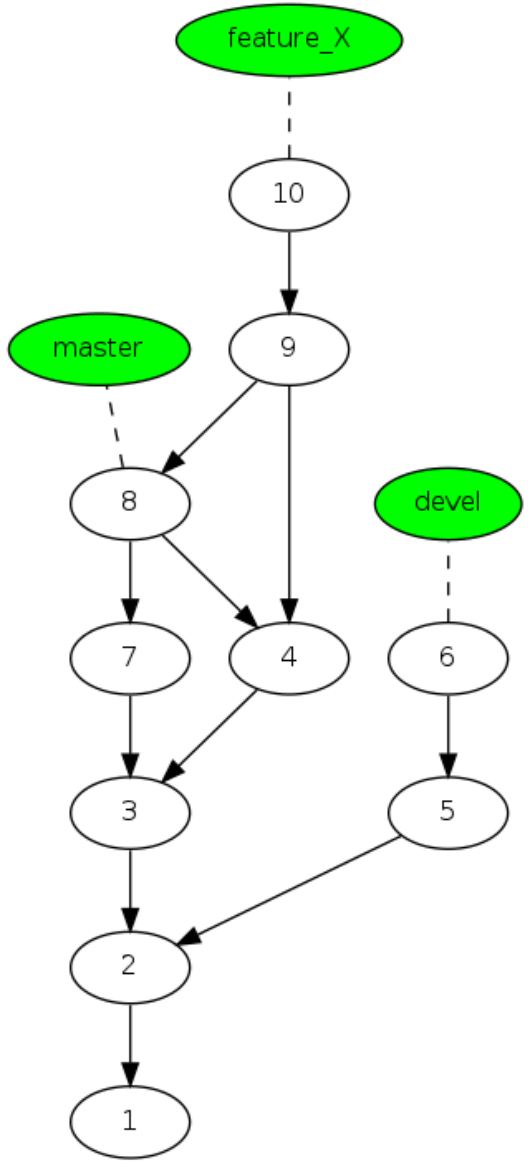
- It's not just 'leaf' nodes, but inner nodes can also have names.
- Recall the result of merging `feature_X` earlier (see the *more than one parent commit* section) :
- At this point, you could leave `feature_X` as it is forever.
- Or you could delete the branch (as we showed in that section), in which case that label would simply disappear. (The commit it points to is safely reachable from `master` because of the merge.)



naming non-leaf nodes (2/2)

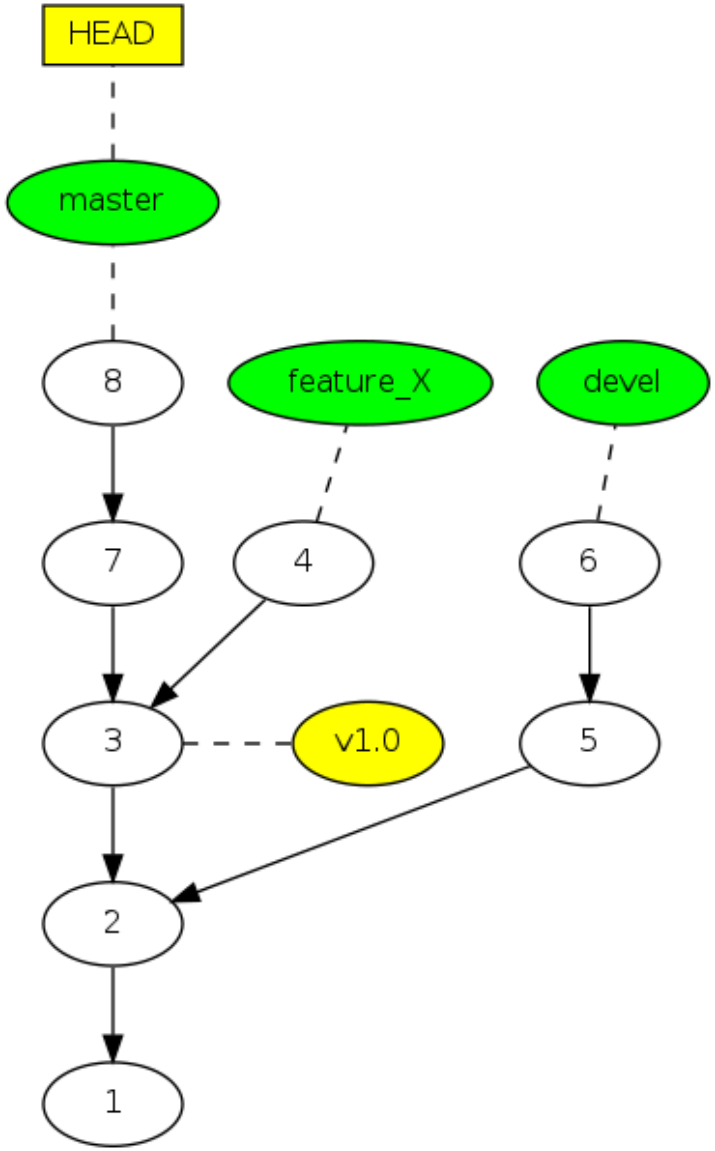
- You can also continue to develop on the `feature_X` branch, further refining it with a view to once again merging it at some later point in time.

Although not relevant to the topic of this document, I should mention that the usual practice is to first merge master back into `feature_X` to make sure it has all the other stuff that master may have acquired till now (this is shown by commit 9 below) before continuing further development :



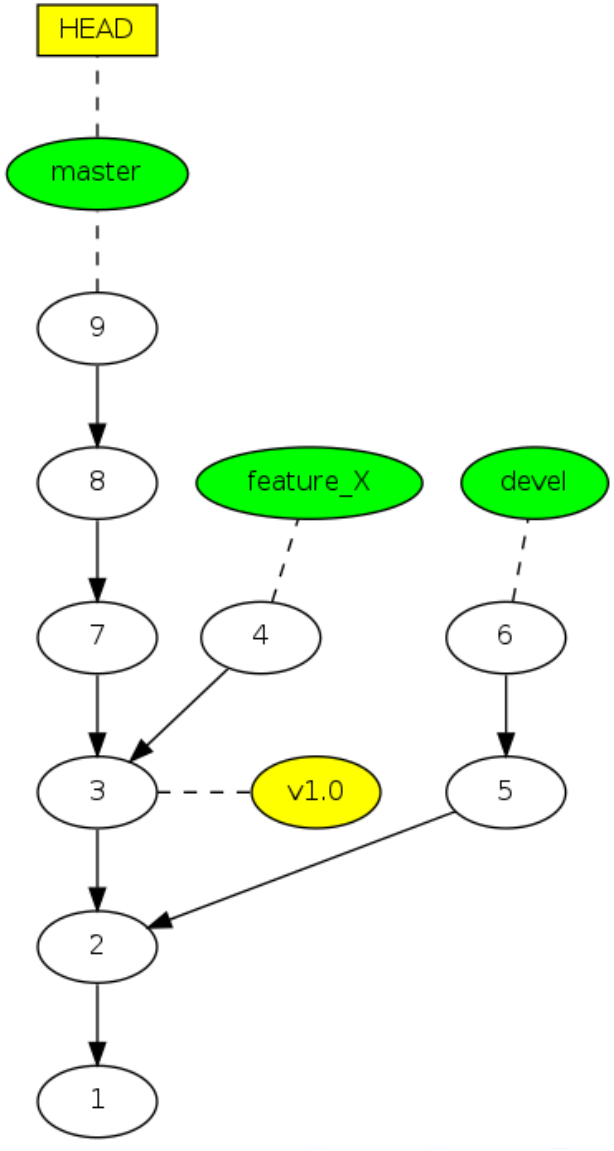
tags

- More commonly, inner nodes are TAGS.
- We show tag names in yellow circles.



the difference between branches and tags

- The main difference between a branch and a tag is branches move, tags don't.
- When you make a commit with the "master" branch currently checked out, master will move to point to the new commit.



what is a git URL ?

- Git repos are accessed by providing a URL.
- There are typically 4 kinds of Git URLs :
  - ssh : like `ssh://[user@]host.xz[:port]/path/to/repo.git/`
  - http : like `http[s]://host.xz[:port]/path/to/repo.git/`
  - git : like `git://host.xz[:port]/path/to/repo.git/` – note that this is an unauthenticated protocol suitable only for allowing downloads of open source or similar software
  - local file : like `file:///full/path/to/reponame`

(see 'man git-clone' for all the allowed syntaxes for git URLs).

what is a "remote" ?

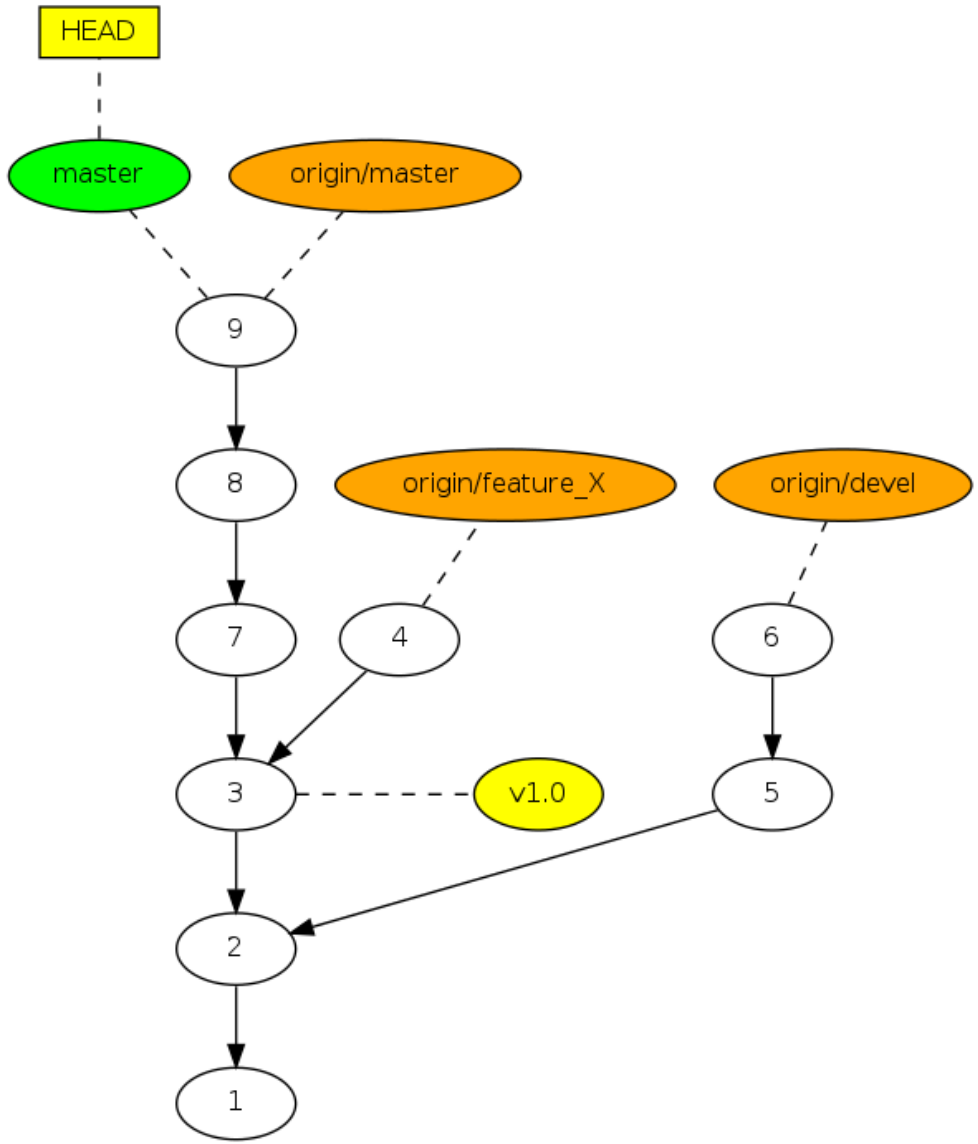
- A remote is a short name (like an alias) used to refer to a specific git repository.
- Instead of always saying

  ```
  git fetch git://sitaramc/gitolite
  ```

  you can add that as a remote and use that short name instead of the long URL.
- For convenience, a 'remote' called 'origin' is automatically created when you clone a repo, pointing to the repo you cloned from.

remote branches

- Git is a distributed version control system.
- So when you clone someone's repo, you get all the branches in that one.
- Remote branches are prefixed by the name of the remote, and we show them in orange.



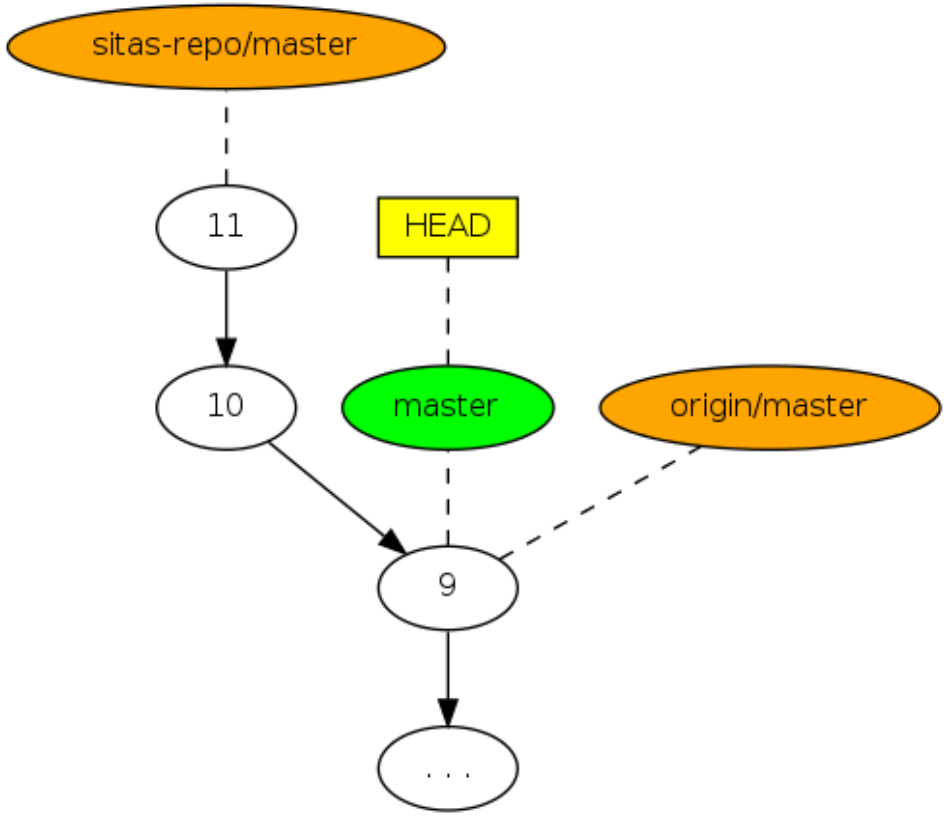
multiple remotes

You can have several remotes.

fetching and merging from another repo (1/2)

Now let's say Sita's repo had a couple of new commits on its master, and you run

```
git fetch sitas-repo
```



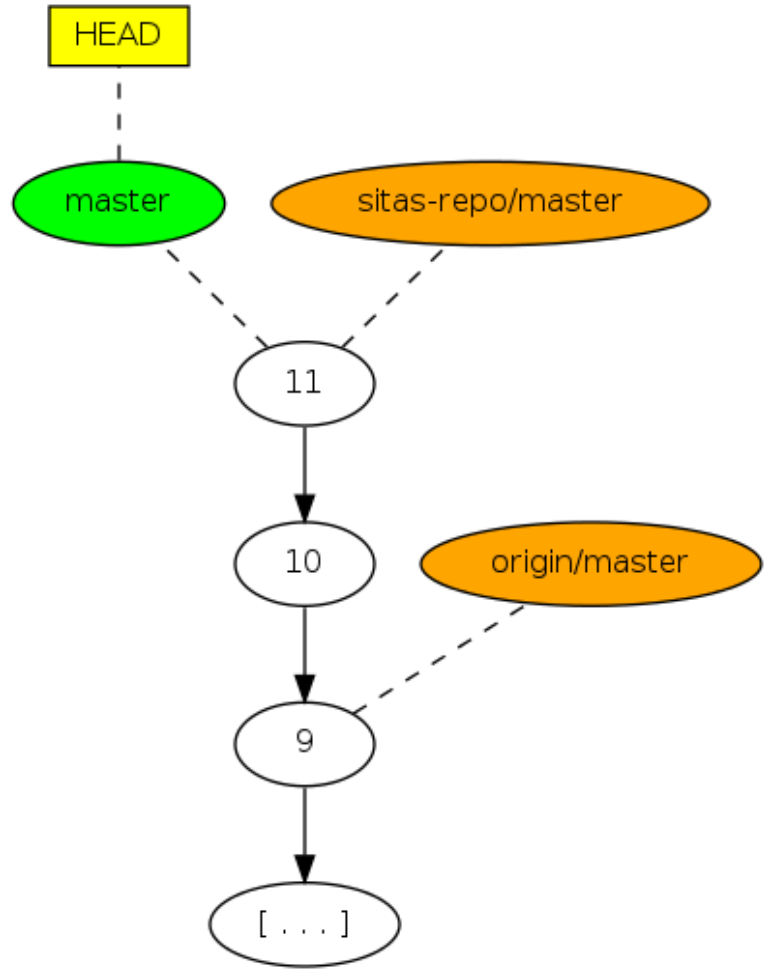
(We have pruned the graph a litle for clarity, showing only the relevant commits; the rest of the commits and branches are assumed to be present as in the previous picture).

fetching and merging from another repo (2/2)

- Now you want to merge Sita's master branch into yours.
- Since your master does not have any commits that Sita's master doesn't have (i.e., Sita's master is like a superset of yours), running

```
git merge sitas-repo/master
```

will get you this :



the object store

- Git stores all your data in an "object store".
- There are 4 types of objects in this store : files (called "blobs"), trees (which are directories+files), commits, and tags.
- All objects are referenced by a 160-bit SHA.

(Details, if you like : a blob is the lowest in the hierarchy. One or more blobs and trees make a tree. A commit is a tree, plus the SHA of its parent commit(s), the commit message, author/committer names and emails, and timestamps. Under normal usage, you don't need to deal with all this).

what is a repo (again)

- Earlier, we saw that a repo was a graph of commits.
- At the file system level, however, it is basically a directory called `.git` which looks somewhat like this

```
$ ls -al .git
total 40
drwxrwxr-x 7 sitaram sitaram 4096 Sep 14 18:54 ./
drwx------ 3 sitaram sitaram 4096 Sep 14 18:54 ../
drwxrwxr-x 2 sitaram sitaram 4096 Sep 14 18:54 branches/
-rw-rw-r-- 1 sitaram sitaram   92 Sep 14 18:54 config
-rw-rw-r-- 1 sitaram sitaram   73 Sep 14 18:54 description
-rw-rw-r-- 1 sitaram sitaram   23 Sep 14 18:54 HEAD
drwxrwxr-x 2 sitaram sitaram 4096 Sep 14 18:54 hooks/
drwxrwxr-x 2 sitaram sitaram 4096 Sep 14 18:54 info/
drwxrwxr-x 4 sitaram sitaram 4096 Sep 14 18:54 objects/
drwxrwxr-x 4 sitaram sitaram 4096 Sep 14 18:54 refs/
```

objects and branches/tags (1/4)

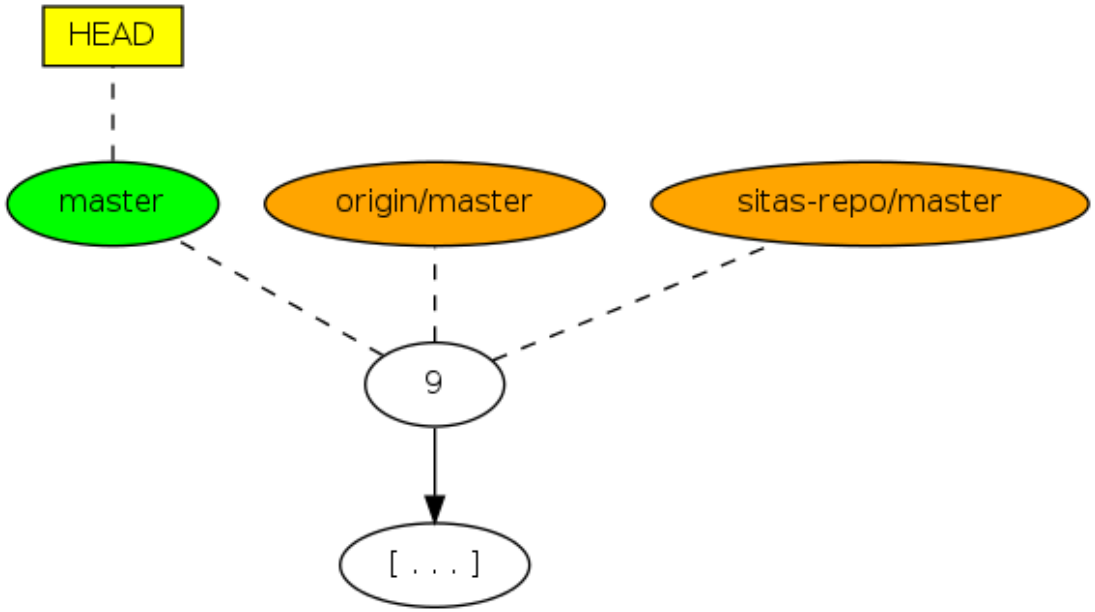
- Hg folks should read this section carefully.
- Among various crazy notions Hg has is one that encodes the branch name within the commit object in some way. Unfortunately, Hg's vaunted "ease of use" (a.k.a "we support Windows better than git", which in an ideal world would be a negative, but in this world sadly it is not) has caused enormous takeup, and dozens of otherwise excellent developers have been brain-washed into thinking that is the only/right way.
- I hope this section gives at least a few of them a "light-bulb" moment.

objects and branches/tags (2/4)

- The really, *really* important thing to understand is that the object store doesn't care where the commit came from or what "branch" it was part of when it entered the object store. Once it's there, it's there!
- Think back to these three diagrams.

objects and branches/tags (3/4)

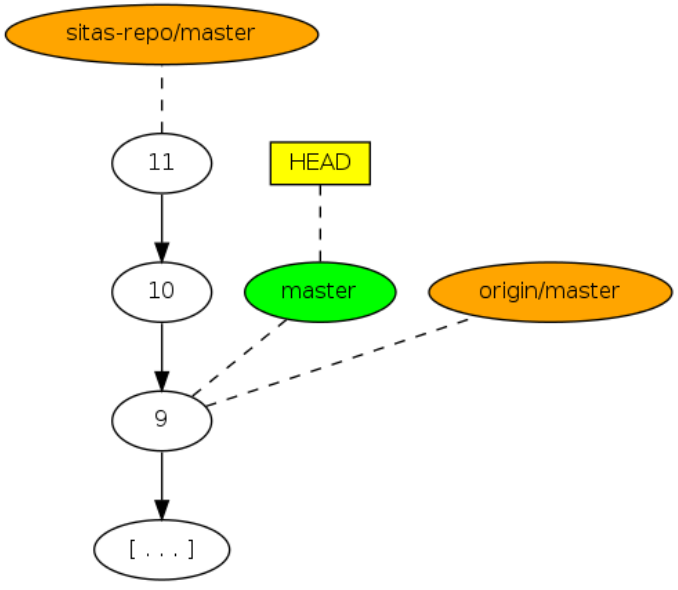
The first is before you did a fetch.



The next two figures are after `git fetch sitas-repo` and `git merge sitas-repo/master`, respectively.

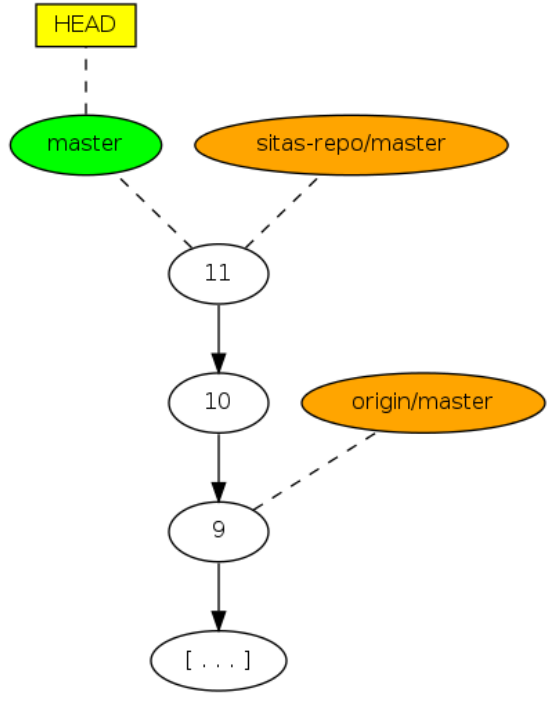
objects and branches/tags (4/4)

`git fetch sitas-repo`



The fetch command added two new commits (10 and 11) to your object store, along with any other objects those commits reference.

`git merge sitas-repo/master`



However, note that commits 10 and 11 did not change in any way simply because they are now in *your local "master" branch*. They continue to have the same SHA values and the object store does not change as a result of this command at all.

All you did was move a pointer from one node to another.

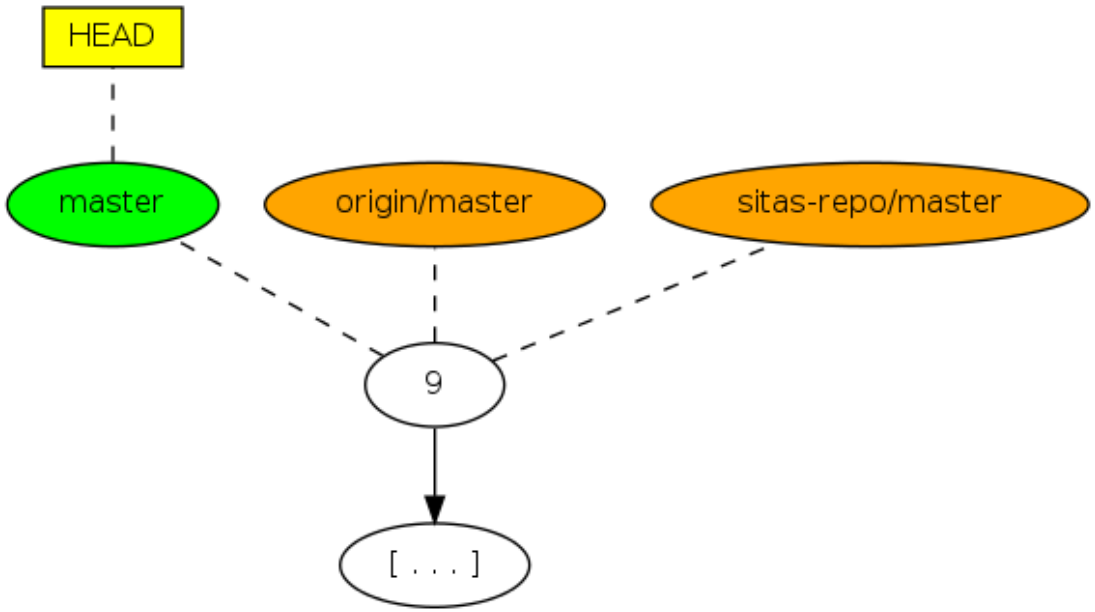
We'll now show some advanced operations with the aid of this same tree.

merging (1/3)

First, let's do merging.

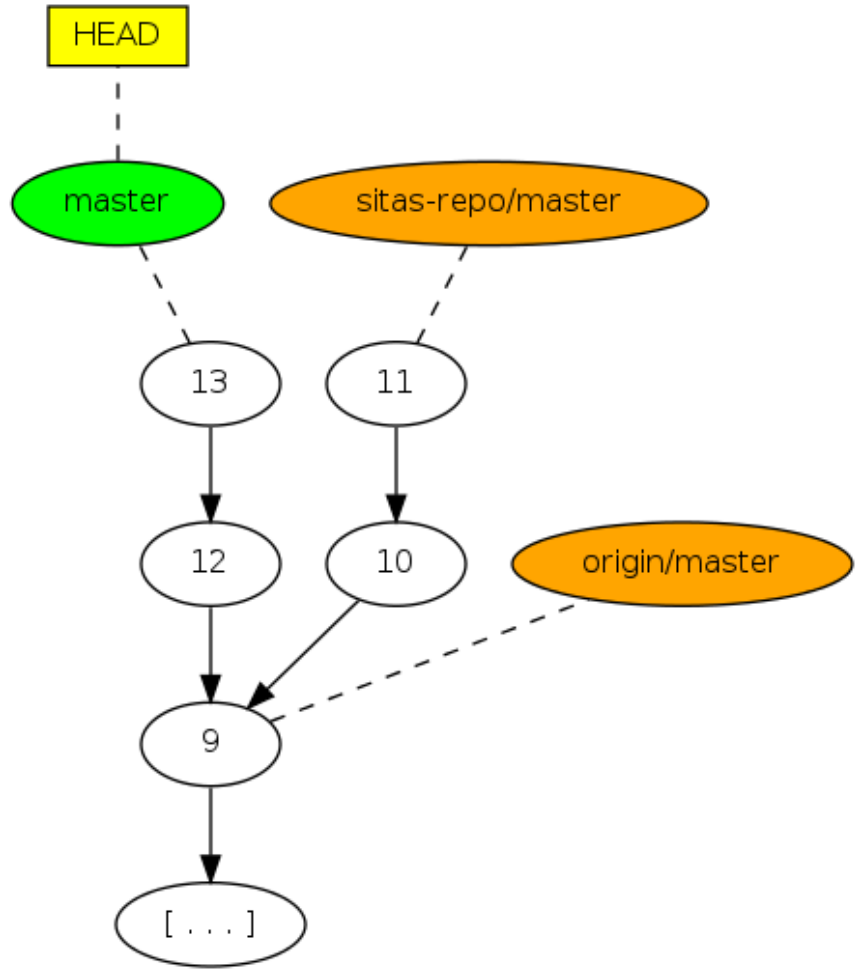
- The merge you saw earlier was what is called a "fast-forward" merge, because your local master did not have any commits that the remote branch you were merging did not have.
- In practice, this is rare, especially on an active project with many developers.
- So let's see what that looks like.

The starting point was this :



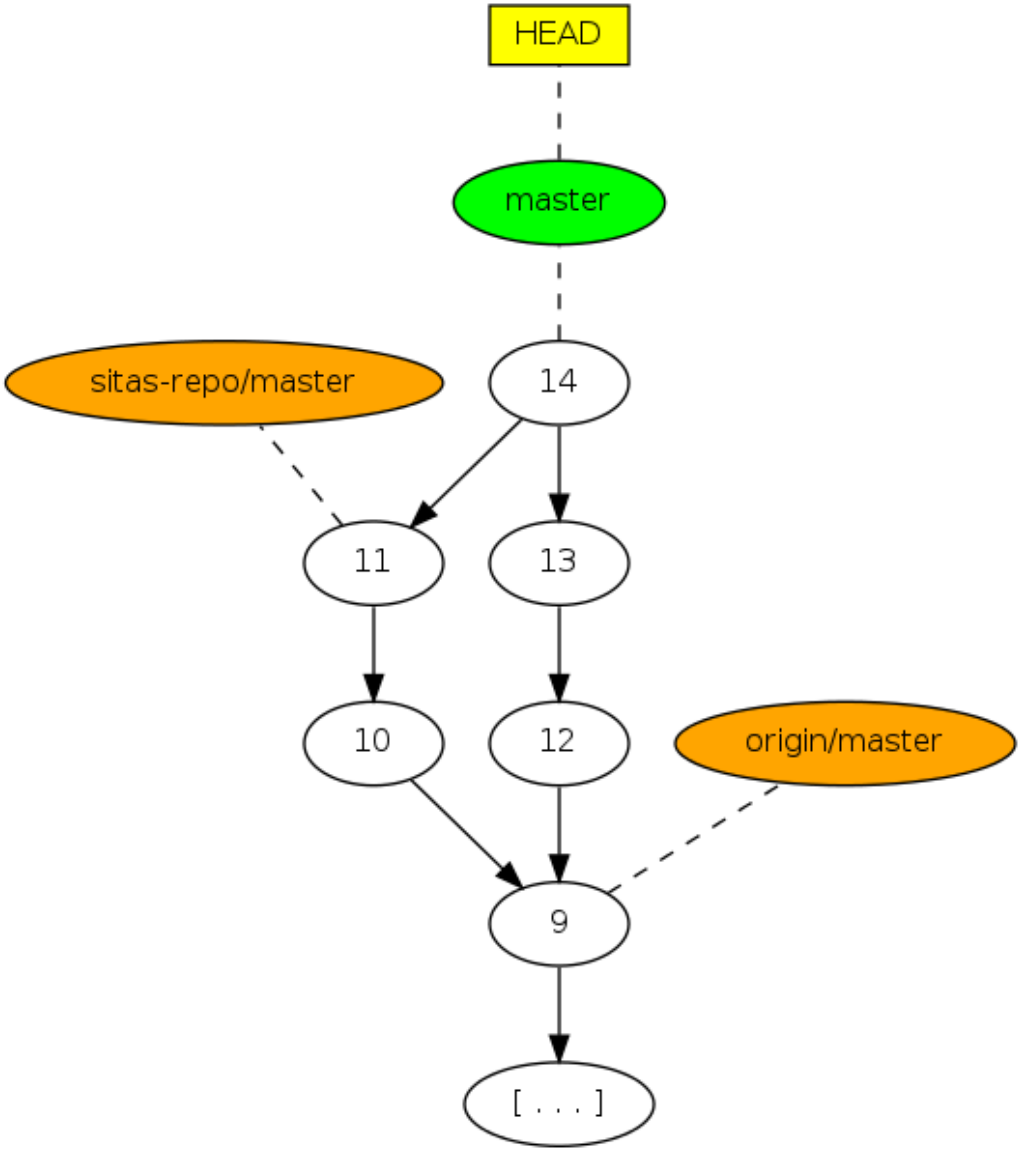
merging (2/3)

- Now, you made some changes on your local master.
- Meanwhile, sitas-repo has had some changes which you got by doing a fetch :



merging (3/3)

When you merge, the end result will usually look like this :

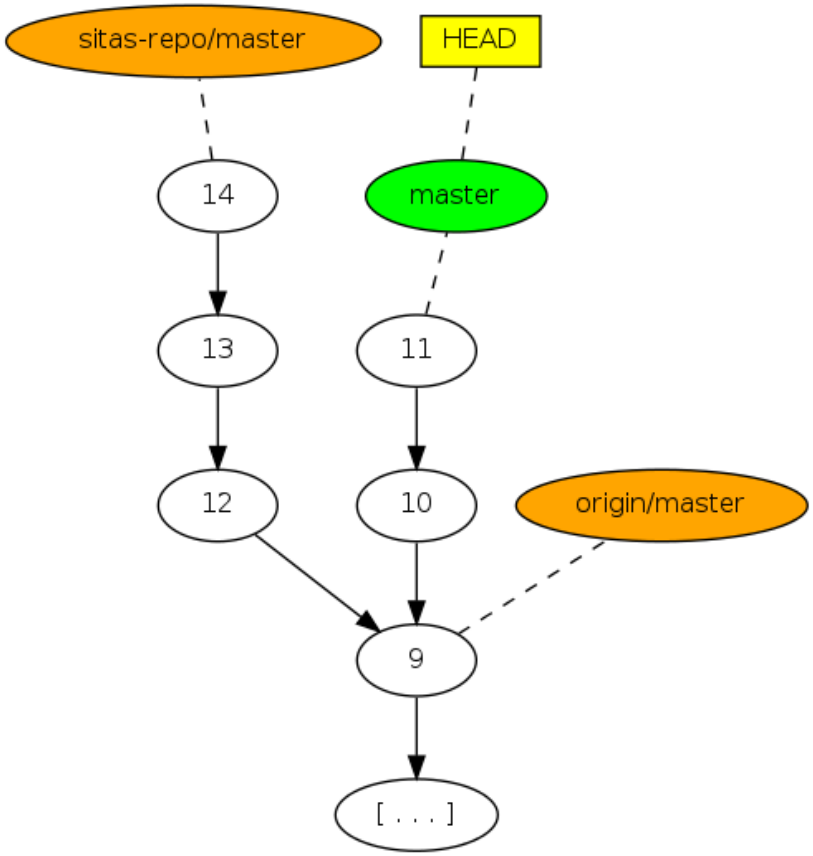


cherry-pick (1/3)

- A cherry-pick is not very commonly done – in well designed workflows it should actually be rare.
- However, it's a good way to illustrate an important concept in git.
- We said before that a commit represents a certain set of files and directories, but since most commits have only one parent, you can think of a commit as representing a set of changes too. (In fact, most older VCSs do this).

cherry-pick (2/3)

- Let's say one of your collaborators (this mythical "Sita" again !) made a whole bunch of changes to his copy of the repo.
- You don't like most of these changes, except one specific change which you would like to bring in to your repo.
- The starting point is this :

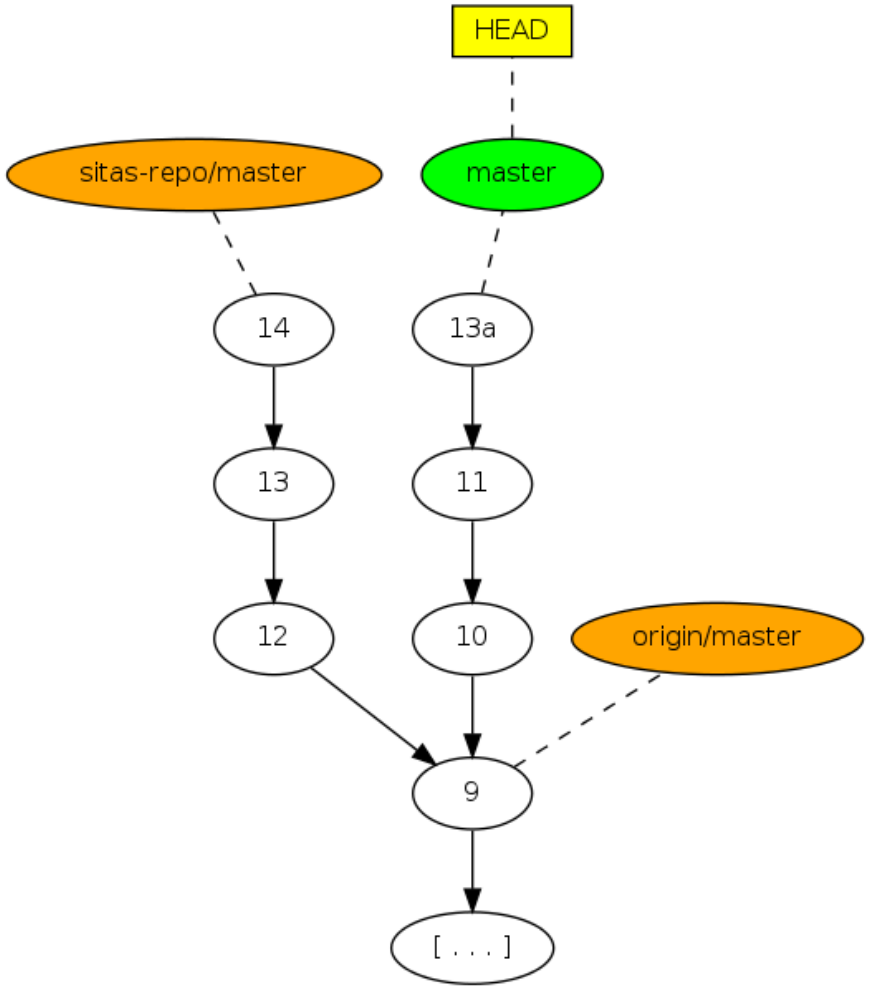


In this, sitas-repo has 3 commits on his master (12, 13, and 14) and you are only interested in the change that commit 13 made.

```
git cherry-pick sitas-repo/master~1
```

This results in the following commit graph.

- Don't worry about the meaning of the ~1 for now (although you ought to be able to guess !)
- Note that I've called the new commit "13a". This is to reflect the fact that, while the **change** made is the same as in the original commit 13, the **SHA** will not be the same anymore (new parent commit, new "tree", new committer name/email, commit time, etc).

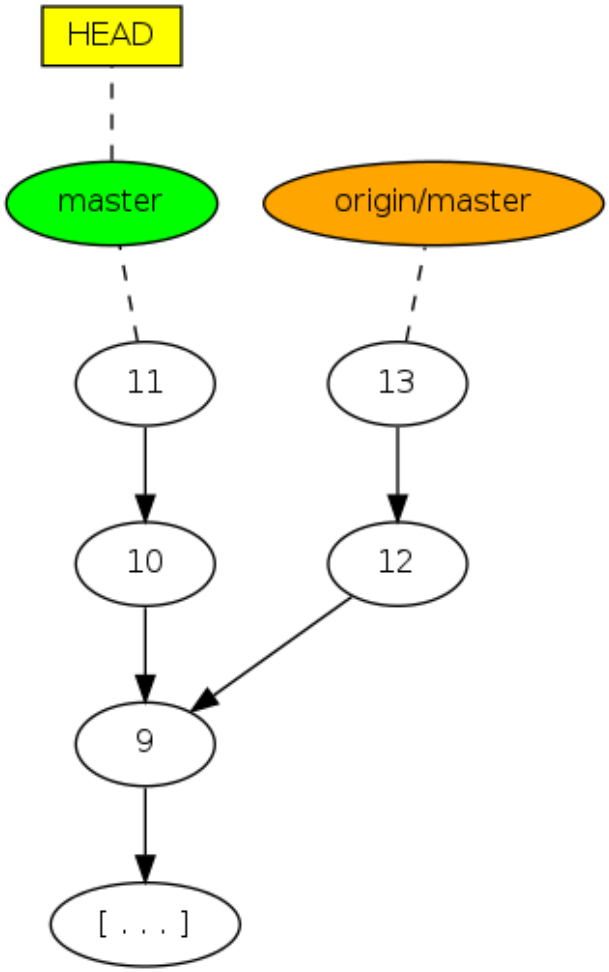


rebasing (1/4)

- Instead of merging, let's say you wanted to rebase your commits on top of Sita's commits.
- First of all, what is rebasing ?
- It's basically transplanting a series of changes from one point in the graph to another point.
- So if you guessed that a rebase was (in principle) a series of cherry-picks, you'd be pretty close, at least from a concept point.

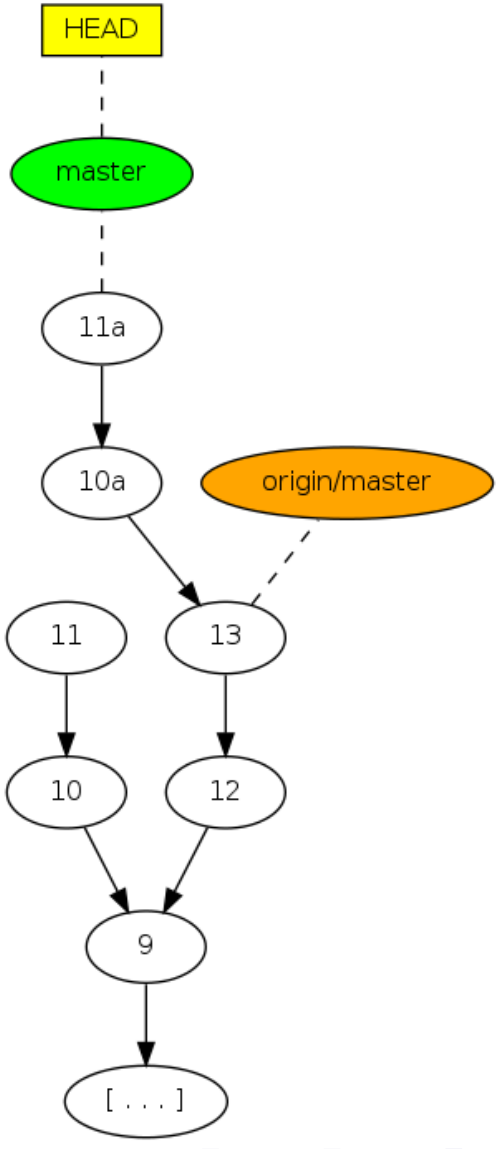
rebasing (2/4)

- So let's use a similar example as in the merge example before, but instead of sitas-repo, the new commits are in "origin" (which is the "main" server for this project).
- You had your own commits, and you did a `git fetch origin` which brought in the latest commits from "origin", so it looks like :



rebasing (3/4)

- Now, instead of merging "origin/master" into your local master, you want to rebase your commits on top of "origin/master".
- That is, you want to pretend your local changes were made **after** commit 13 on the origin.
- So you run `git rebase origin/master`, and this is the result :

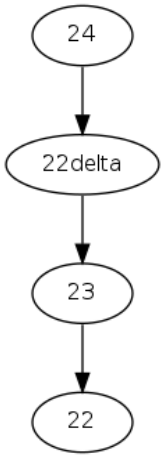


Note that again, we're ignoring command syntax and nuances here. This is about concepts.
Also again, note that the SHAs of the 2 commits have changed, since they now have new parents, trees, etc., so we represent that by suffixing an "a".
Note the **dangling commits** 10 and 11. No branch is now pointing to them, so they're basically wasted disk space. (They can be examined and reclaimed using the 'reflog', or if left alone they will eventually get garbage collected).

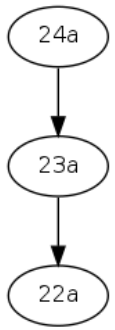
rebasing (4/4)

- Unlike cherry-pick, a rebase is quite often done in real life.
- Rebase also has some other forms.
- This form is one, but the most common is when a developer wants to re-arrange his own local commits in a more logical sequence before publishing/pushing them.

I often do the eqvt of changing this :



where “22delta” is a minor fixup to “22”, into



using `git rebase -i`.

- Notice that since commit 22 changes its SHA, all its child commits – now rebased – will also have new SHAs.
- This is why you should (almost) never rebase branches that have already been published.

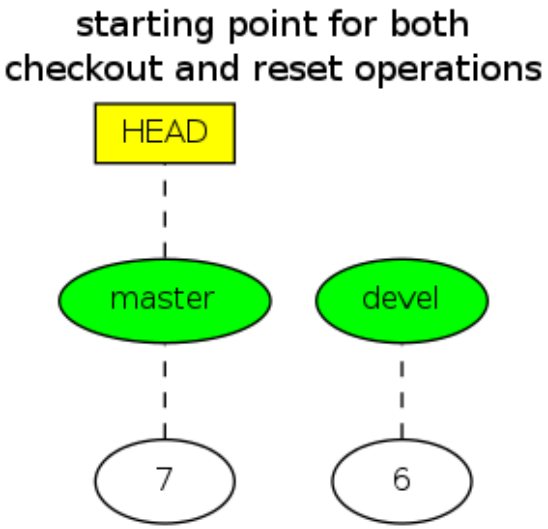
the confusion about checkout versus reset

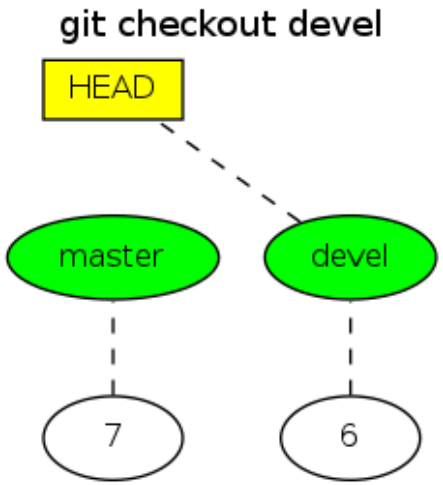
Please note that this section completely ignores the myriad options available to both these commands, especially the ones that pertain to the index and the working tree. All we're trying to do is show what the commands do to the branch and `HEAD`, nothing else. As such, this is not a complete discussion of those two commands, but only about one aspect of them.
In fact, the major reason these two sometimes get confused is that people eventually learn that `git checkout -f` and `git reset --hard` do the same thing, and then extrapolate that to other options. Just remember that both those are specific cases of two quite different commands that just happen to "meet" there, in some sense.

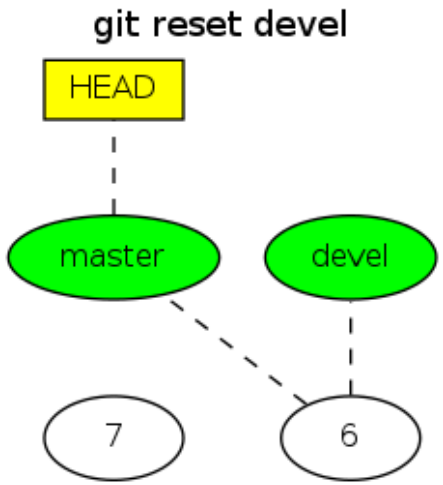
The basic difference is very simple :

- checkout changes what your current branch is (i.e., it changes where HEAD is pointing to)
- reset changes which commit the current branch is pointing to

- Here're some pictures that show you what's happening. (Note that commits below 6 and 7 are omitted for brevity).
- The first picture is the common starting point.
- The next two show the effect of a checkout and a reset, respectively.
- Notice which line is moving in each picture, compared to the starting point.





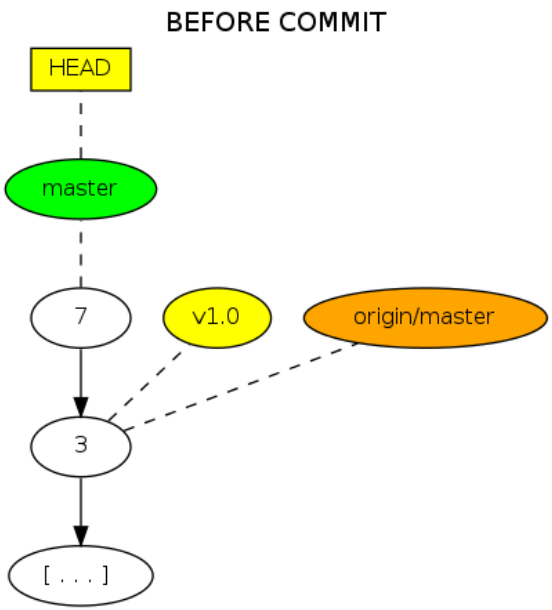


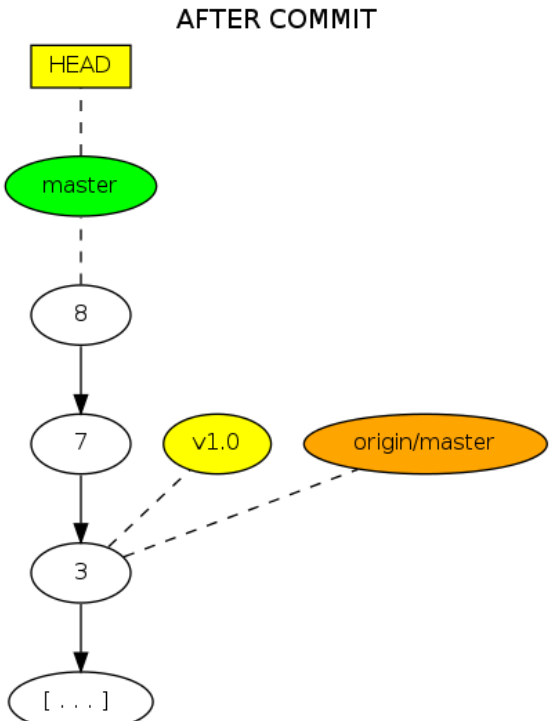
- As you can see, your current branch, when you start, is "master" (indicated by where HEAD is pointing).
- A "checkout" changes what is your current branch – it is now "devel", and so any new commits you make now will go on devel, not master.
- On the other hand, a reset changes what commit your current branch points to. Your branch is still master, but now it is pointing to what could potentially be a **completely** different history.

(Notice that, like in the rebase example, there is now a dangling commit – one that cannot be reached by any branch. It is still accessible using the 'reflog' and will eventually get garbage collected)

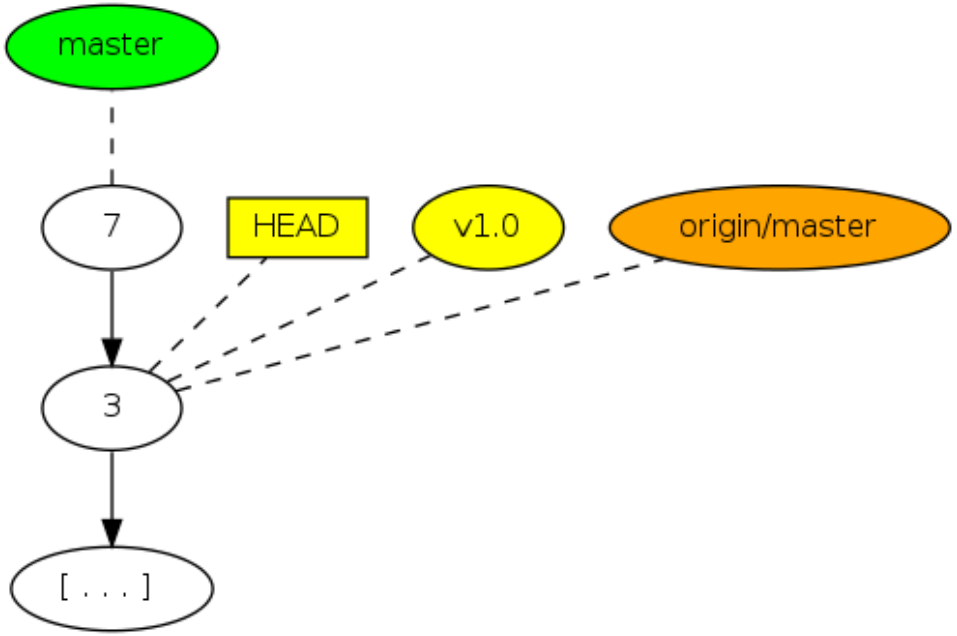
detached HEAD and all that

- All this time you have seen HEAD pointing to a branch name, and the branch itself pointing to a commit.
- Then, when you make a commit, the branch moves to the new commit (although HEAD still continue to point to the branch name).
- If you want to recap, here's a look at just the part of the tree that concerns us, in a simple before/after display :





Now see what happens when you `git checkout v1.0` :



Notice the subtle difference ?

- HEAD is no longer a *symbolic* ref (i.e., pointing to a real branch).
- Instead, it is pointing directly to a commit.

- In real terms, this is literally what happens.
- When you had "master" checked out, the contents of HEAD (it's just a file in .git) are simply

```
ref: refs/heads/master
```

- If you `git checkout devel` (which is a local branch), it becomes

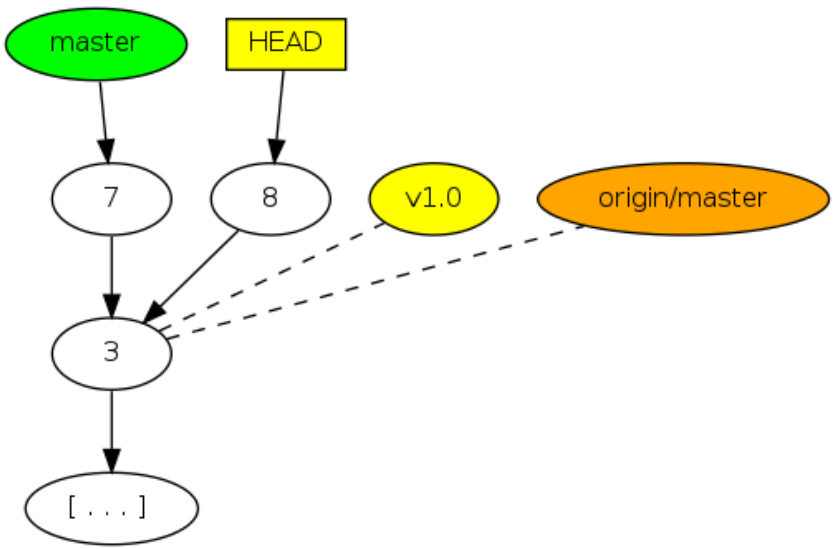
```
ref: refs/heads/devel
```

- However, remember what we said earlier ?
- Only branches can move, tags cannot.
- So when you `git checkout v1.0`, HEAD now contains

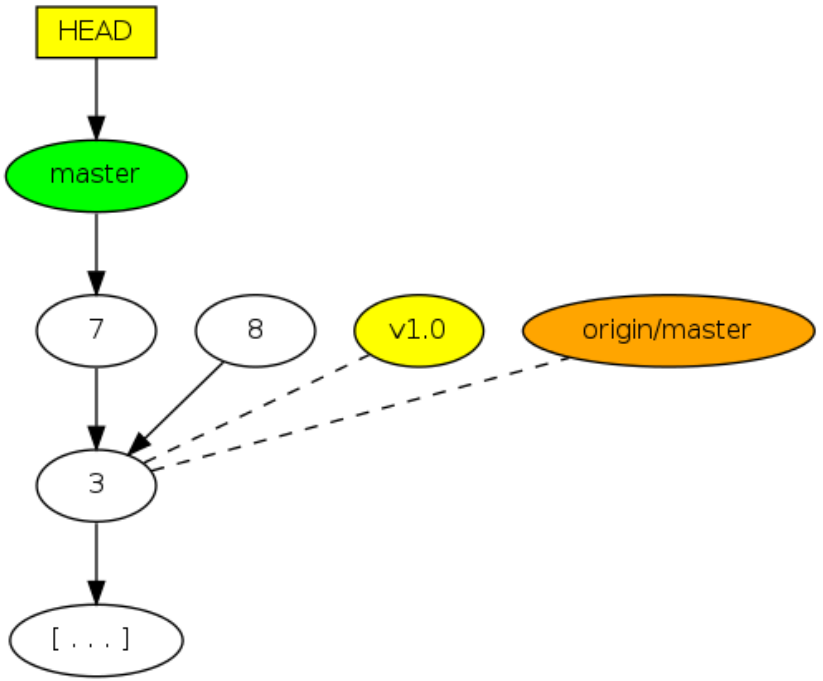
```
90fed7792746a9a33e24059fb171f6bbb6ffebe6
```

  or some such hash.
- If it did what it did for local branches, it would imply the tag would move, right ?
- Which it shouldn't – wouldn't be much of a tag if it moved !

So from this point on, if you make a commit, only HEAD changes, nothing else, since it is no longer "attached" to any local branch name"



The reason this is considered dangerous is that, if you now do `git checkout master`, you get this :



- Notice what happened to your new commit 8 ?
- It's dangling. Unreachable.

(Except by the reflog, of course, but beginners can get shaken up !)

other ways to detach your HEAD

- The most common way to get into a detached HEAD state is to check out a remote branch (like `git checkout origin/branch`), without realising that you need to make a local copy before you make commits on it.
(Side note : the correct syntax to create and checkout a local branch starting from a remote branch used to be `git checkout -t -b branch origin/branch`, but modern gits will do the right thing if you simply say `git checkout branch` ; yaay for progress !)
- The reason `git checkout origin/branch` creates a detached HEAD is that, while remote branches can move, they can only do so as a result of a `git fetch` or equivalent. After all, they are meant to track what the **remote** has, so it wouldn't make sense for them to acquire **local** commits !
- Here are various ways to detach HEAD :

```
git checkout origin/master  # (described above)
git checkout master^        # parent of master
git checkout HEAD~2         # grandparent of current HEAD
git checkout tagname        # since you cant commit to a tag!
git checkout <SHA>          # hex digits forming a full or partial SHA
git checkout master^0       # (see note below)
```

- These will all make the file called `HEAD` contain the actual (40-hex-digit) SHA of the corresponding commit instead of some string like `ref: refs/heads/branch`.
- That last one (`git checkout master^0`) is interesting. The `master^0` notation means "the actual commit that master points to", so it's just like saying `git checkout <SHA>`.

re-attaching the HEAD

- Modern git will warn you about losing commits on a detached HEAD if you're at the command line, and tell you how to recover (immediately).
- Recovery is quite easy. If you realised you're on a detached HEAD **before** you switched to some other branch :

```
git checkout -b newbranch
```

- If you switched, possibly made some commits, and **then** realised you lost some commits on a detached HEAD, you need to check the reflog to find the lost HEAD and switch to that using some command like :

```
git branch newbranch HEAD@{1}
```

the reflog

- The reflog can show you all the values that HEAD has taken in the past.
- Here's a simulated example.
- I cloned a repo that had a master and a 'foo' branch. I then checked out `origin/foo`, ignored all the warnings, and made two commits on what I thought was the "foo" branch. Then I switched back to master and made two normal commits on master before I realised that I had 'lost' two commits.
- At this point, running `git reflog show` gives me the following (the most recent HEAD value is first) :

```
86a8ee0 HEAD@{0}: commit: my second commit on master
e42c4d0 HEAD@{1}: commit: my first commit on master
58c1539 HEAD@{2}: checkout: moving from e2558a9d1527e5a76b39ffede1dc5ca9c650de01 to master
e2558a9 HEAD@{3}: commit: second commit on foo
1c9dfa0 HEAD@{4}: commit: first commit on foo
802f184 HEAD@{5}: checkout: moving from master to origin/foo
58c1539 HEAD@{6}: clone: from /tmp/tmp.lv5IqjK0ZI/b
```

- That `moving from <SHA>...` is usually a sign that some commits may have been lost.
- In this case you can run `git branch newbranch HEAD@{3}` or `git branch newbranch e2558a9` to save those commits.

(Notice that the SHA value of `HEAD@{3}` is the same one mentioned in the `moving from <SHA>...`

message on the line above it).

-
-
- However, of necessity, there are code examples within those documents. I believe that the principle of fair use should cover use of those snippets ; see especially factors 3 and 4 in the list of factors here.
- However, if you're not convinced that it would be fair use, then you may consider those code snippets, as well as associated "comments" if any, to be under the GPLv2 license.