Continuation of the git Demonstration

The log command shows past commits, give hash tag for each.

Leave `123.txt` untracked

Edit `ABC.txt` to add a line. Similarly, edit `123.txt` to add a line.

Stage `ABC.txt` but not `123.txt` and then commit

`git log` (Observe hash tag and committal message.)

`git checkout`, to roll back to the previous commit. Observe the contents of the two files.
Checkout should only be used when all tracked files are unmodified (are unedited since most recent commit)
Examples:

```
Git checkout master^      # rolls back all present edits to the previous commit
Git checkout master       # rolls forward to the "tip" of the master (undoes previous checkouts that roll back)

Git checkout master^^     # rolls back two commits.
```

Explain: Don't want to re-write history. If you want to fetch something from an older commit, copy it out to an untracked file (best: but that untracked file in another directory) then return to the tip of the master and edit the changes you want back into the repository.

1

Creating a Branch

Here is a scenario: You have a commit that represents a real software product now being sold.
You wish to create a new version of the software to include new features but you need to keep the old version and perhaps you will need to update that too in the future.

The technique: Create a new branch.

Let sequential letters of the alphabet represent commits.
As the original software program is developed the sequence of commits can be visualized as a sequence in time.

master branch → A . . . . B . . . . C . . . . D . . . . E . . . . F
time→

Here, "F" is the final commit that is being sold as a product. You do not want to lose access to "F"
You could always continue editing and then do checkouts to roll back to F, but this is super awkward. Instead. . .

Create a branch.

But first, the concept of the HEAD.

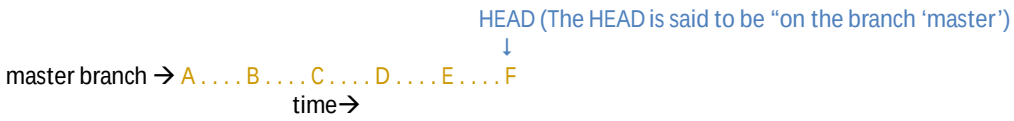
2

The concept of the HEAD in git

HEAD is an alias within .git for a structured variable.
The HEAD is shorthand for "the current branch" and "the current commit."
All files are edited and manipulated with respect to the HEAD.

Up to this point in our demonstration HEAD has always pointed to the "master" branch (so far, the only branch)
Except during the demonstration of the checkout command, HEAD also pointed to the most recent commit.
The folder view presented to you via the operating system is always a view of the HEAD (plus any changes subsequent to the last commit).

Adding a notation for the HEAD shows us that our sequence of commits looks like:

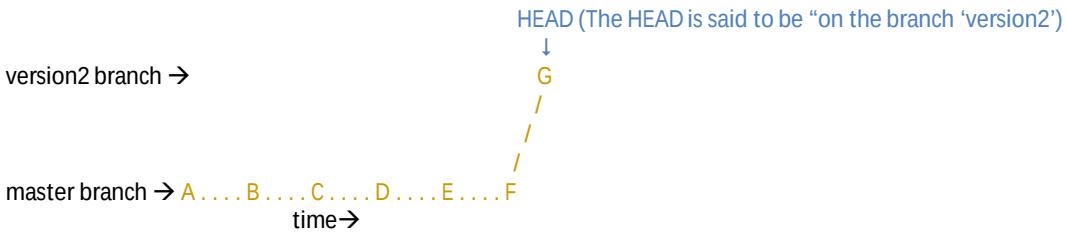


Now, create a branch off the master.

3

Branching in git

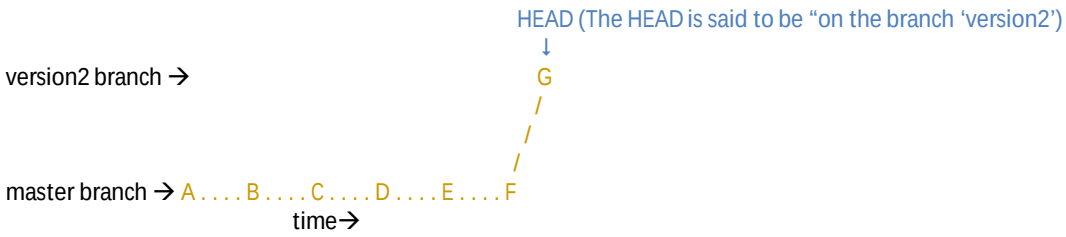
```
git branch version2
```



The branch command creates a named commit from another commit. (G created from F here)
Now new edits may be made to create the new version of the software in the version2 branch.
These may be staged and committed as usual, but these actions only happen on the "version2" branch.

4

Branching in git



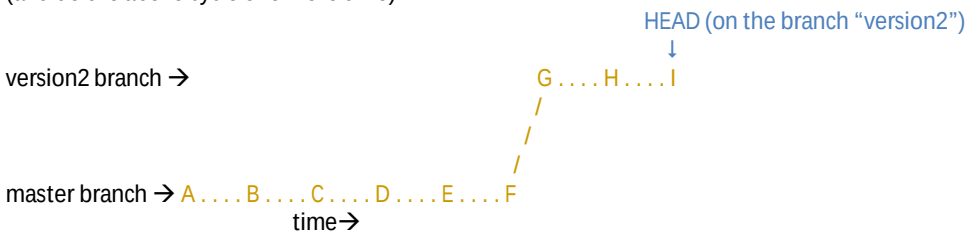
Suppose work commences on developing version2 of the software.

5

Branching in git
(Various edits made to the files using tools under the supervision of the OS.)

```
git add *
git commit
```

(and do the above cycle one more time)

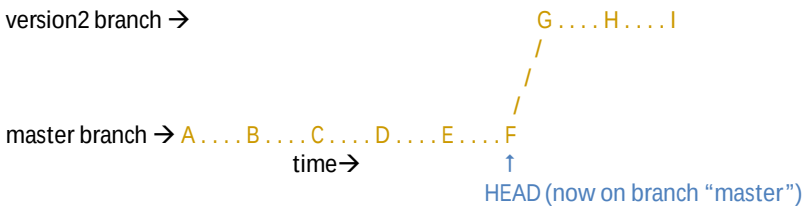


Now suppose a bug is found in the software that is for sale at the moment and we want to fix it.
It will be necessary to roll back to the commit at F. This commit happens to be the *tip* of the master branch.
Tip = A commit (or a merge) at the end of a branch.
In the illustration above F is the tip of the "master" branch and I is the tip of the "version2" branch.
We need to checkout the master branch.

6

Branching in git

```
Git checkout master
```

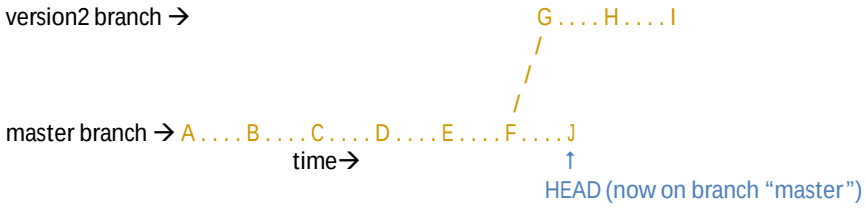


Perform the bug fix by using an editor or IDE or any other tool under the supervision of the OS.
When the fix is ready, stage it and commit it.

7

Branching in git
(various edits via tools under the supervision of the OS)

```
git add *
git commit
```

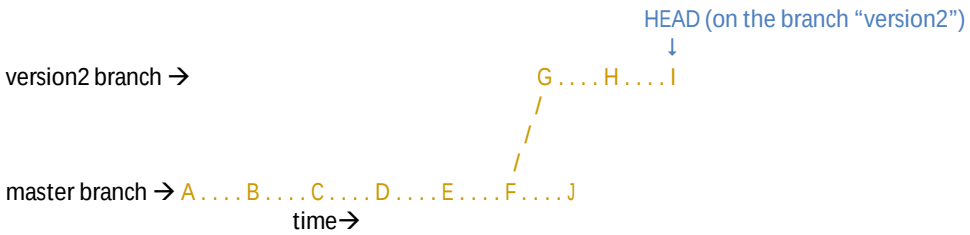


And with that job done, we return to working on the new version.

8

Branching in git

```
git checkout version2
```

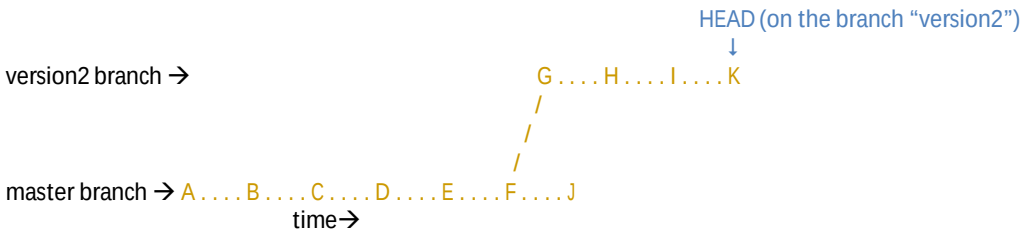


Probably need to fix the same bug there too.

9

Branching in git
(various edits via tools under the supervision of the OS—fix the same bug that was fixed in the master branch)

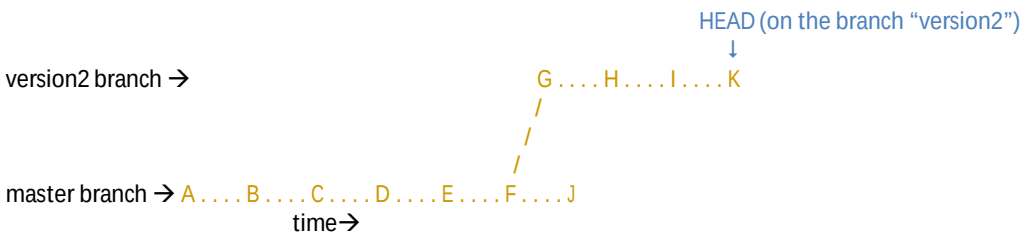
```
git add *
git commit
```



The project can now go on with two versions of the software, each independently editable.

10

Branching in git



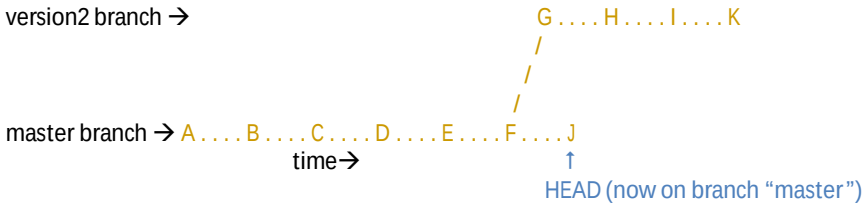
Suppose the "version2" branch becomes so well developed that it is going to be released to the public and the original version is going to be depreciated (no longer supported).

We want to go back to the master branch and merge the new version into it. We do this in two steps.

11

Branching in git

```
git checkout master
```

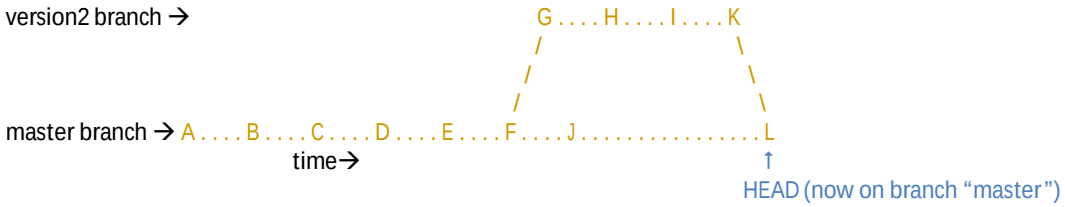


Now merge into the current branch.

12

Branching in git

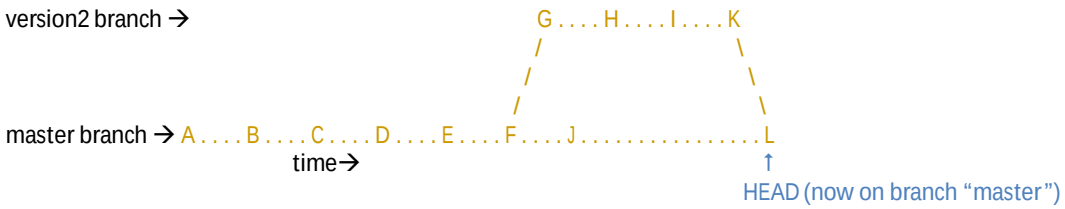
```
git merge version2
```



The "version2" branch no longer exists!
Also, what was the version2 branch no longer has a tip. K is not the end of the branch and L is not on the branch.
Differences between the files in version 2 and master have been "merged" in a semi-automatic manner.
Further development goes along the master branch. This development may contain new branches.

13

Detaching the HEAD from a tip

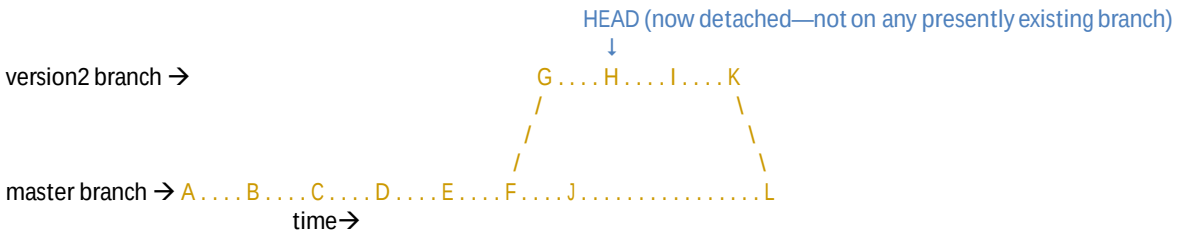


Suppose for some reason somebody wants information about the very first edits made in the version 2 branch.
But version2 no longer exists! (Because it no longer has a tip.)
One can use `git log` to find the hash for commit H (The hash is typically a long and random-looking string)
Then checkout commit H via the hash.

14

Detaching the HEAD from a tip

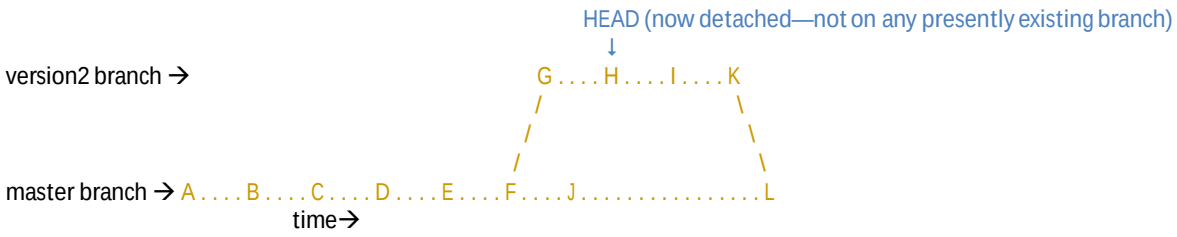
`git log` (find hash for H in the text output of this command)
`git checkout <hash for commit H>`



Now you can inspect all the files just as they were after commit H.
Any information needed can be copied out to an untracked file, and then later edited into the current branch if so desired.
If you edit files with the head detached all the edits will be discarded at the next checkout.
If you commit with a detached head you create a new un-named branch.
The unnamed branch can be found via its hash—but that's inconvenient. It would be better to start a new branch.

15

Re-attaching the HEAD

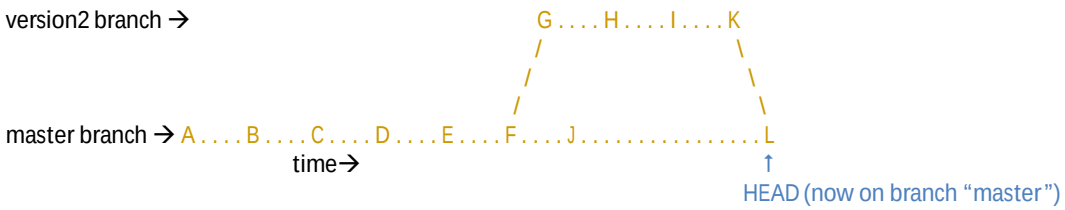


One could observe files, collect information, and then return to the tip of any branch with a checkout command.

16

Re-attaching the HEAD

```
git checkout master
```

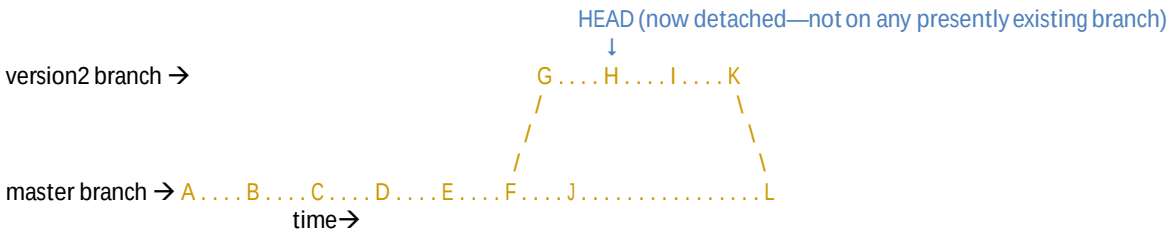


Now ready to resume work on the master branch.

Or. . . Going back to the detached head situation. . .

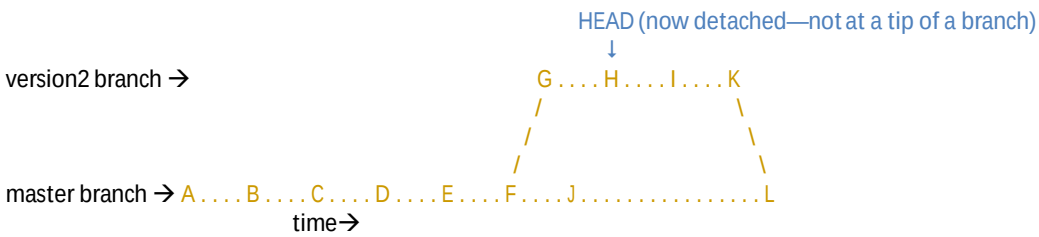
17

Detaching the HEAD from a tip



18

Detaching the HEAD from a tip

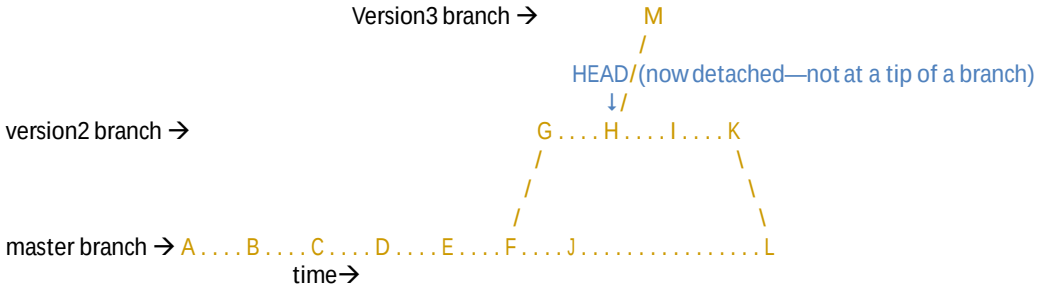


One could create a new branch from the present HEAD.

19

Detaching the HEAD from a tip

```
git branch version3
```

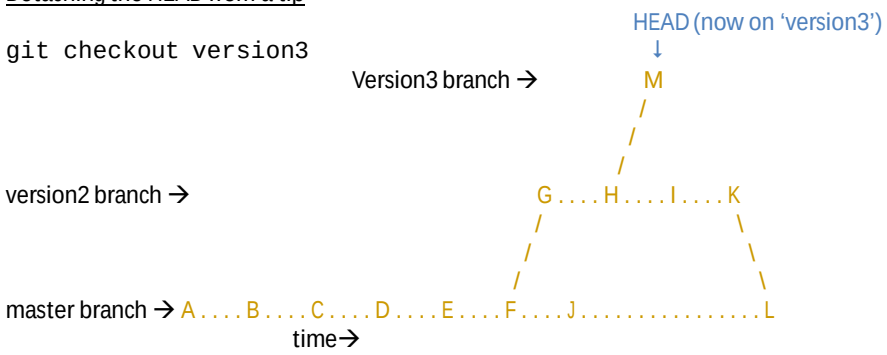


One could create a new branch from the present HEAD.
Since the HEAD is detached and the version2 branch no longer has a tip, the branch instruction did not move the head to the tip of the new branch.
But now there is a branch that has the same information as commit H. One can checkout that branch.

20

Detaching the HEAD from a tip

```
git checkout version3
```



Now one could proceed to develop version3.
One could easily switch back and forth between version 3 and the master branches using the checkout command.

Q: What is the difference between checking out a commit via a hash and checking out a branch?
A: Technically, only the matter of the detached head, which prevents convenient future access.
Branches are, in a sense, named commits. The names make access convenient.
In most projects there are at least daily commits, but it is undesirable to name them all.
Have you ever taken 1000 pictures at a party? Had difficulty finding one in particular?
You should have named it when you first realized it was a keeper. But, all the snapshots are there to peruse!

21