On becoming a Git Master

By Ralph Vancampenhoudt and Celeste Willems

1. Branches
2. Merging
3. Remote Branches
4. Pull & Push revisited
5. Additional topics

Overview

Branches

Chapter 1

1. Branches
    - ✓Introduction
    - ✓Branches in Git
    - ✓Creating a branch
    - ✓Switching branches
    - ✓Deleting a branch
2. Merging
3. Remote Branches
4. Pull & Push revisited
5. Additional topics

Overview

Branches

Introduction

A **branch** can be seen as an **independent line**, **diverged** from your **main line** of development. It allows to **make changes in an isolated way**, without affecting the main line directly.

*A **branch represents** a **new working directory** and **history** for your project.*

Branches

Branches in Git

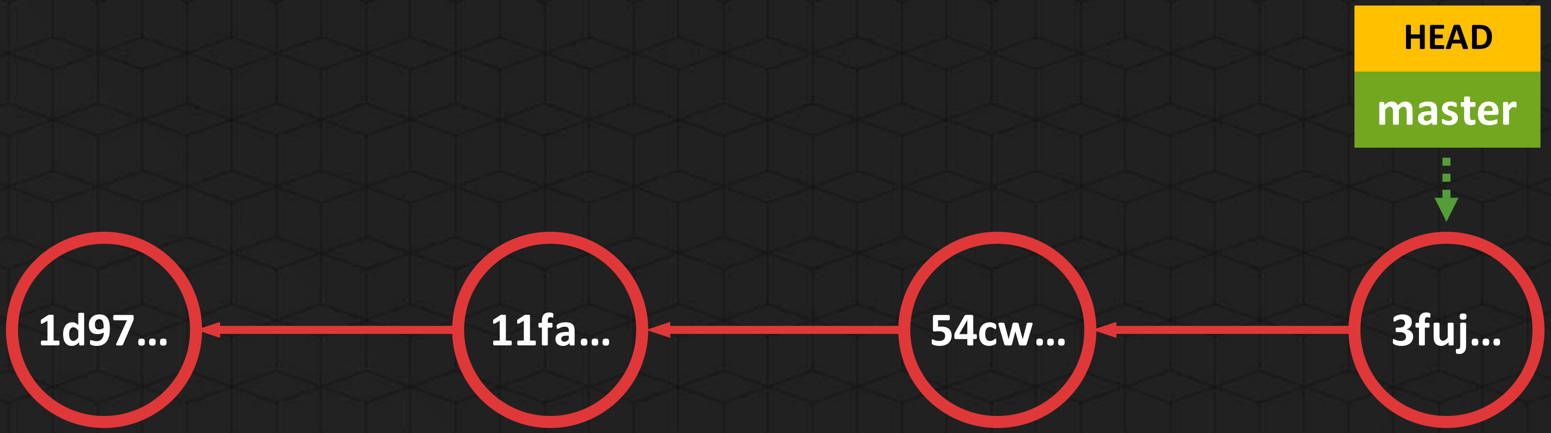
Branches in Git are **very cheap** and **lightweight**. They’re effectively **simple pointers** (references) to **commits** (snapshots).

The **master** branch is
the **default branch.**

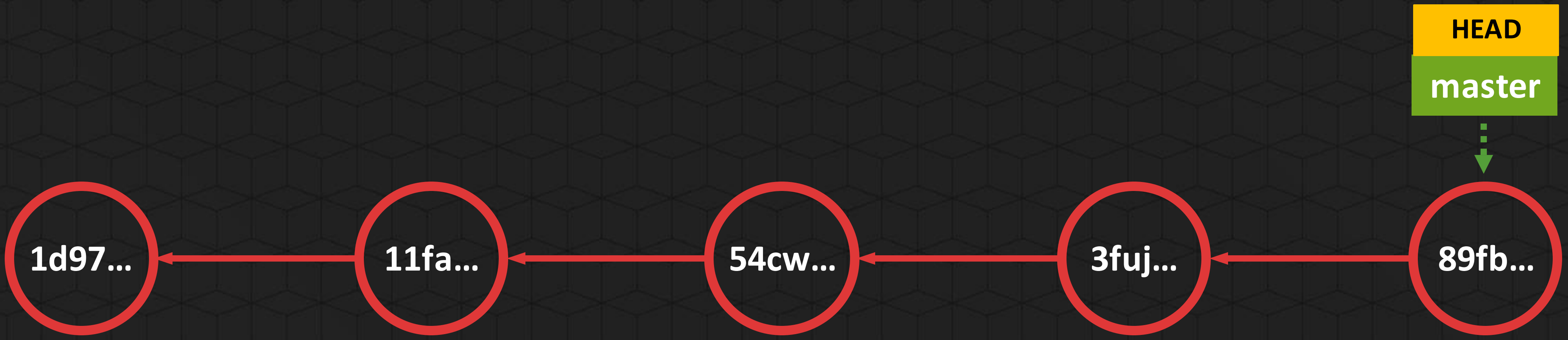
HEAD is a special pointer that **keeps track on which branch** you currently **are**.

HEAD points to the last commit of the currently checked-out branch

Currently, we're on the **master branch**.



Every time we **commit**, the **pointer** of the **branch** we're on **moves forward automatically.**



Branches

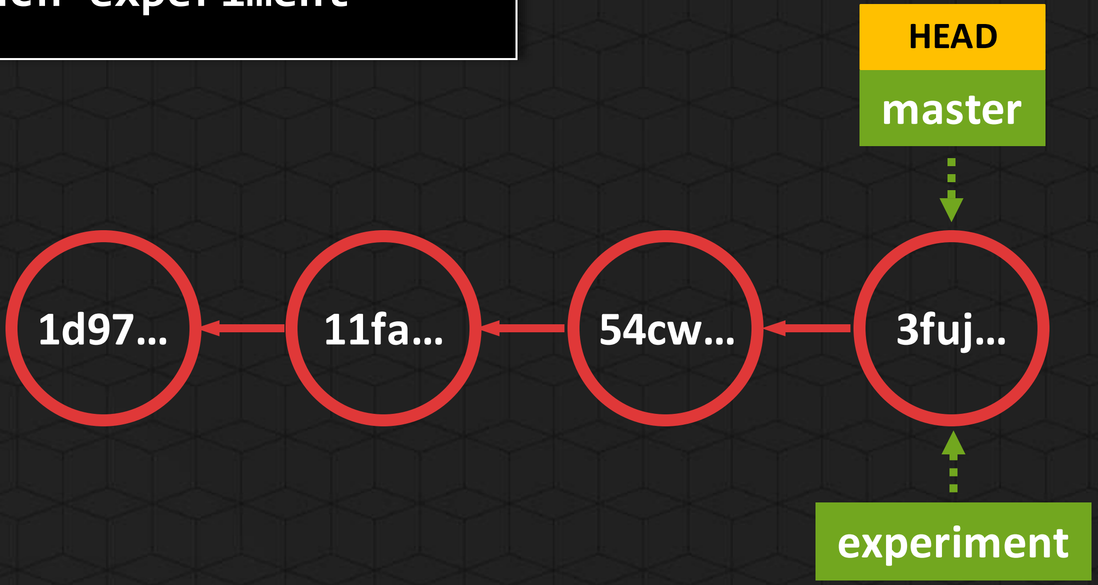
Creating a branch

Creating a branch simply **creates a new pointer** to the **current commit** you're **on**.

The commit to which the current branch you're on is pointing to.

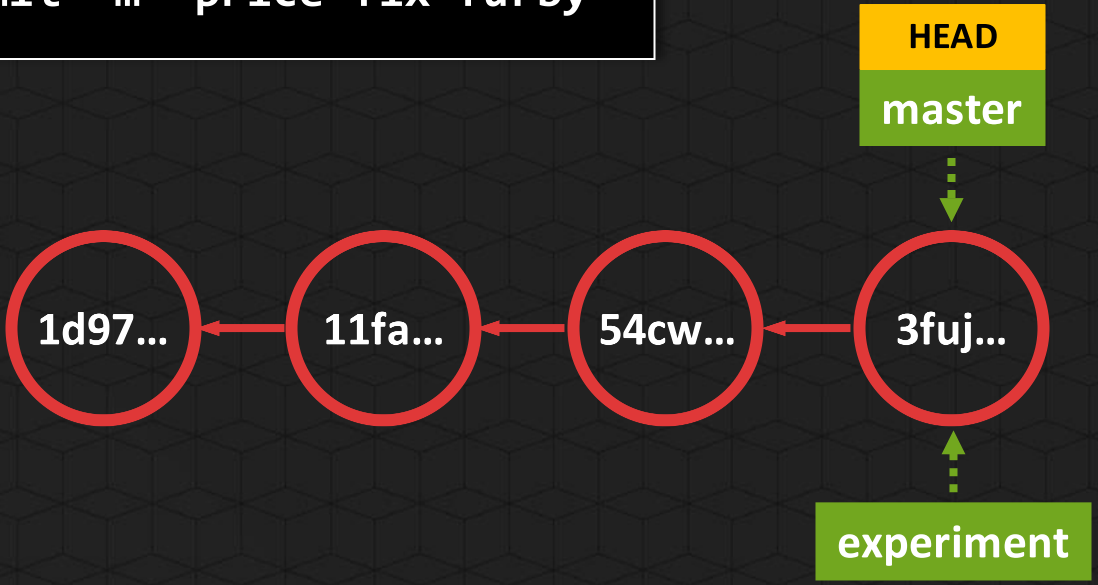
Creating a **branch** is done by the **branch command**

```
git branch experiment
```



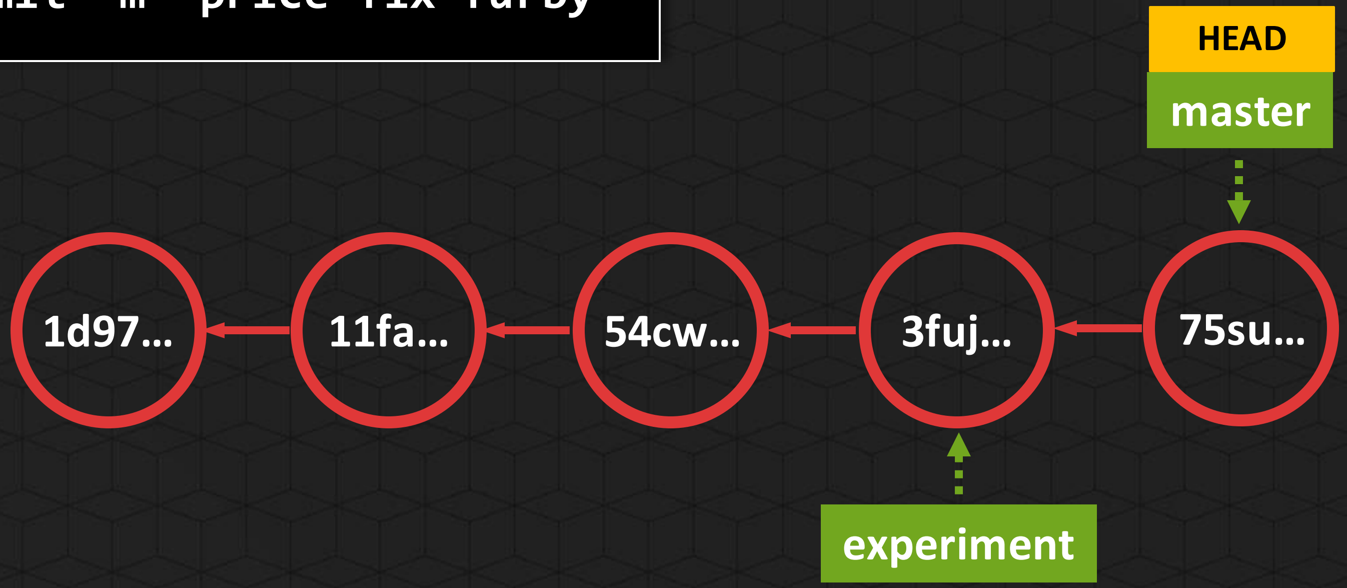
Creating a **branch** does **not automatically switch** to that branch.

```
git commit -m “price fix furby”
```



Creating a **branch** does **not automatically switch** to that branch.

```
git commit -m “price fix furby”
```



Branches

Switching branches

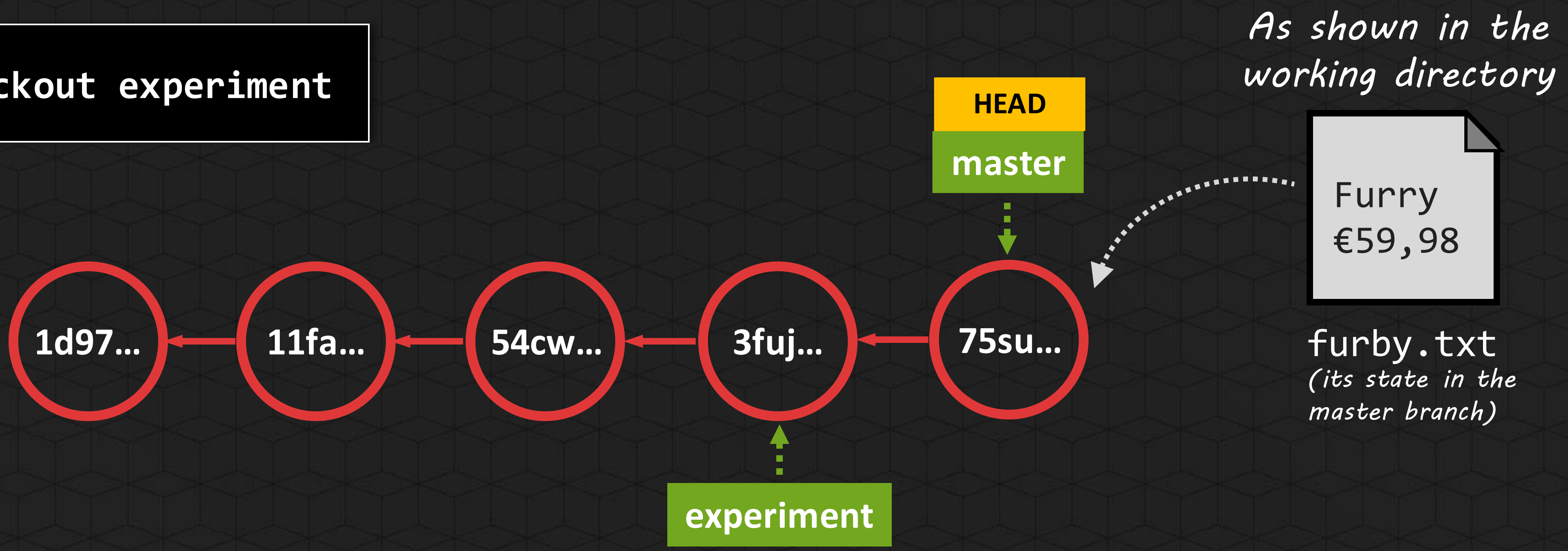
Also known as "checking out a branch"

Switching branches is done by letting **HEAD point** to the **branch** you want **to use.**

Switching branches

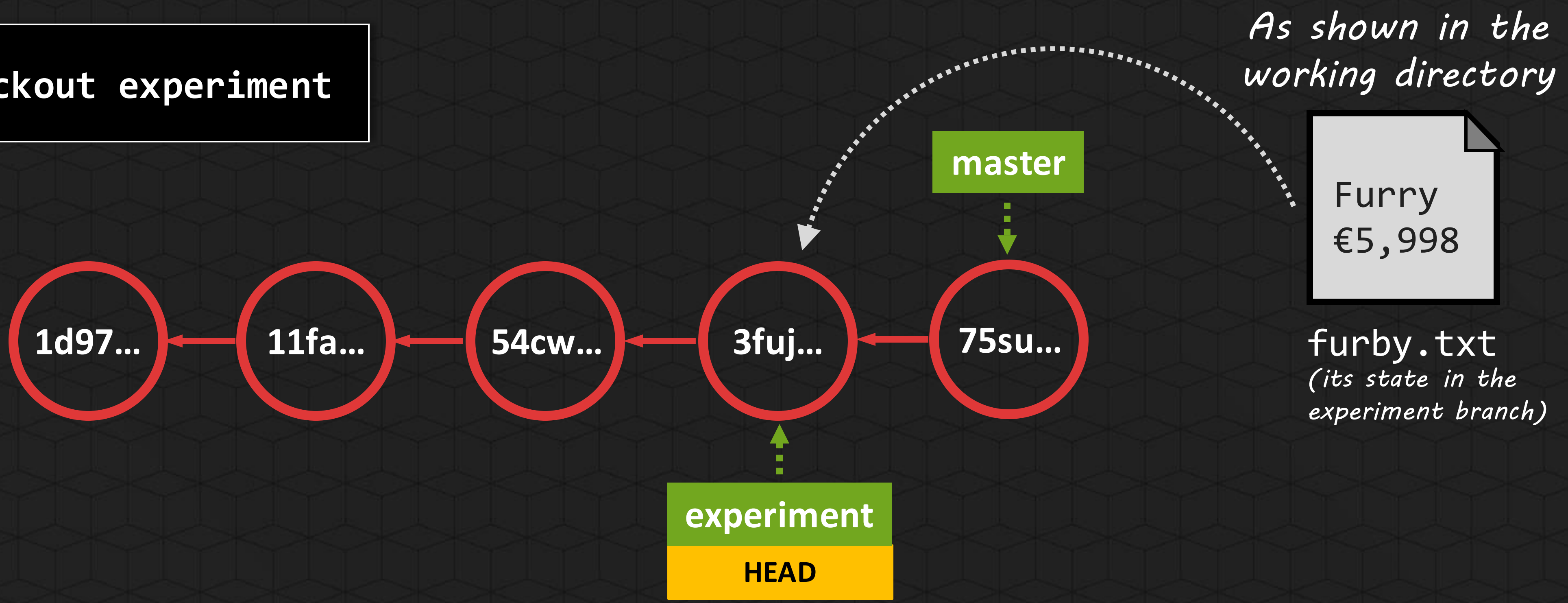
Switching to a branch is done using the checkout command

```
git checkout experiment
```


As shown in the working directory
HEAD
master
Furry
€59,98
furby.txt
(its state in the master branch)
1d97…
11fa…
54cw…
3fuj…
75su…
experiment

Switching to a **branch** is done using the **checkout command**

```
git checkout experiment
```

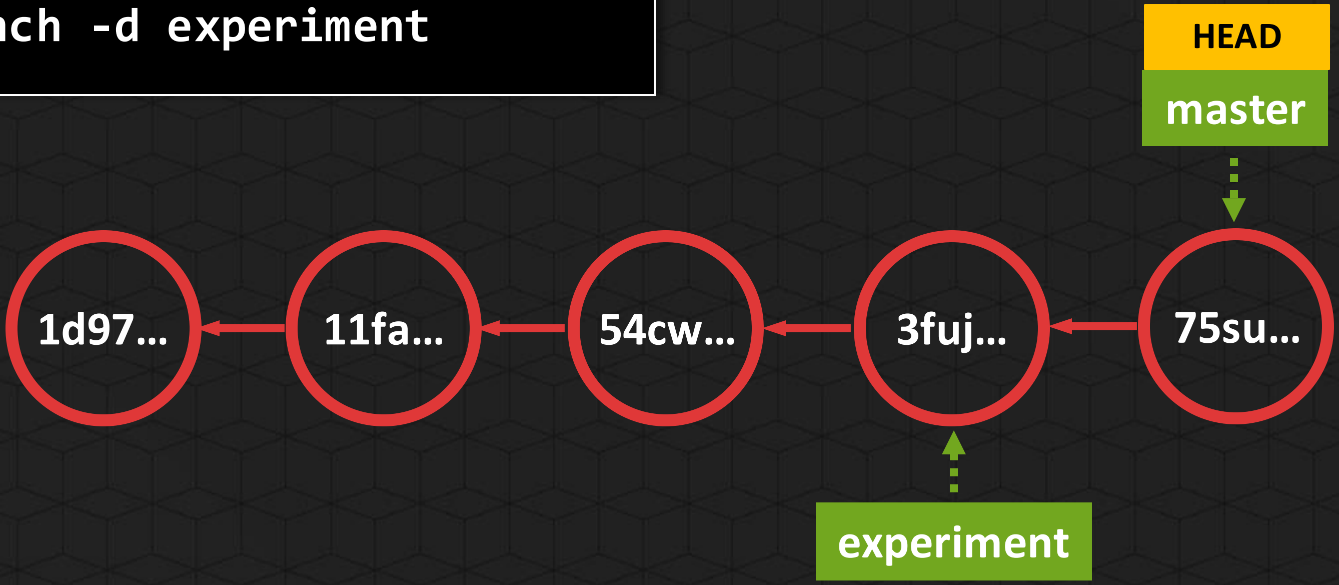


Branches

Deleting a branch

Deleting a **branch** is done whenever a branch has lost its use-case. **Proper branch management** is recommended.

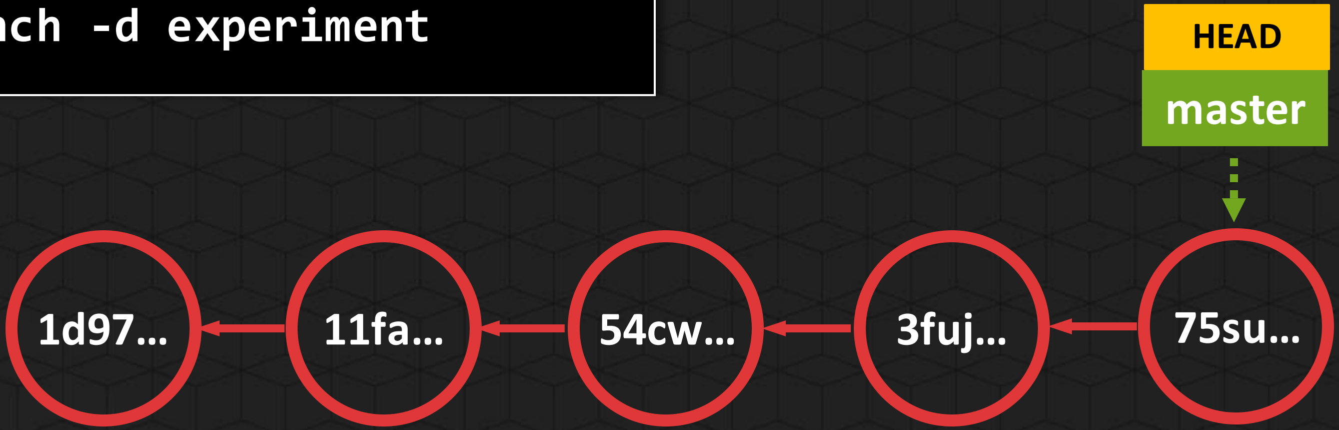
```
git branch -d experiment
```



Deleting a branch

Deleting a **branch** is done whenever a branch has lost its use-case. **Proper branch management** is recommended.

```
git branch -d experiment
```



As a branch is a pointer to a commit, deleting a branch is the process of deleting the pointer. If the branch (has a diverged history and) is not fully merged, deleting the branch is not possible with the above command (as you would lose work). Option `-D` *forces a deleted.*

Deleting a branch

Merging

Chapter 2

Overview

Merging

Introduction

Let's study **how changes from one branch** can be **incorporated into another branch.**

It comes down to the question: How can we merge one commit with another commit?

Git **merges changes in two different ways**

1. **Fast-Forward** merging *(non-diverging histories)*
2. **Three-way** merging *(diverging histories)*
   - ✓ **Auto-merge** (for non conflicting changes)
   - ✓ **Manual merge** (for conflicting changes, merge conflicts)

Branches

Fast-Forward merge

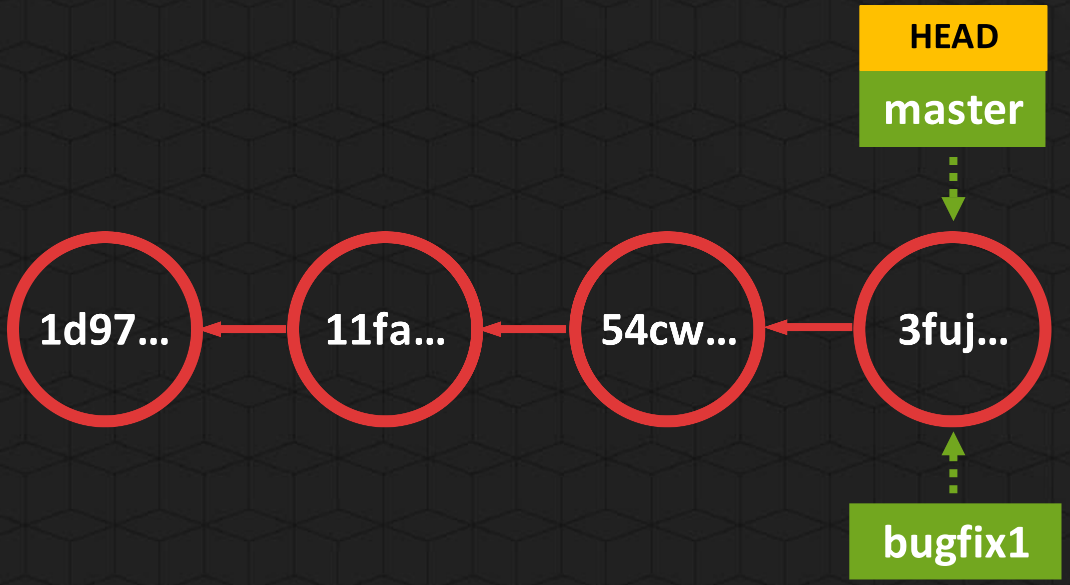
Git **automatically simplifies merging** one commit with another commit when there is **no divergent history to merge** together. This simplified process is called **Fast-Forward merge** (or mode).

Fast-Forward merge

Merging branches: Fast-Forward merge

```
git branch bugfix1
```

```
git checkout bugfix1
```

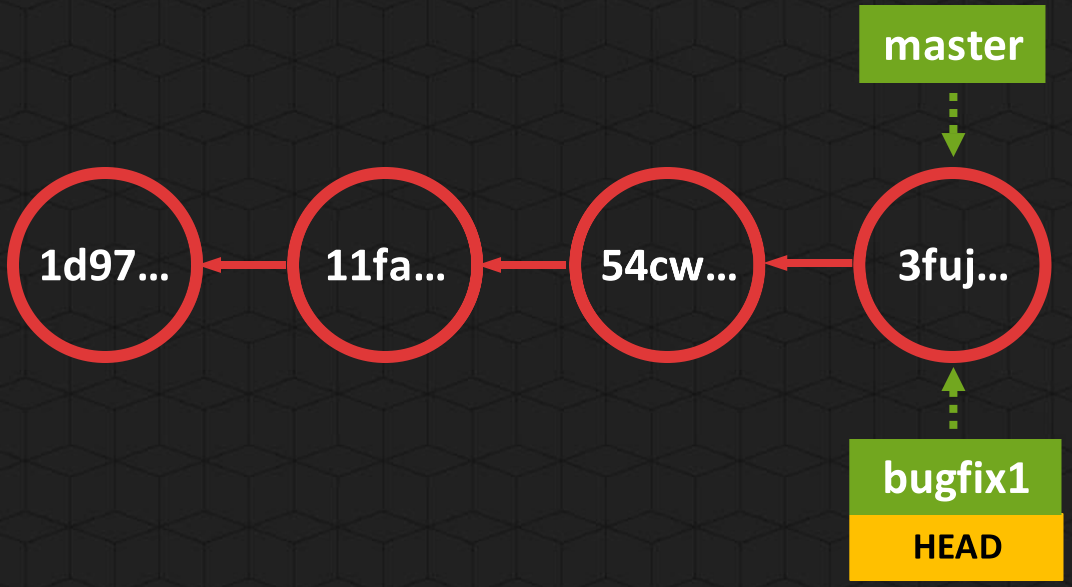


Merging branches: Fast-Forward merge

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug
#1 fixed”
```

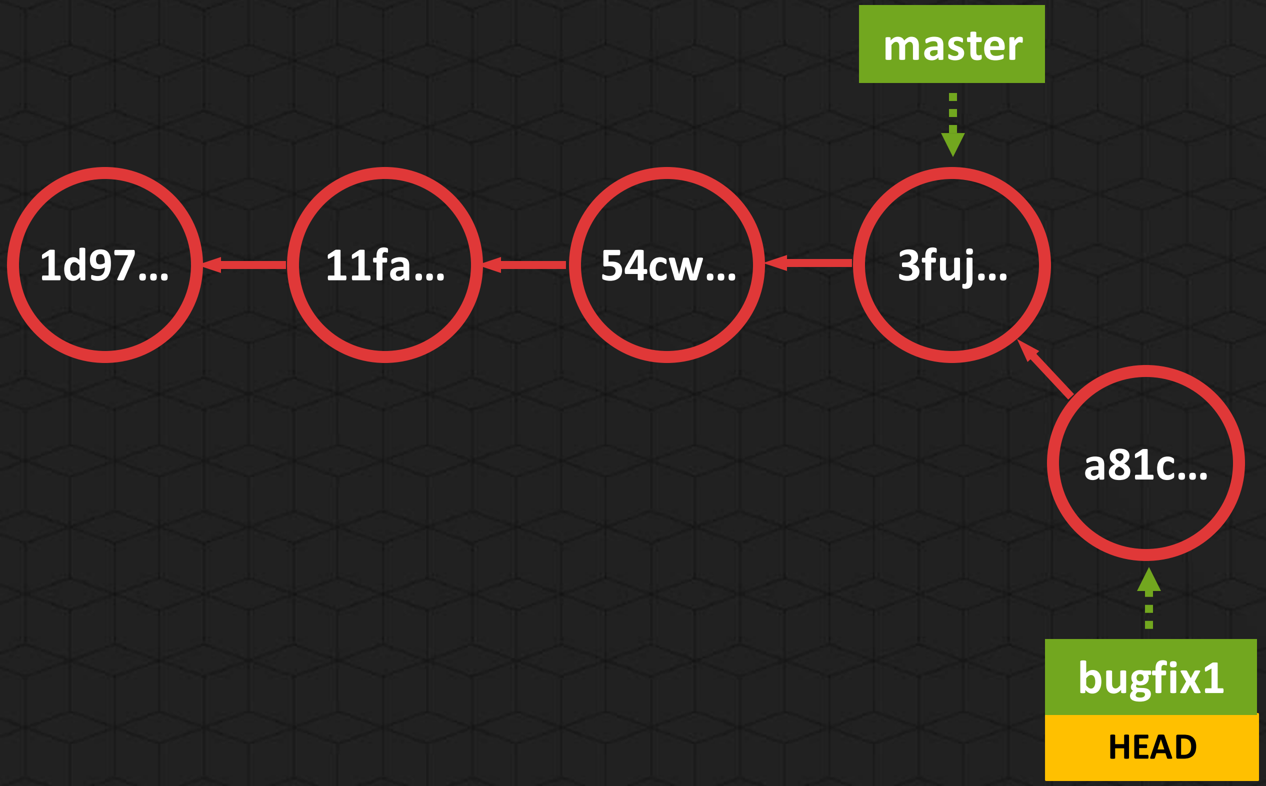


Merging branches: Fast-Forward merge

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```



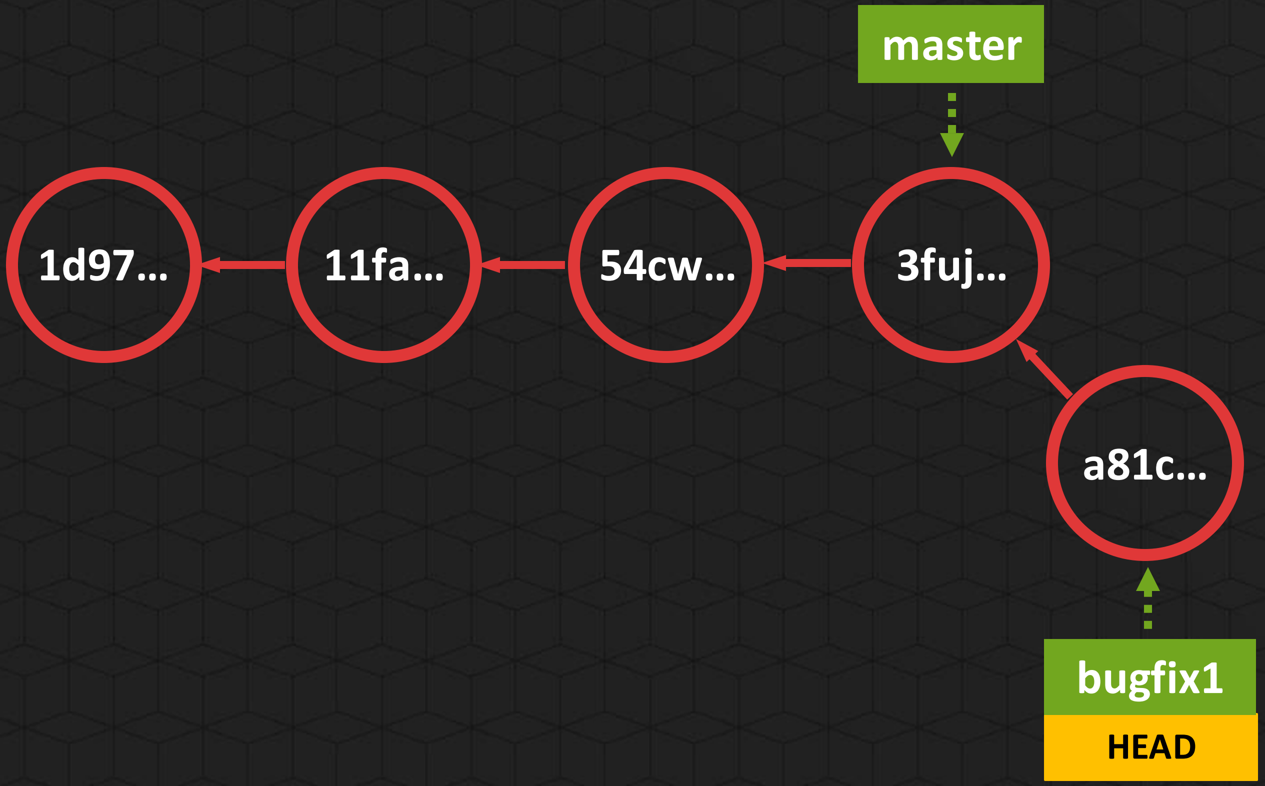
Merging branches: Fast-Forward merge

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git commit –m “bug #1 refactored”
```



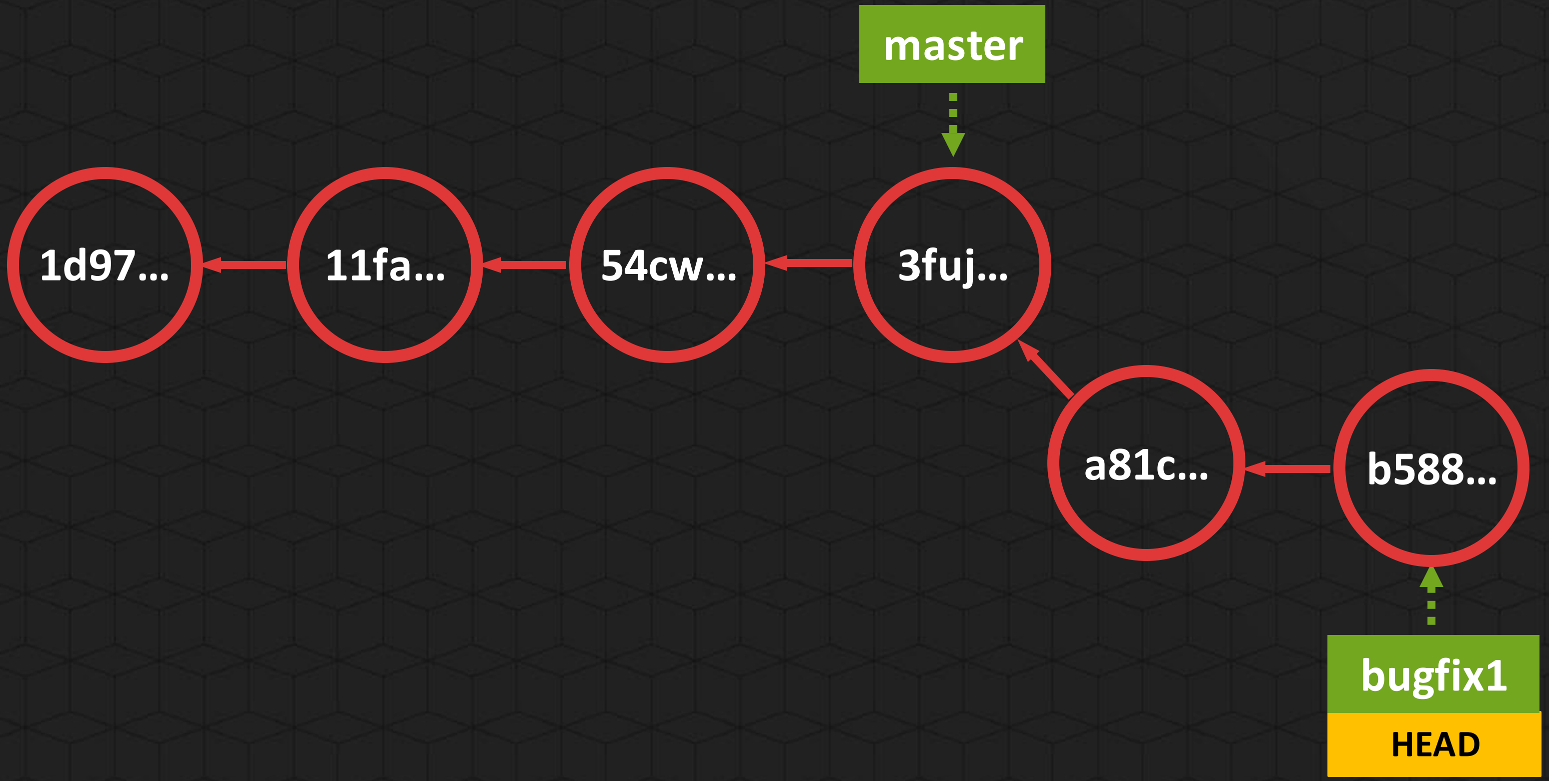
Merging branches: Fast-Forward merge

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git commit –m “bug #1 refactored”
```



Merging branches: Fast-Forward merge

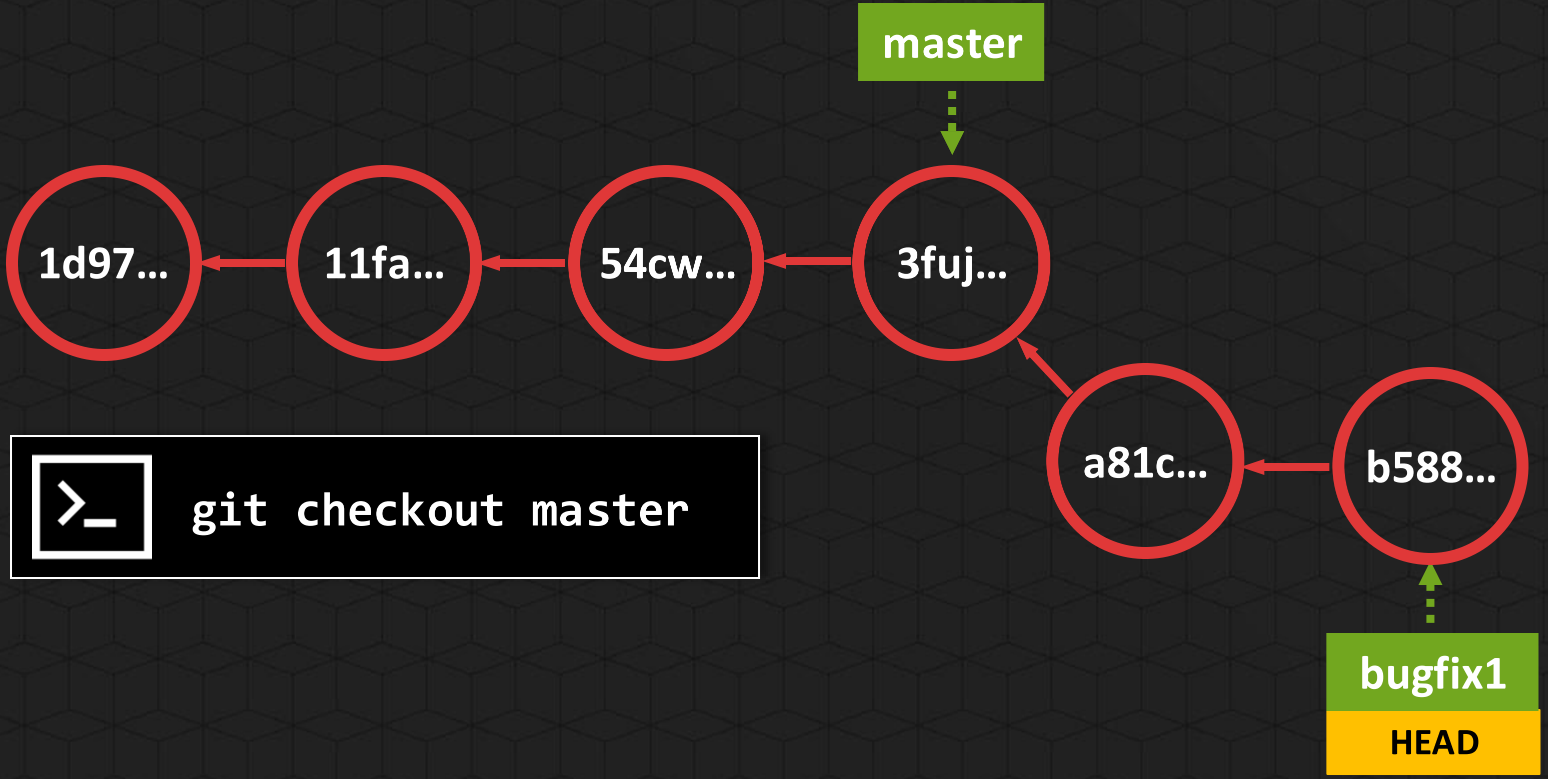
```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git commit –m “bug #1 refactored”
```

```
git checkout master
```



Merging branches: Fast-Forward merge

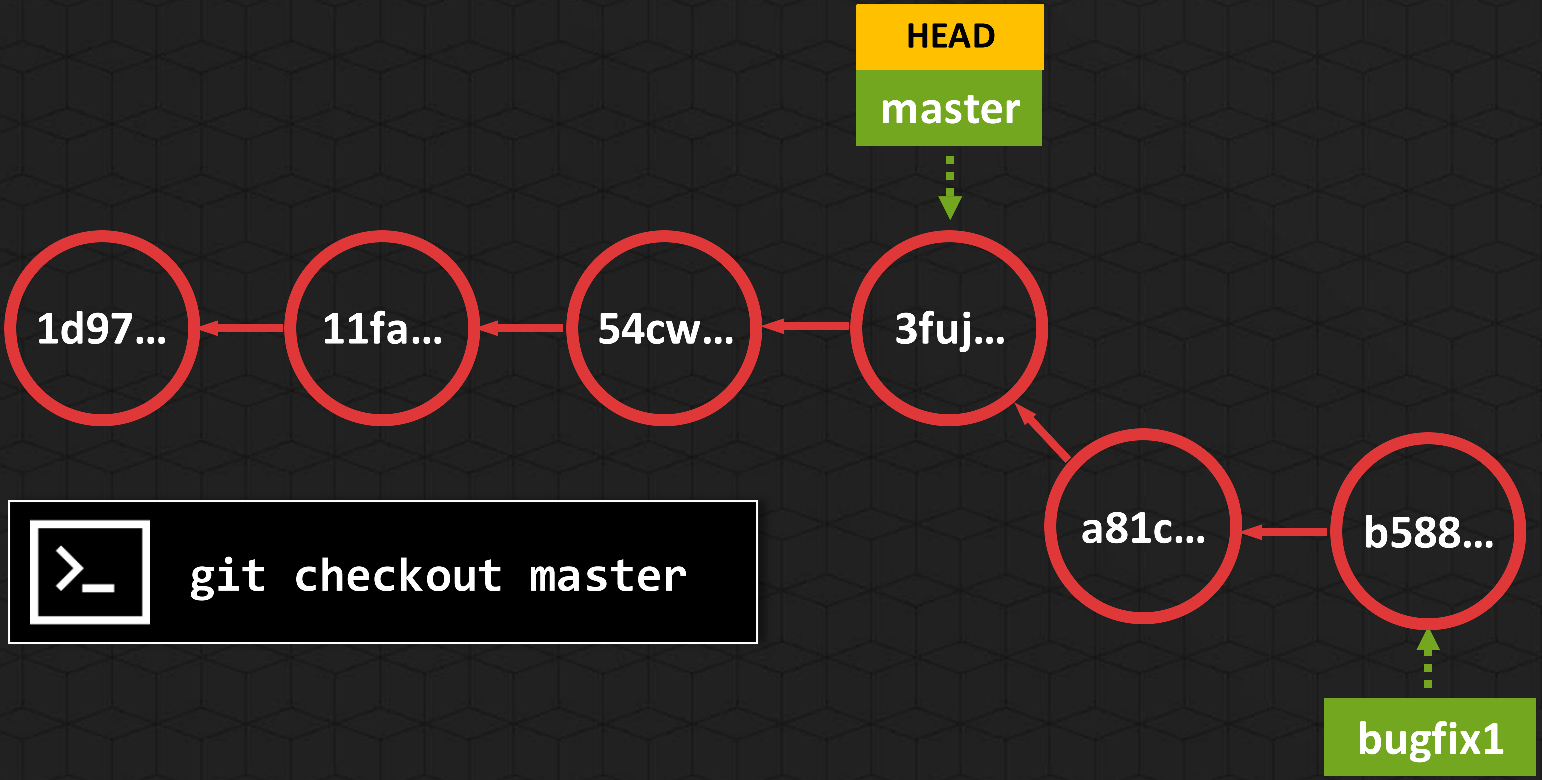
```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git commit –m “bug #1 refactored”
```

```
git checkout master
```



Merging branches: Fast-Forward merge

```
git branch bugfix1
```

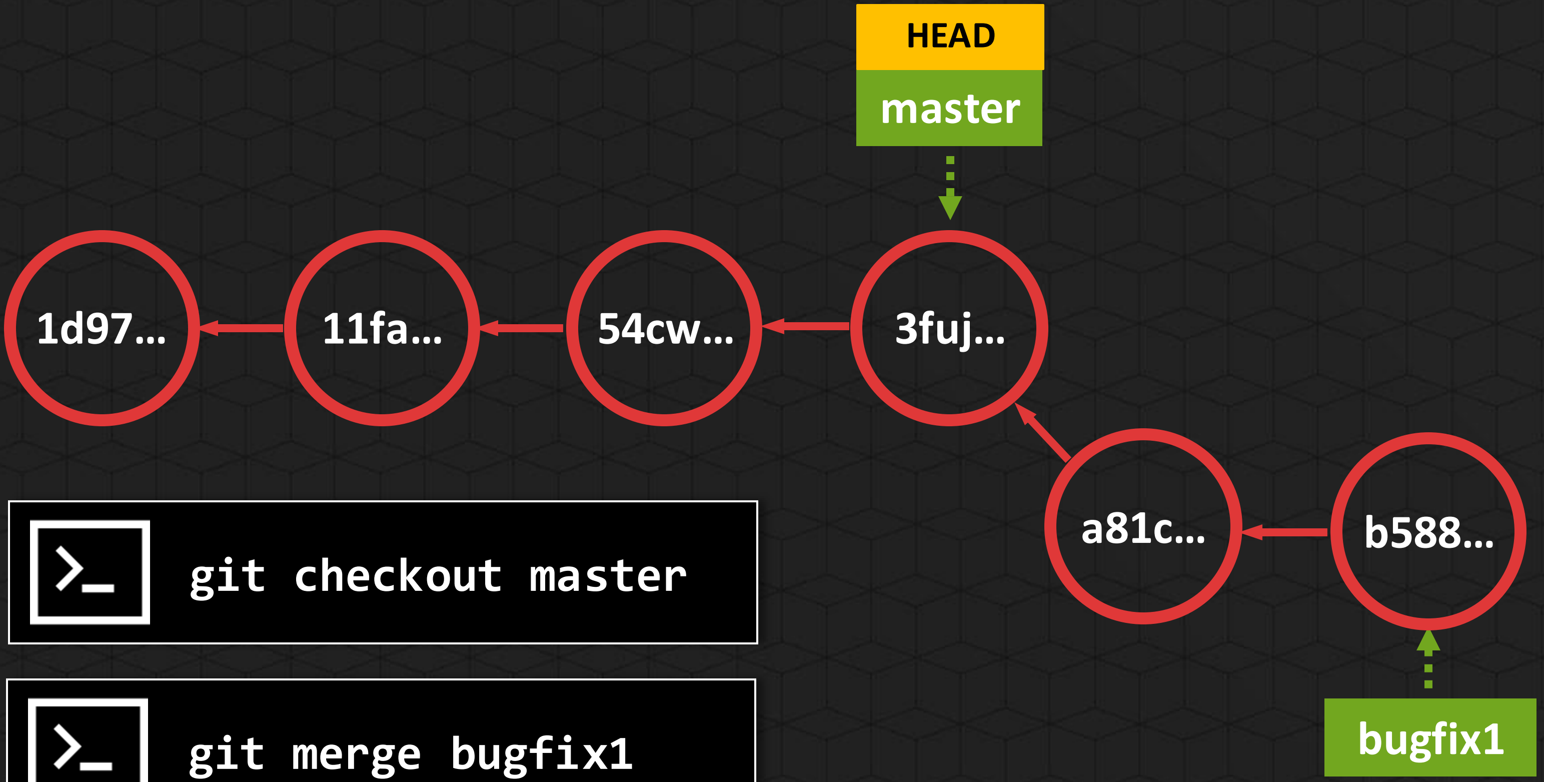
```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git commit –m “bug #1 refactored”
```

```
git checkout master
```

```
git merge bugfix1
```



Merging branches: Fast-Forward merge

```
git branch bugfix1
```

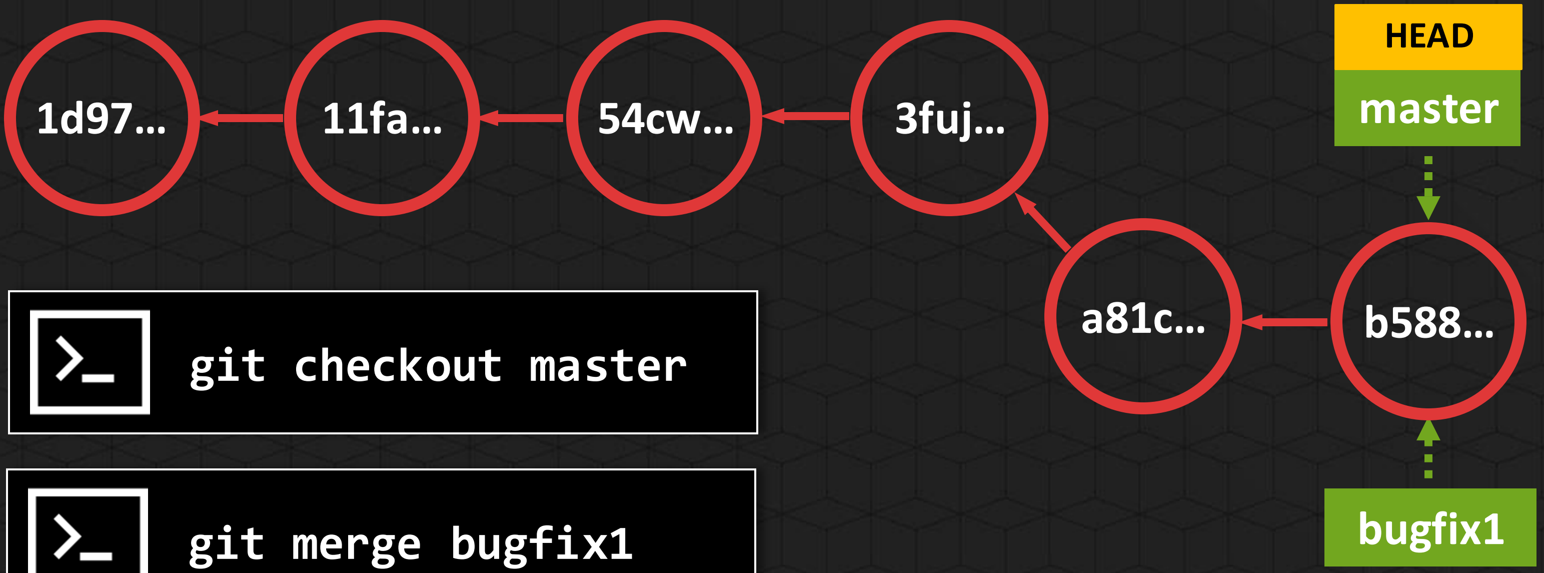
```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

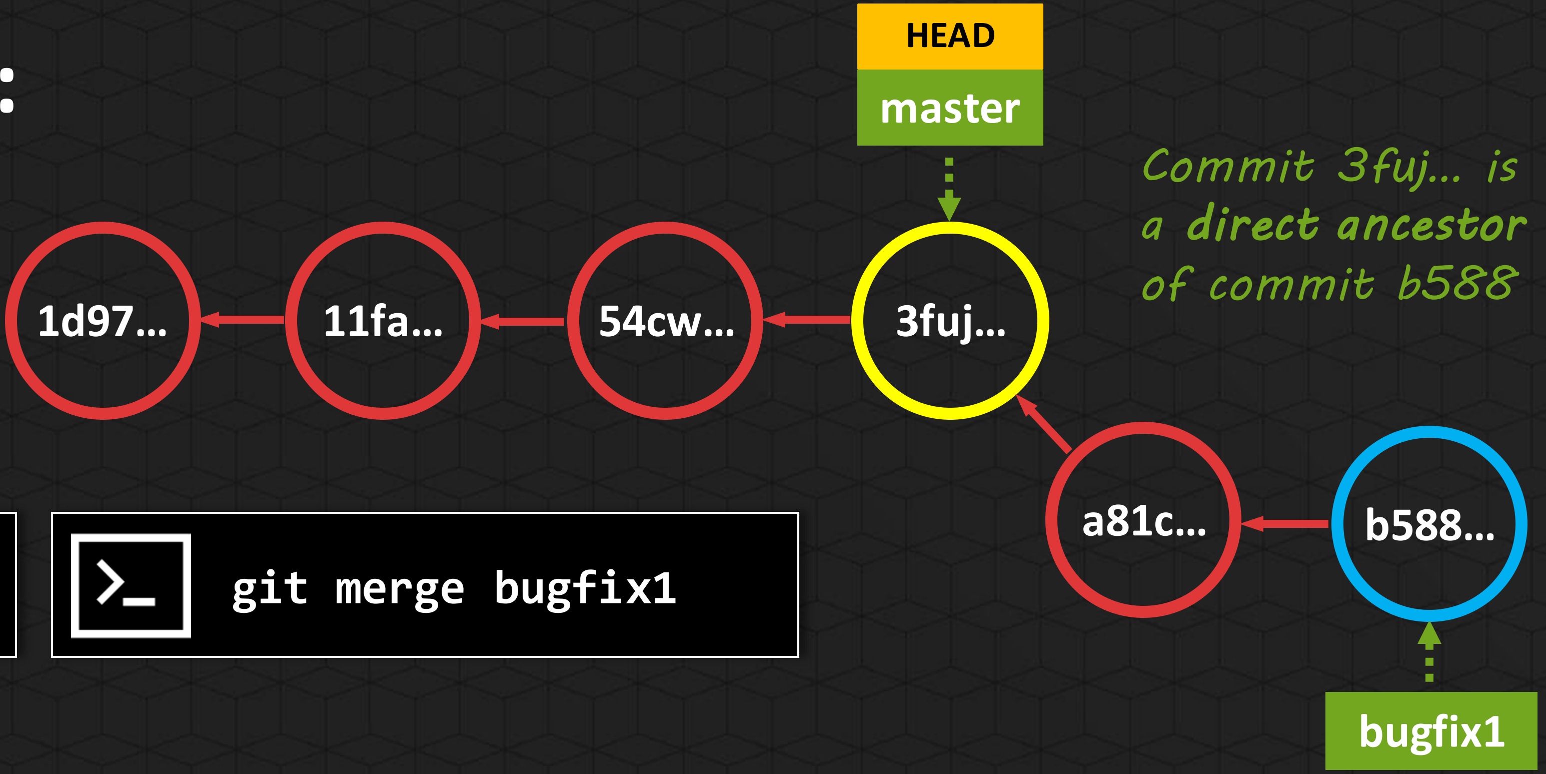
```
git commit –m “bug #1 refactored”
```

```
git checkout master
```

```
git merge bugfix1
```



Merging branches: Fast-Forward merge

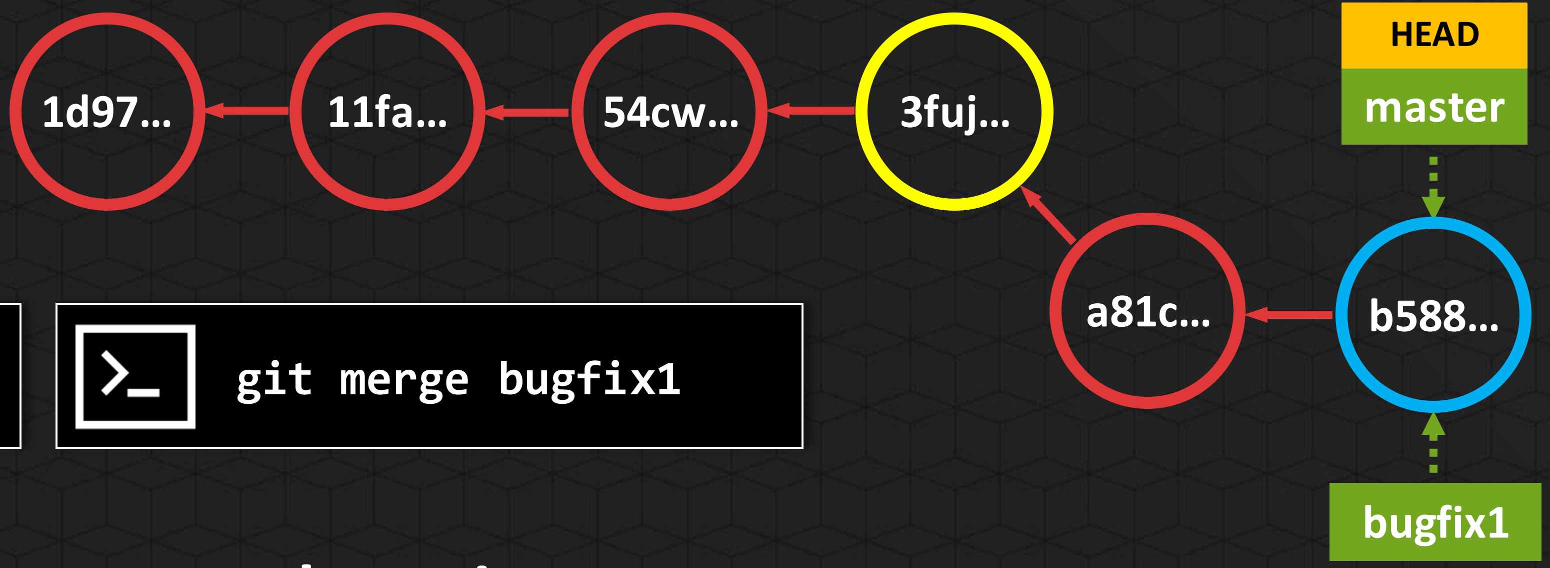


```
git checkout master
```

```
git merge bugfix1
```

When **merging a commit** (b588) with a **commit** (3fuj) that can be **reached** by **following** the **first commit's** (b588) **history**, git will **automatically** apply **Fast-Forward** merging.

Merging branches: Fast-Forward merge



```
git checkout master
```

```
git merge bugfix1
```

In that scenario, git can simply **move the pointer** of **your current branch forward**, to the commit you wanted to merge with.

Fast-Forward merging is possible as long as the **git histories have not diverged**.

Let’s **demonstrate** this *(hands-on in group)*

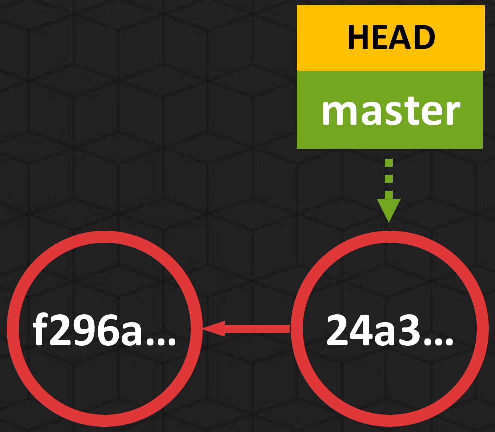
- ❑ Let’s create the following directory structure using CMD

```
|-- diary
    |-- mydiary.txt
```

- ❑ Let’s put some text in `mydiary.txt` (e.g. “Dear diary, git is lit.”)
- ❑ Let’s initialize Git inside our **`diary`** working directory
  - ❑ `> git init`
- ❑ Let’s start tracking the file and commit the changes.
  - ❑ `> git add .`
  - ❑ `> git commit –m “mydiary.txt added”`
- ❑ Let’s put an extra line of text in `mydiary.txt` (e.g. “I forgot to feed the cat”)
- ❑ Commit the changes (message: “cat entry added to mydiary”).

Let’s **demonstrate** this *(hands-on in group)*

```
> git log --all --decorate --oneline --graph
* 24a3ab1 (HEAD -> master) cat entry added to mydiary
* f2d96a8 mydiary.txt added
```



Let's **demonstrate** this *(hands-on in group)*

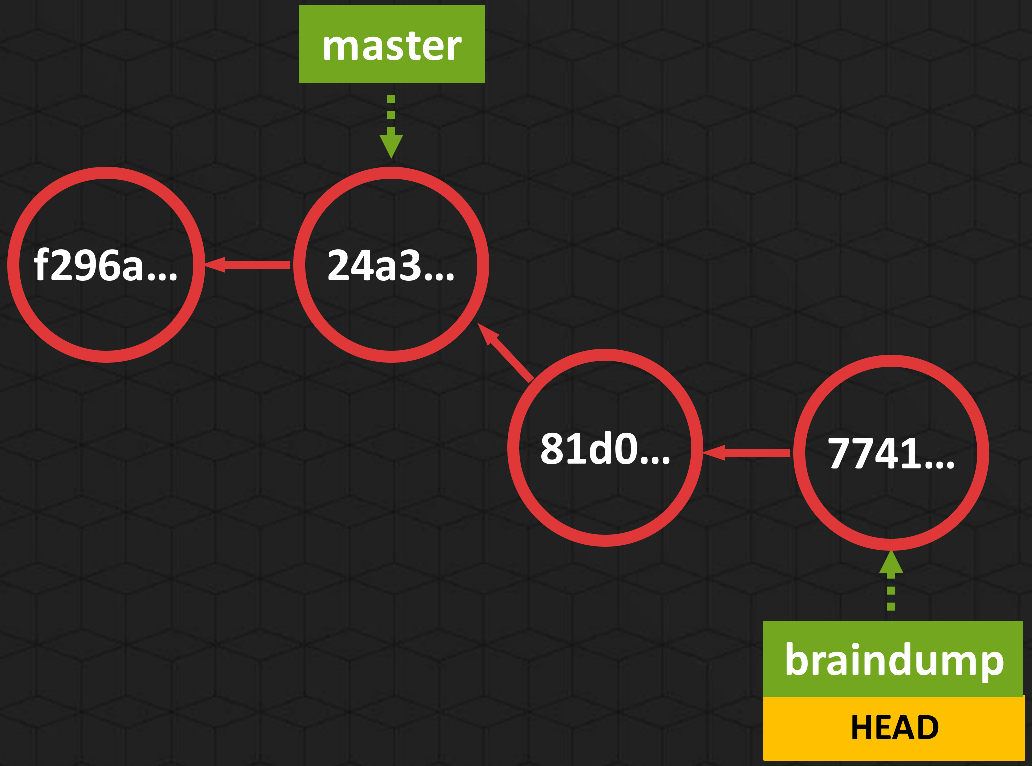
- ❑ Let's create a new branch
  - ❑ `> git branch braindump`
- ❑ Then, switch to it
  - ❑ `> git checkout braindump`
- ❑ Let's put an extra line of text in `mydiary.txt` (e.g. "Winter is coming")
- ❑ Commit the changes (message: "winter entry added").
- ❑ Let's put another extra line of text in `mydiary.txt` (e.g. "Slipped on ice")
- ❑ Commit the changes (message: "slippery entry added").

Let’s **demonstrate** this *(hands-on in group)*

```
> git log --all --decorate --oneline --graph
* 7741d10 (HEAD -> braindump) slippery entry added
* 81d04f4 winter entry added
* 24a3ab1 (master) cat entry added to mydiary
* f2d96a8 mydiary.txt added
```

- ❑ Let’s inspect the **`mydiary.txt`** file on the **current branch**
  - ❑ > `start mydiary.txt`

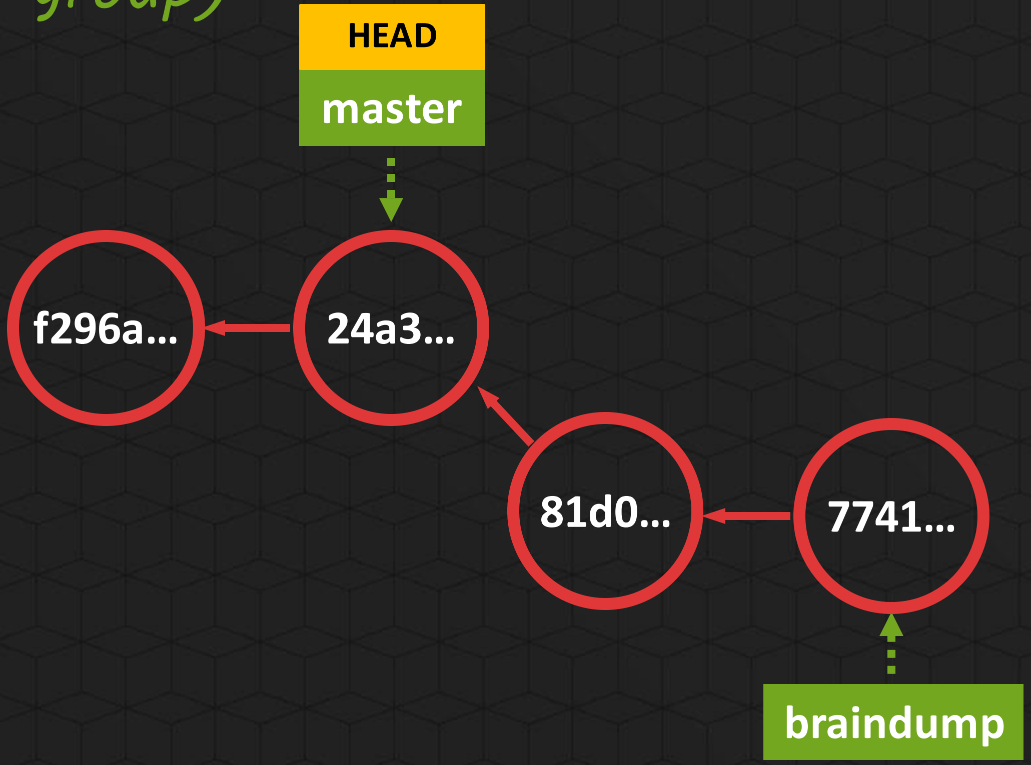
```
Dear diary, git is lit.
I forgot to feed the cat
Winter is coming
Slipped on ice
```



Let's **demonstrate** this *(hands-on in group)*

- ❑ Now, let's inspect the **`mydiary.txt`** file on the **master branch**
  - ❑ > `git checkout master`
  - ❑ > `start mydiary.txt`

```
Dear diary, git is lit.
I forgot to feed the cat
```



Merging

Fast-Forward merge

Let’s **demonstrate** this *(hands-on in group)*

- ❑ Now, let merge the **`braindump`** branch back into the **`master`** branch. First, let’s make sure we’re on the **`master`** branch.
  - ❑ > `git branch -v`
- ❑ Then, when on the **`master`** branch, merge the **`braindump`** branch.
  - ❑ > `git merge braindump`
- ❑ The following output shows a **successful Fast-Forward merge**.

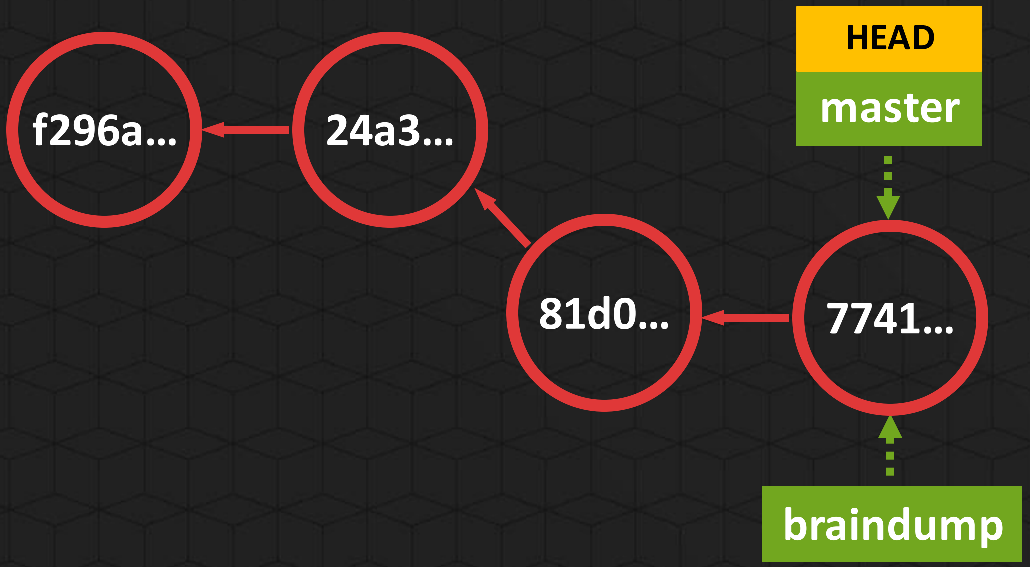
```
> git merge braindump
Updating 24a3ab1..7741d10
Fast-forward
 mydiary.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Let's **demonstrate** this *(hands-on in group)*

```
> git log --all --decorate --oneline --graph
* 7741d10 (HEAD -> master, braindump) slippery entry added
* 81d04f4 winter entry added
* 24a3ab1 cat entry added to mydiary
* f2d96a8 mydiary.txt added
```

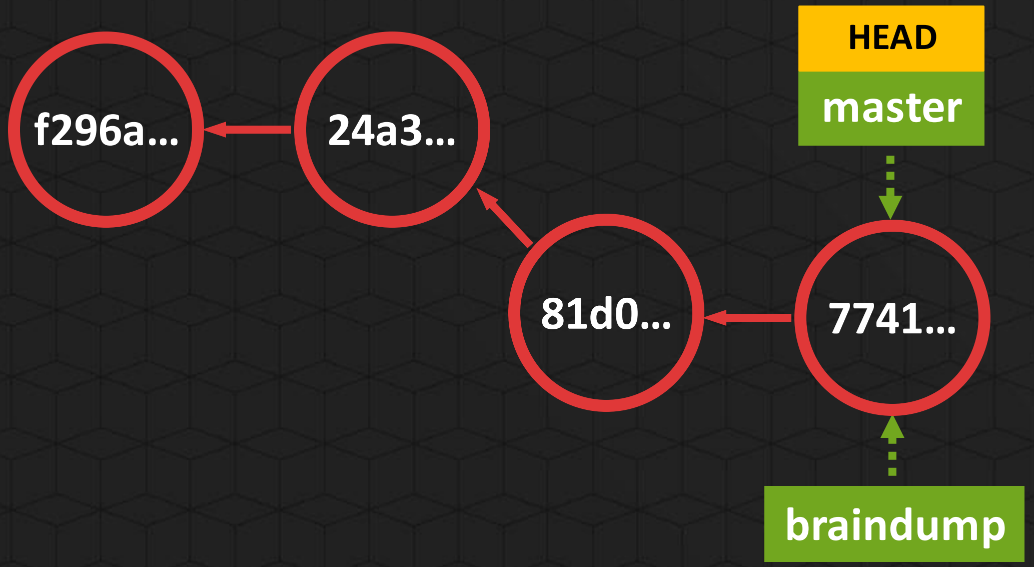
- ❑ Again, let's inspect the **`mydiary.txt`** file on the **master branch**
  - ❑ `> start mydiary.txt`

```
Dear diary, git is lit.
I forgot to feed the cat
Winter is coming
Slipped on ice
```



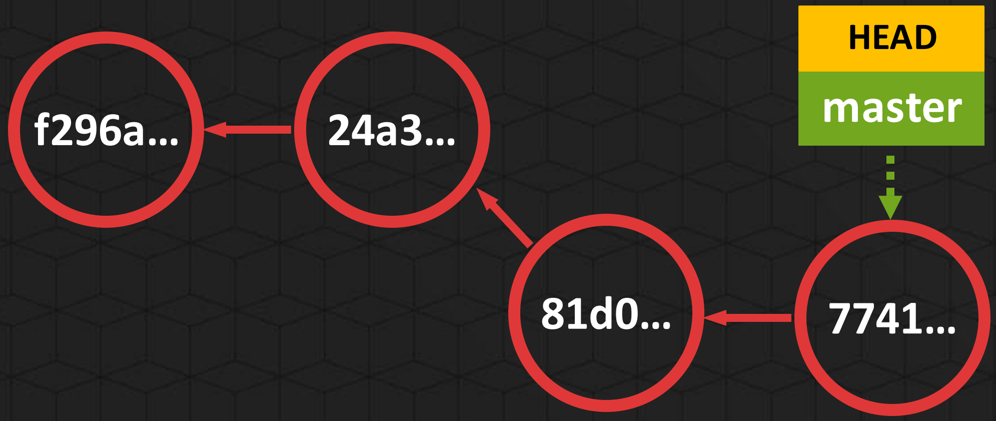
Let’s **demonstrate** this *(hands-on in group)*

- ❑ Our **`braindump`** branch has served its purpose, let’s now delete it.
  - ❑ `> git branch -d braindump`



Let's **demonstrate** this *(hands-on in group)*

- ❑ Our **`braindump`** branch has served its purpose, let's now delete it.
  - ❑ > `git branch -d braindump`
- ❑ Let's validate the **`braindump`** branch is removed.
  - ❑ > `git branch -v`



Branches

Three-way merge

Git **merges changes in two different ways**

1. **Fast-Forward** merging *(non-diverging histories)*
2. **Three-way** merging *(diverging histories)*
   - ✓ **Auto-merge** (for non conflicting changes)
   - ✓ **Manual merge** (for conflicting changes, merge conflicts)

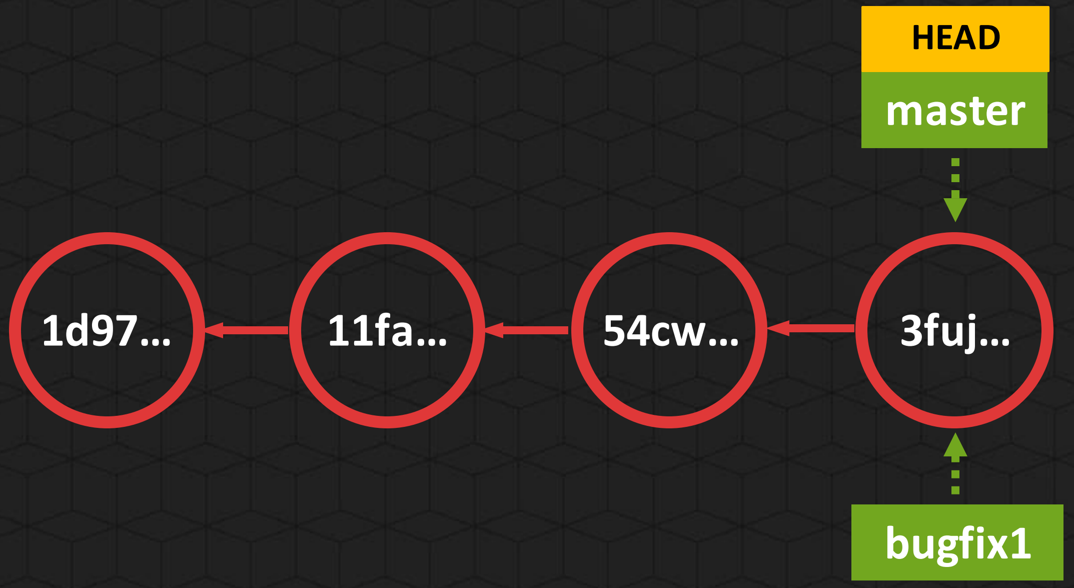
Let’s now study **merging when the histories have diverged**, when **no Fast-Forward** can be applied.

Three-way merge

Merging branches: diverged history

```
git branch bugfix1
```

```
git checkout bugfix1
```

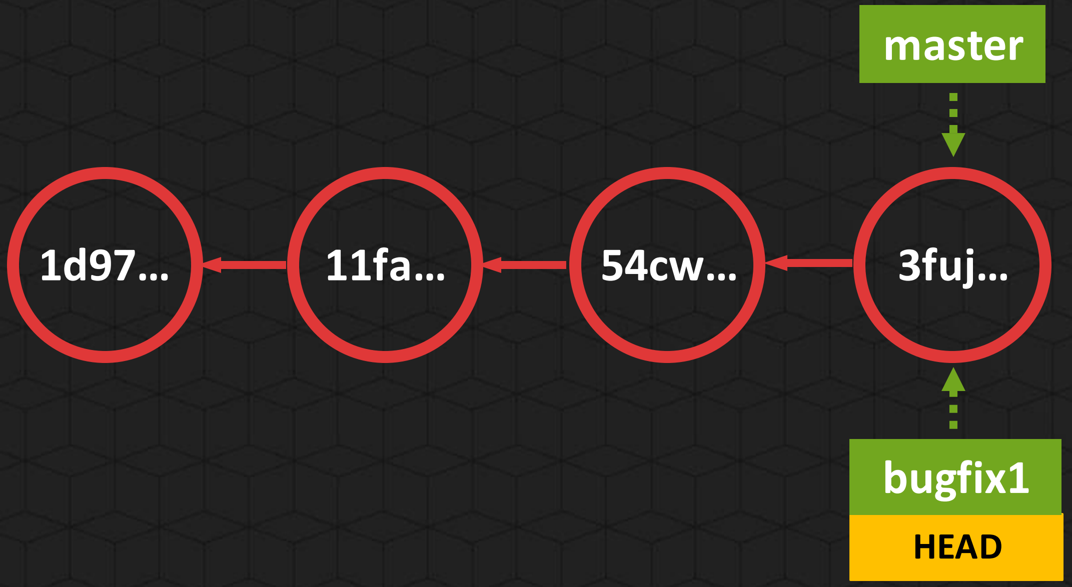


Merging branches: diverged history

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

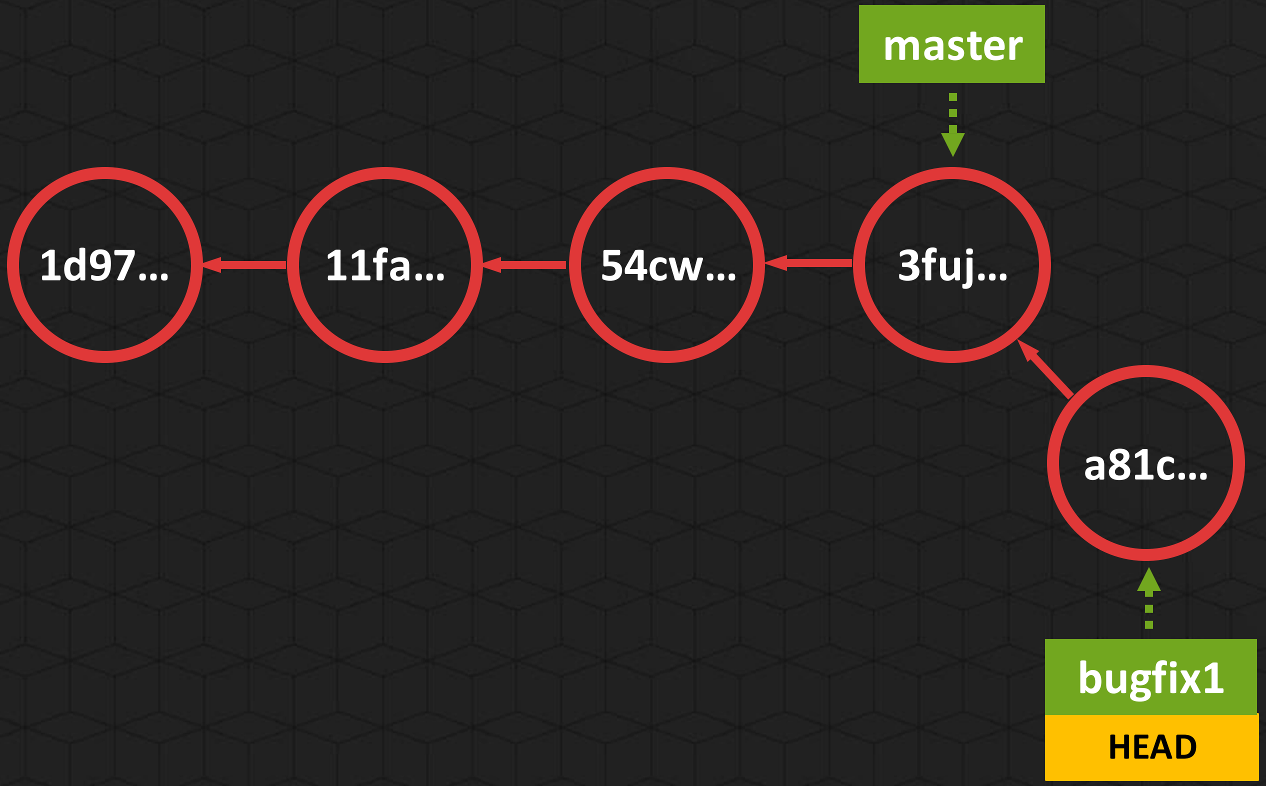


Merging branches: diverged history

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```



Three-way merge

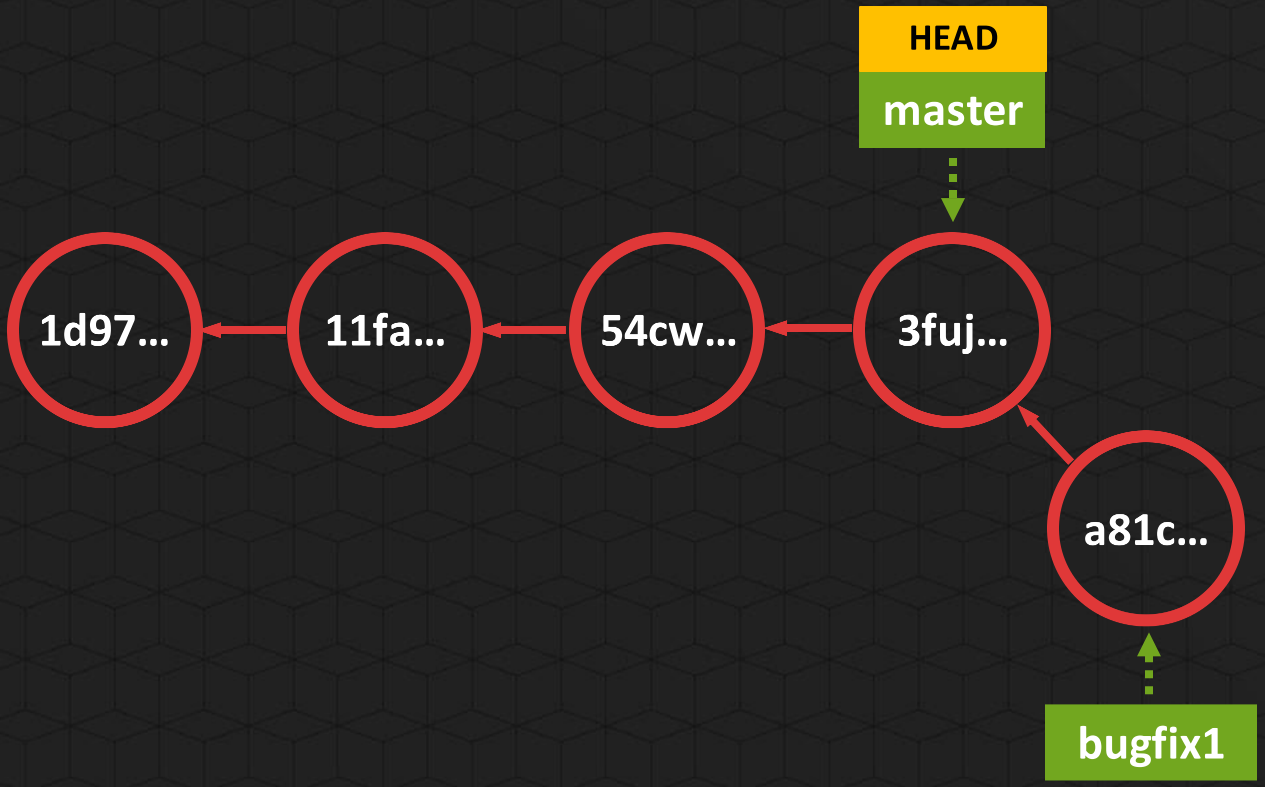
Merging branches: diverged history

```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git checkout master
```



Merging

Three-way merge

Merging branches: diverged history

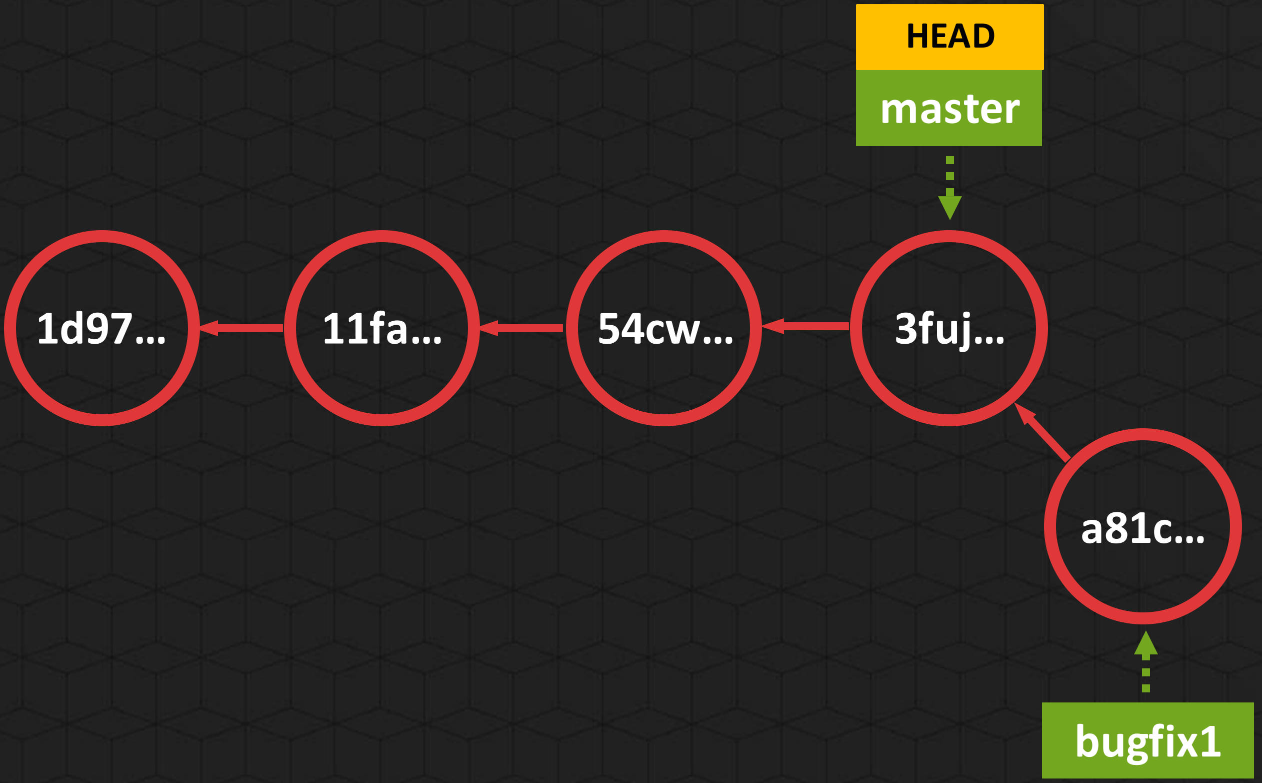
```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git checkout master
```

```
git commit -m “test”
```



Merging branches: diverged history

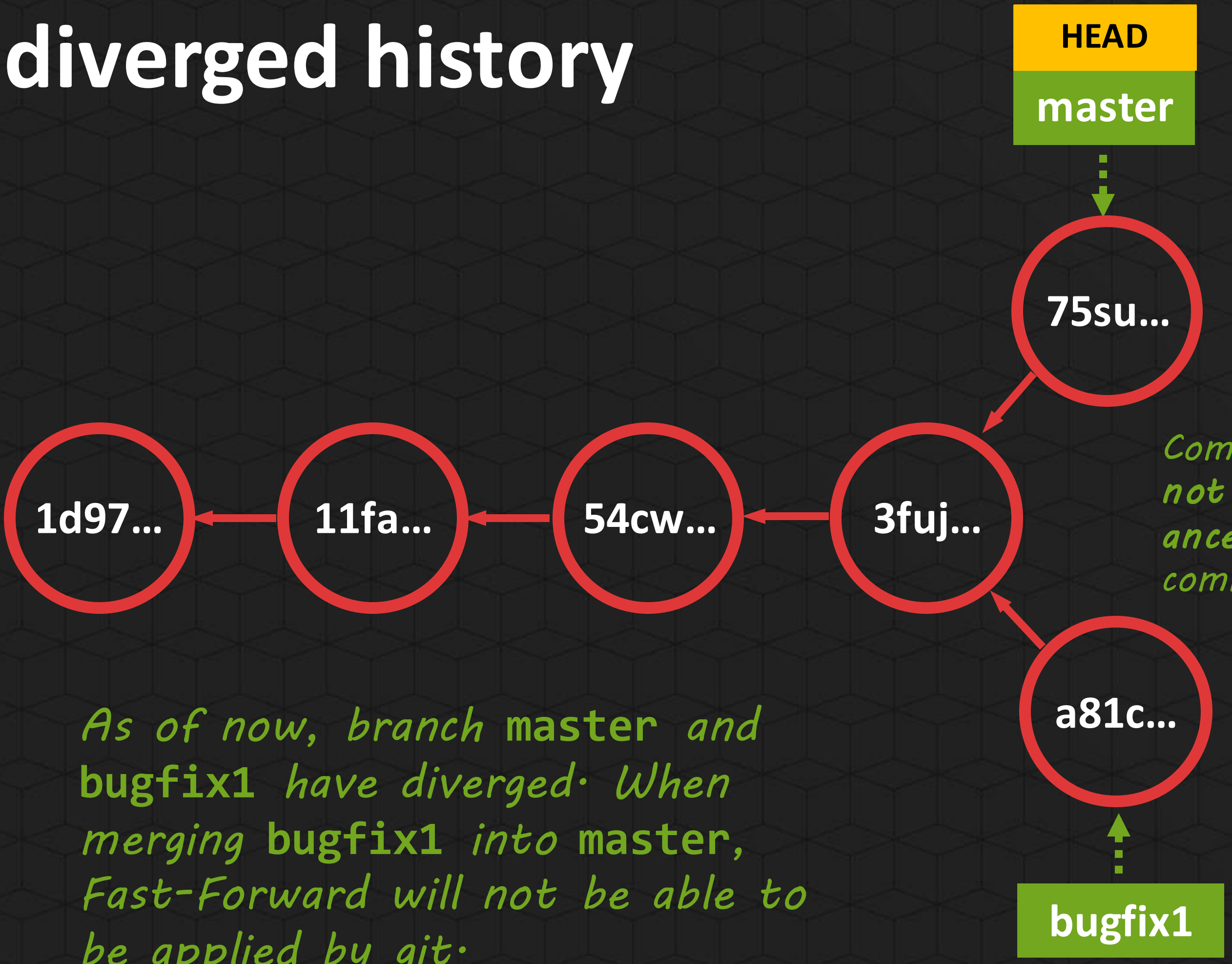
```
git branch bugfix1
```

```
git checkout bugfix1
```

```
git commit -m “bug #1 fixed”
```

```
git checkout master
```

```
git commit -m “test”
```



*Commit 75su is **not a direct ancestor of** commit a81c*

As of now, branch **master** *and* **bugfix1** *have diverged. When merging* **bugfix1** *into* **master**, *Fast-Forward will not be able to be applied by git.*

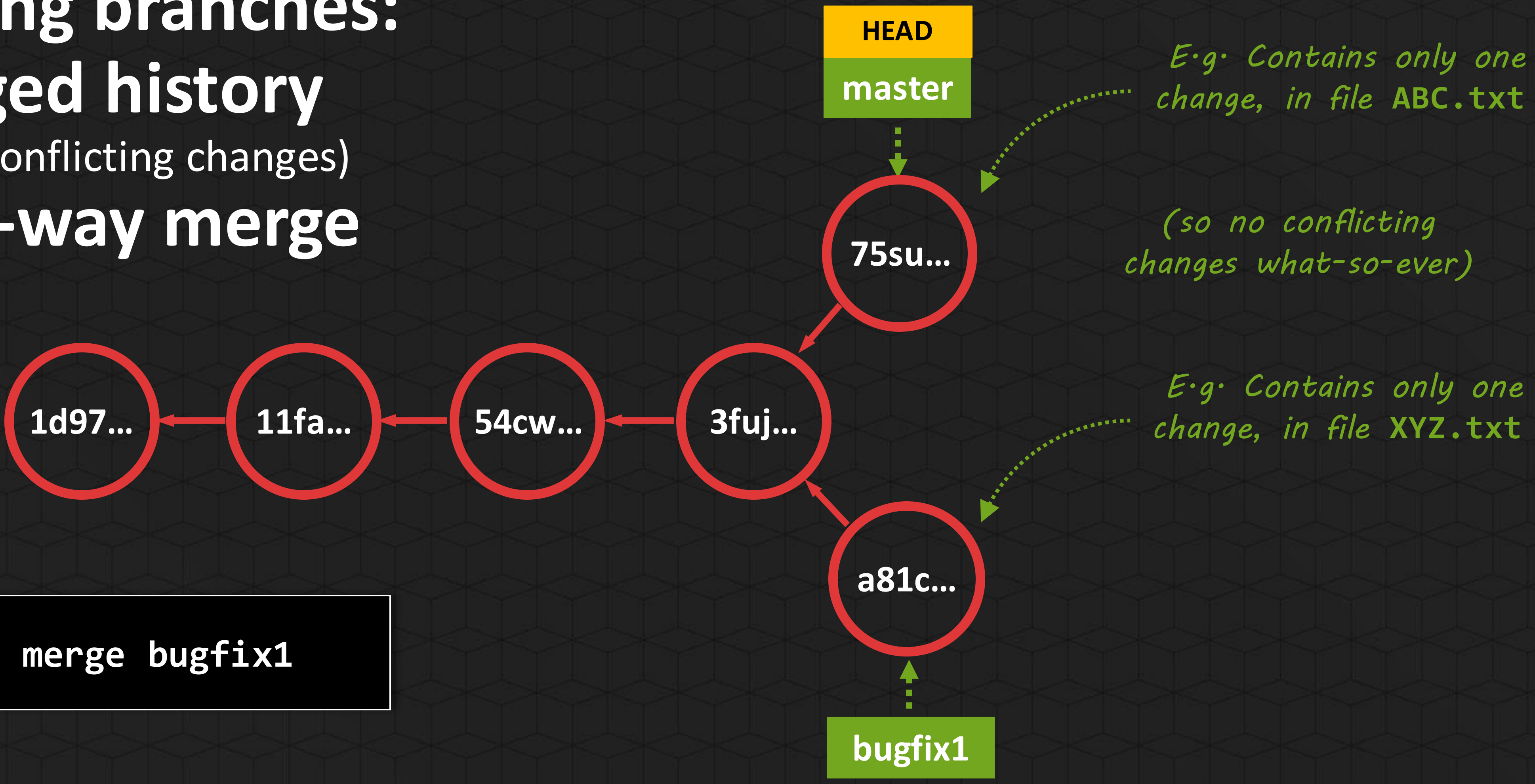
When **merging two commits with a diverged history**, git **automatically** performs a **three-way merge**

*If the changes are made in different files or made in different parts (~lines) of the same file, the changes will **not** conflict.*

Merging branches: diverged history

(with no conflicting changes)

Three-way merge

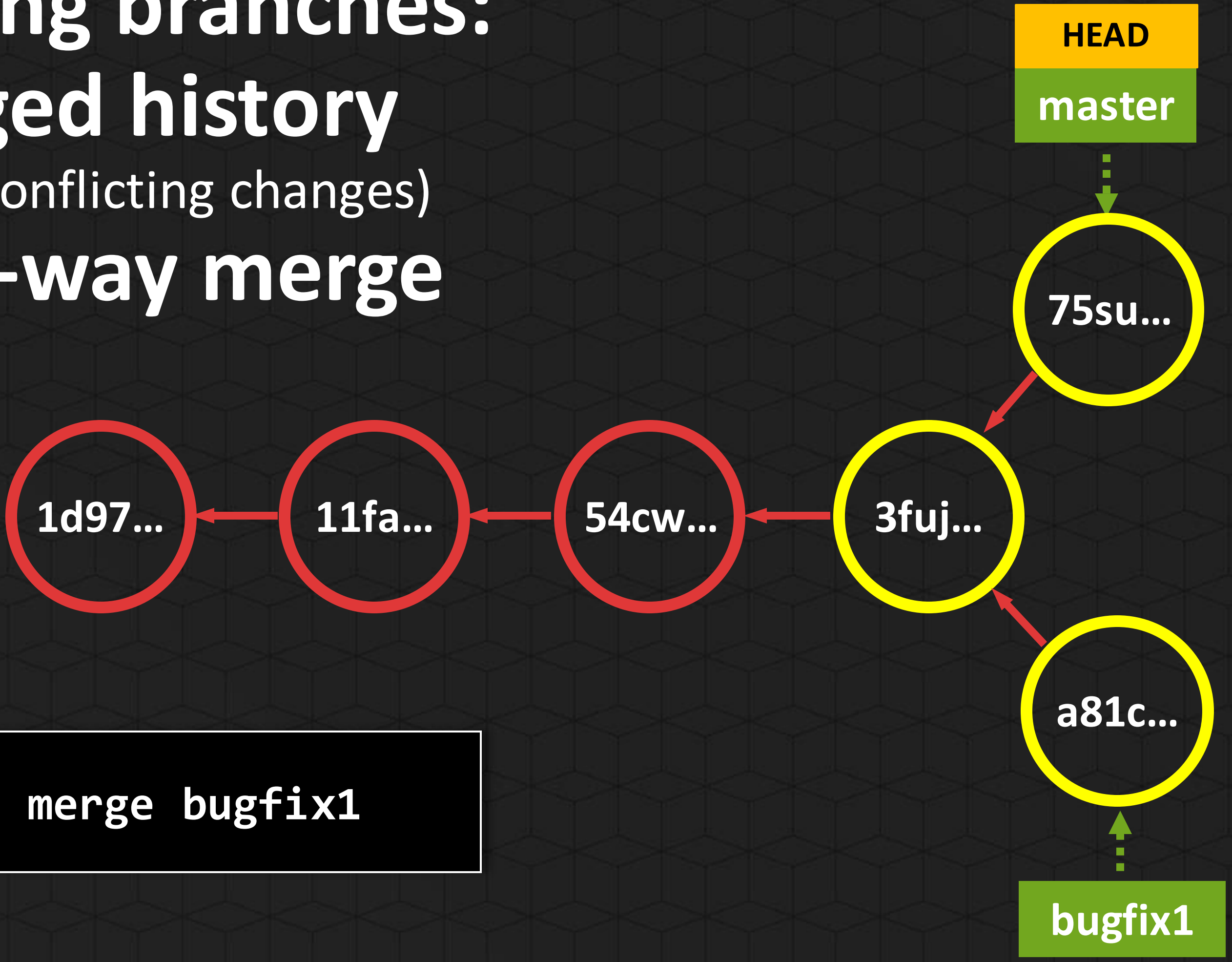

HEAD
master
75su...
E.g. Contains only one change, in file ABC.txt
(so no conflicting changes what-so-ever)
1d97...
11fa...
54cw...
3fuj...
E.g. Contains only one change, in file XYZ.txt
a81c...
bugfix1

```
git merge bugfix1
```

Merging branches: diverged history

(with no conflicting changes)

Three-way merge



```
git merge bugfix1
```

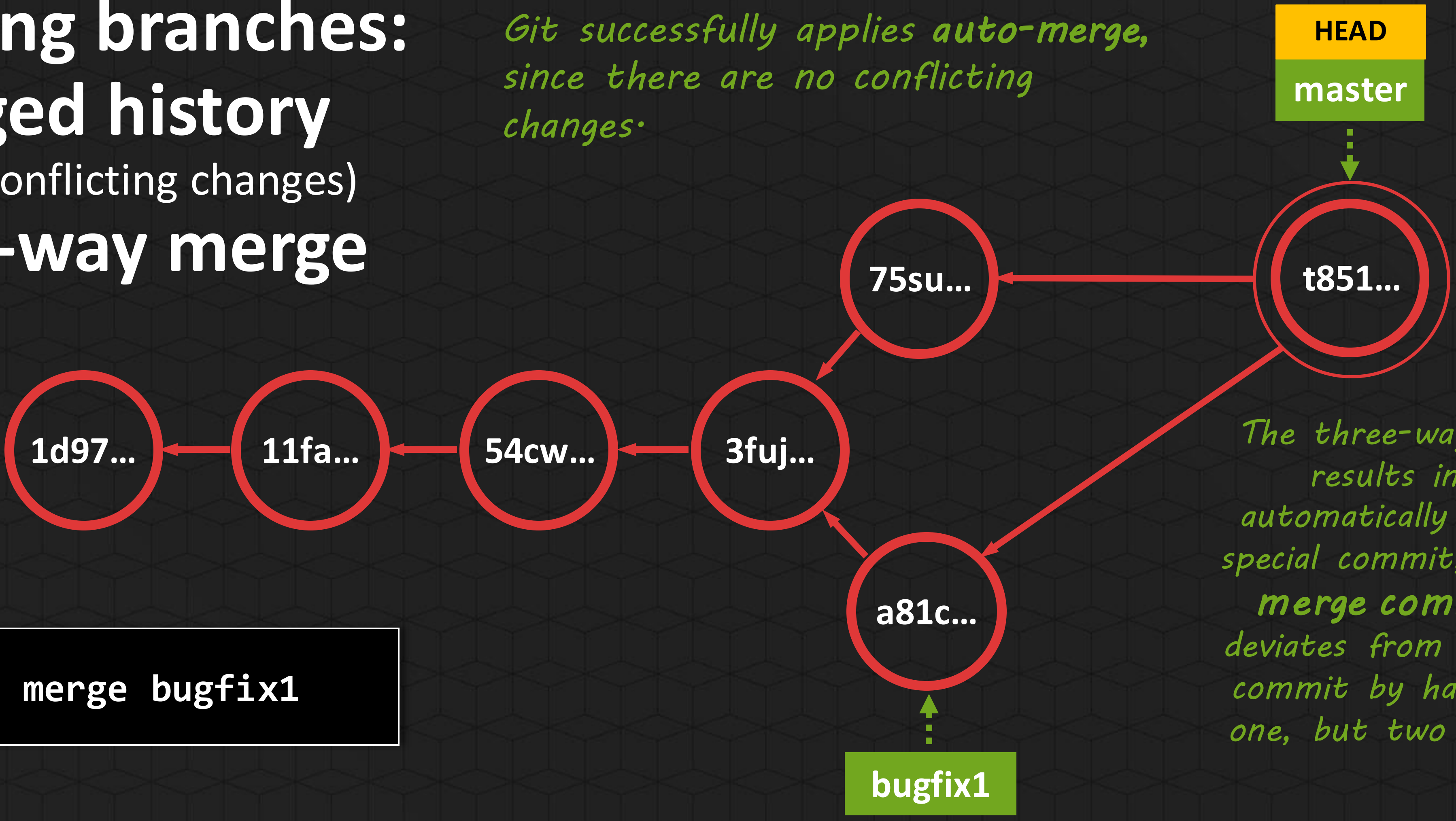
Git uses the commits to which both branches are pointing and their common ancestor to perform a three-way merge (what state do both branches have in common and where do they diverge).

Merging branches: diverged history

(with no conflicting changes)

Three-way merge

*Git successfully applies **auto-merge**, since there are no conflicting changes.*



*The three-way merge results in an automatically created special commit, called a **merge commit**. It deviates from a normal commit by having not one, but two parents.*

```
git merge bugfix1
```

Let’s **walk through** an example *(not hands-on, just walk through it)*

- ❑Let’s create the following directory structure using CMD

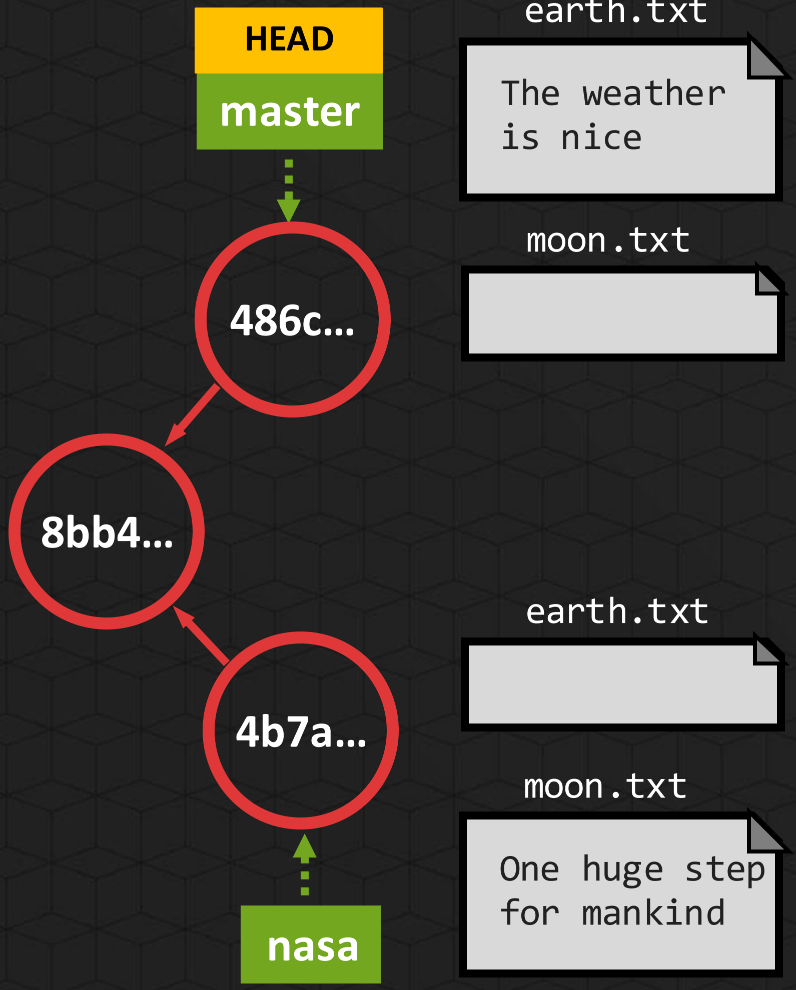
```
|-- planets
    |-- earth.txt
    |-- moon.txt
```

- ❑Let’s initialize Git inside our **`planets`** working directory
  - ❑ > `git init`
- ❑Let’s start tracking both, then commit the changes (message: “earth and moon added”)
- ❑Let’s create a branch **`nasa`** and then switch to that branch (in one go)
  - ❑ > `git checkout -b nasa`
- ❑Let’s put an extra line of text in `moon.txt` (e.g. “One huge step for mankind”)
- ❑Commit the changes (message: “words of Armstrong added”).

Let’s **walk through** an example

- ❑ Let’s switch back to branch **`master`**
  - ❑ > `git checkout master`
- ❑ Let’s put an extra line of text in `earth.txt` (e.g. “The weather is nice”)
- ❑ Commit the changes (message: “weather update”).

```
> git log --all --decorate --oneline --graph
* 486c105 (HEAD -> master) weather update
| * 4b7ad68 (nasa) words of armstrong added
|/
* 8bb4a59 earth and moon added
```

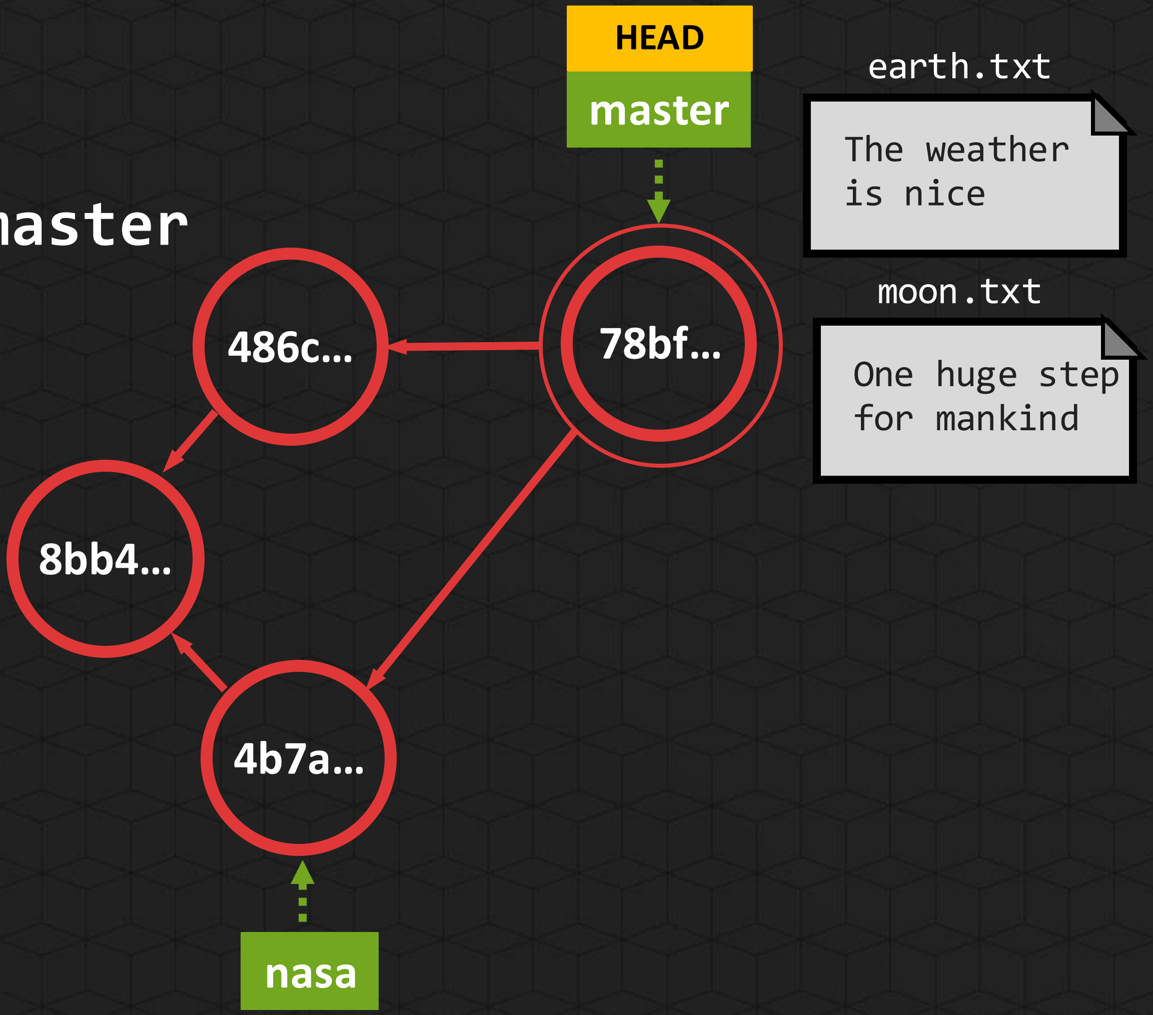


Let's **walk through** an example

- ❑ Let's now merge branch **nasa** into branch **master**
  - ❑ > `git merge nasa`

```
> git merge nasa
Merge made by the 'recursive' strategy.
 moon.txt | 1 +
 1 file changed, 1 insertion(+)
```

```
> git log --all --decorate --oneline --graph
*   78bfd90 (HEAD -> master) Merge branch 'nasa'
|\
| * 4b7ad68 (nasa) words of armstrong added
* | 486c105 weather update
|/
* 8bb4a59 earth and moon added
```



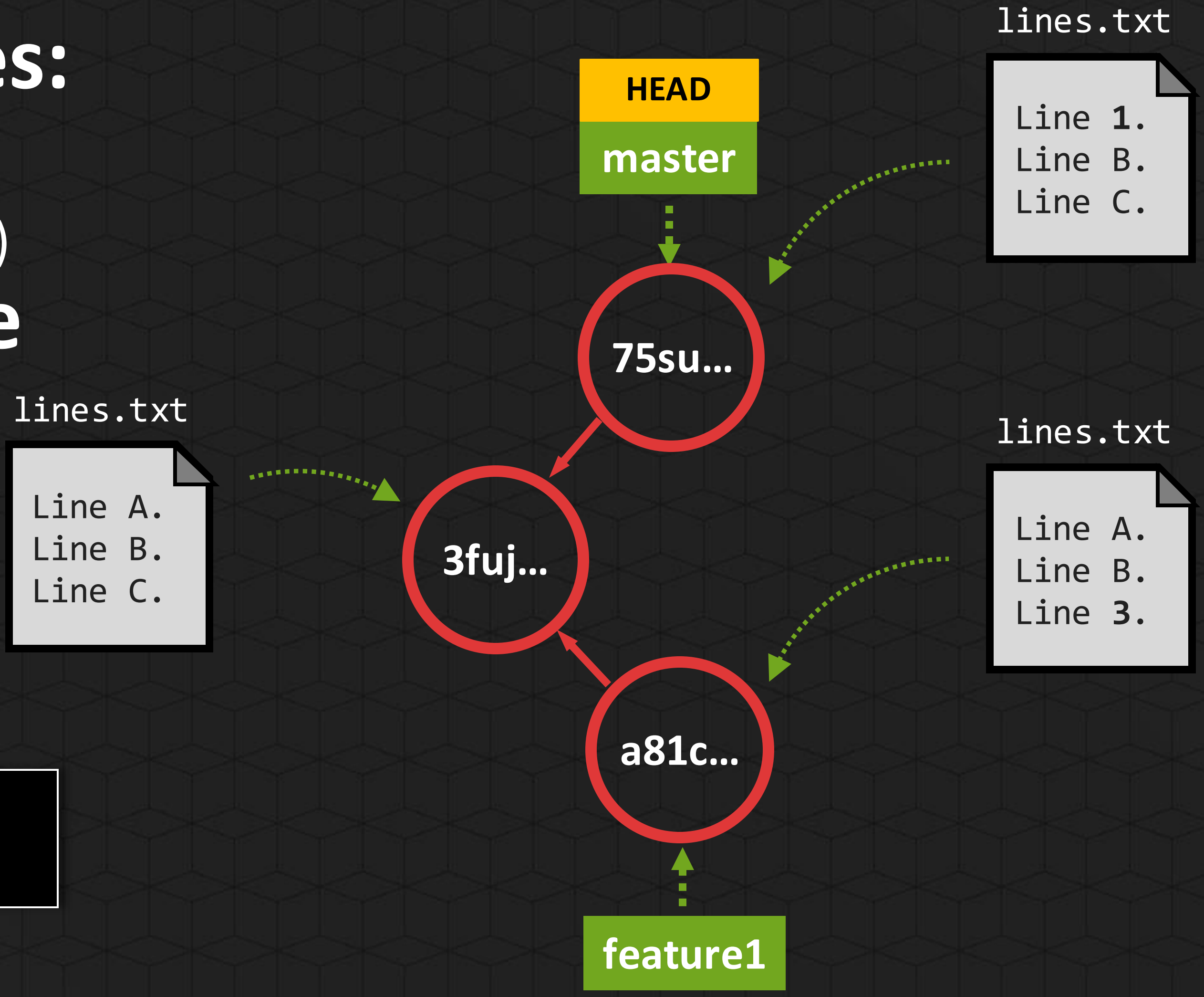
Merging branches: diverged history

(with no conflicting changes)

Three-way merge

This time, the changes are made in the same file, but in a different part. Therefore, the changes are still not conflicting.

```
git merge feature1
```



Merging branches: diverged history

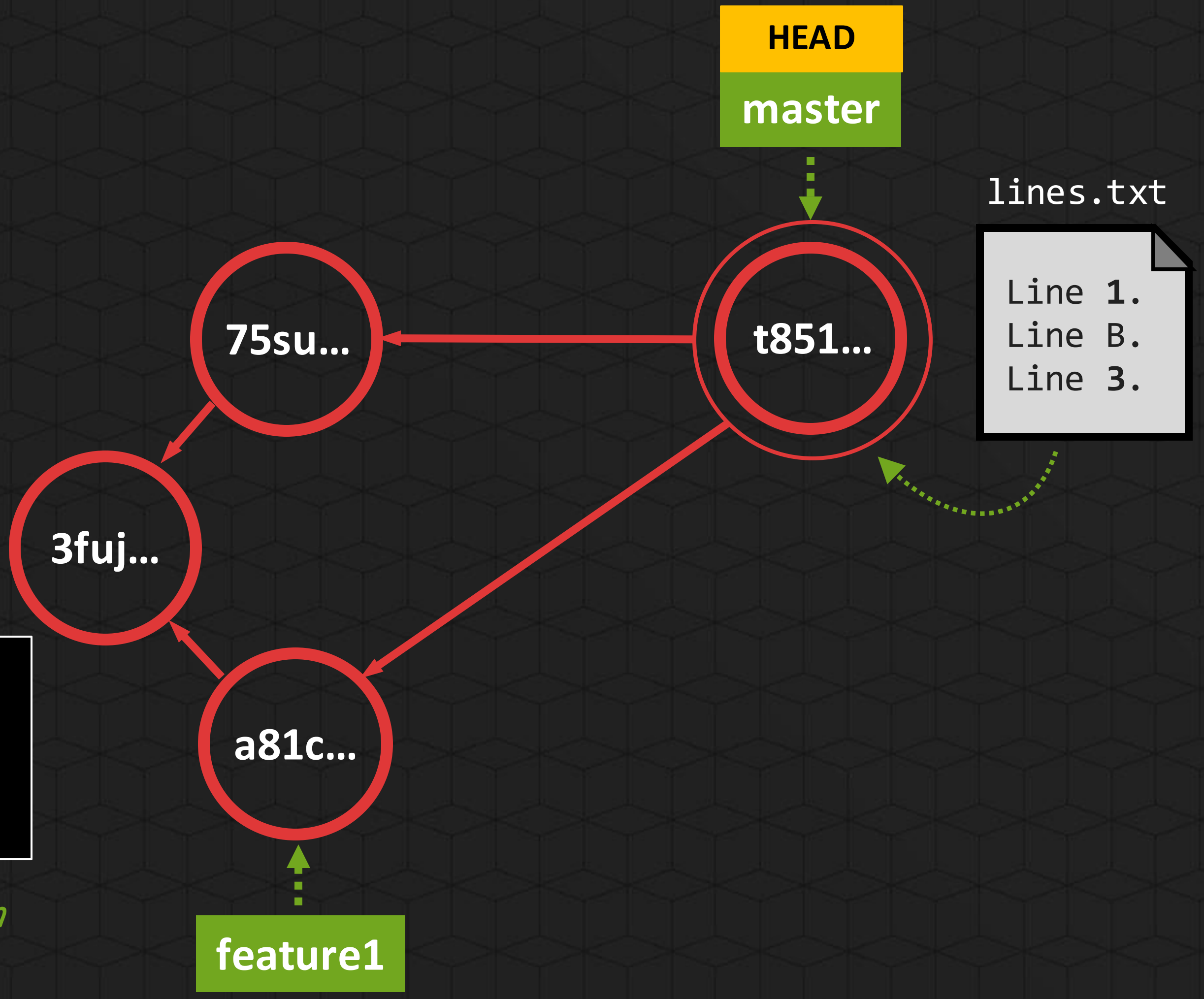
(with no conflicting changes)

Three-way merge

```
git merge feature1
```

```
Auto-merging lines.txt
Merge made by the 'recursive' strategy.
 lines.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git is able to auto-merge changes that were made in the same file, as long as they're made in different parts of the file.



When **merging two commits with a diverged history**, when there **are conflicting changes,** git will **not be able to automatically merge.**

If the changes are made in the same part of the same file, the changes ***will conflict****. and we as developers have to manually step in and resolve the conflict.*

Three-way merge

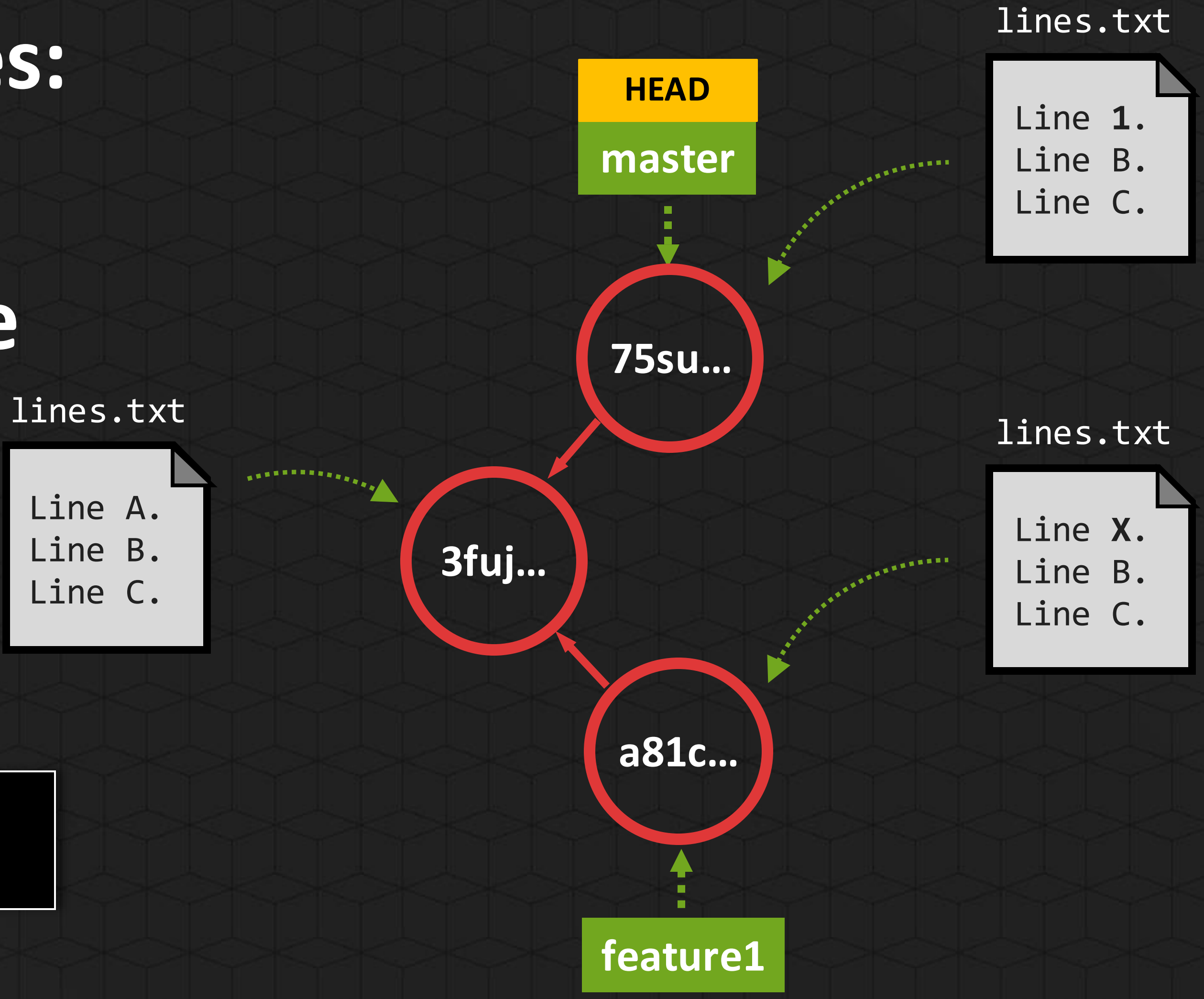
Merging branches: diverged history

(with conflicting changes)

Three-way merge

This time, the changes are made in the same file and in the same part. Git will not able to automatically merge.

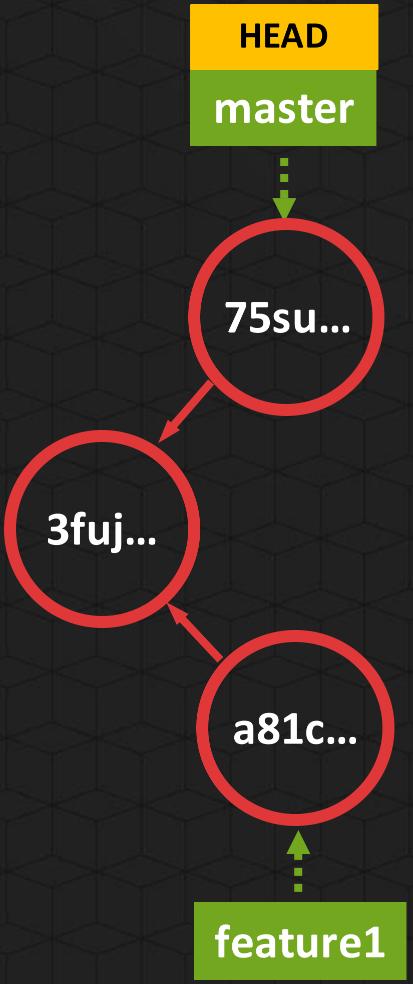
```
git merge feature1
```



Merging branches: diverged history
(with conflicting changes)
Three-way merge

```
git merge feature1
```

```
Auto-merging lines.txt
CONFLICT (content): Merge conflict in lines.txt
Automatic merge failed; fix conflicts and then
commit the result.
```



lines.txt

```
<<<<<<< HEAD
Line 1.
=======
Line X.
>>>>>>> feature1
Line B.
Line C.
```

Merging branches: diverged history

(with conflicting changes)

Three-way merge

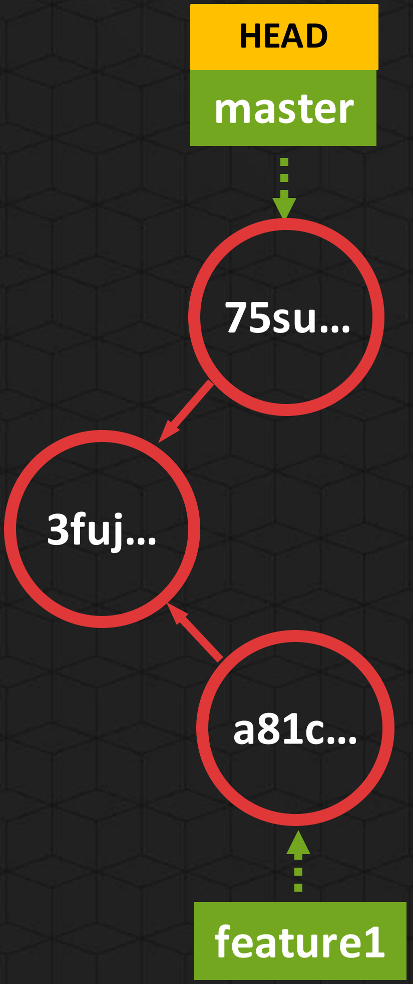
```
git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   lines.txt

no changes added to commit (use "git add"
   and/or "git commit -a")
```



lines.txt

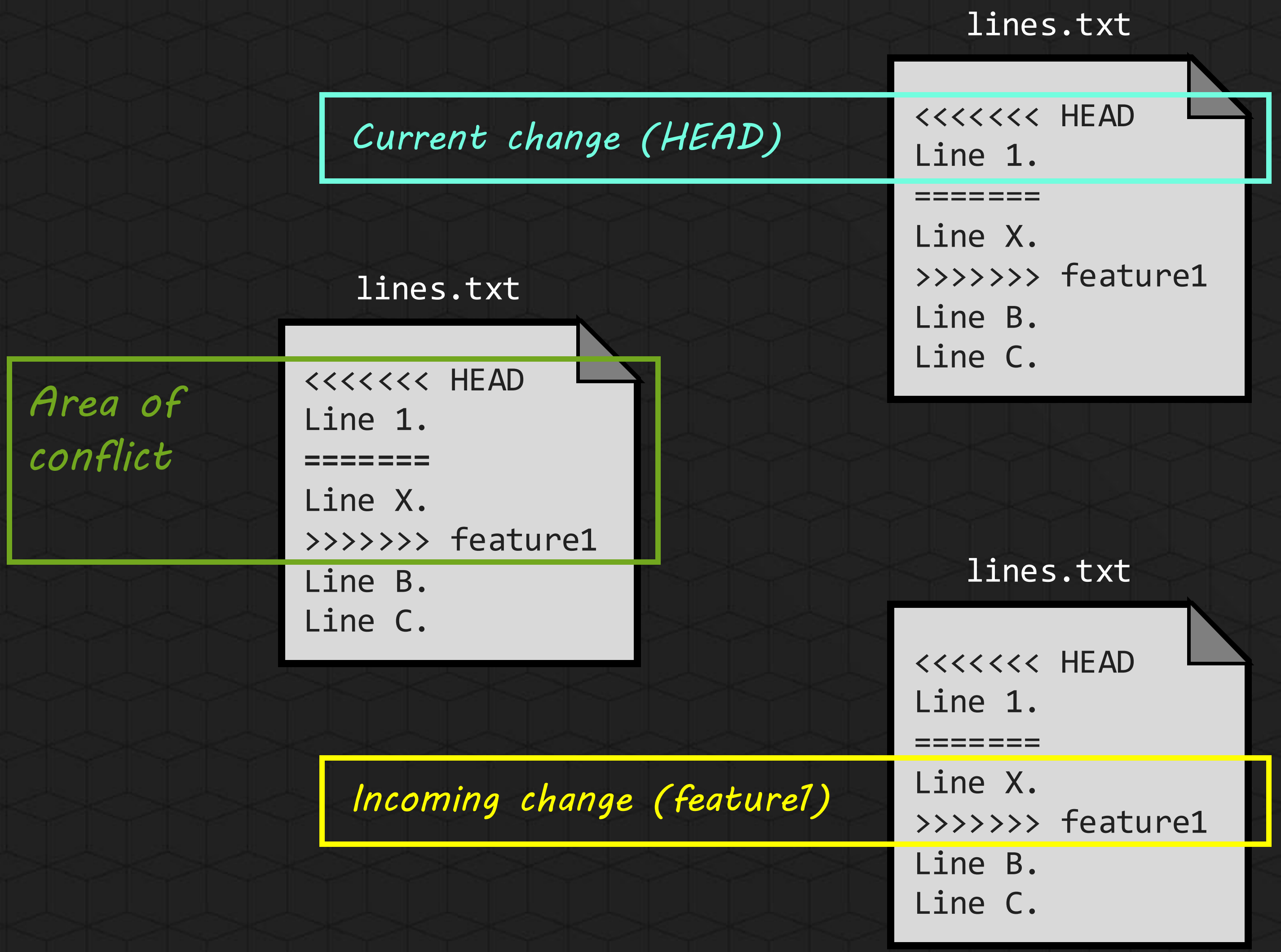
```
<<<<<<< HEAD
Line 1.
=======
Line X.
>>>>>>> feature1
Line B.
Line C.
```

Merging branches: diverged history

(with conflicting changes)

Three-way merge

Resolve the conflict



lines.txt

```
<<<<<<< HEAD
Line 1.
=======
Line X.
>>>>>>> feature1
Line B.
Line C.
```

lines.txt

```
<<<<<<< HEAD
Line 1.
=======
Line X.
>>>>>>> feature1
Line B.
Line C.
```

lines.txt

```
<<<<<<< HEAD
Line 1.
=======
Line X.
>>>>>>> feature1
Line B.
Line C.
```

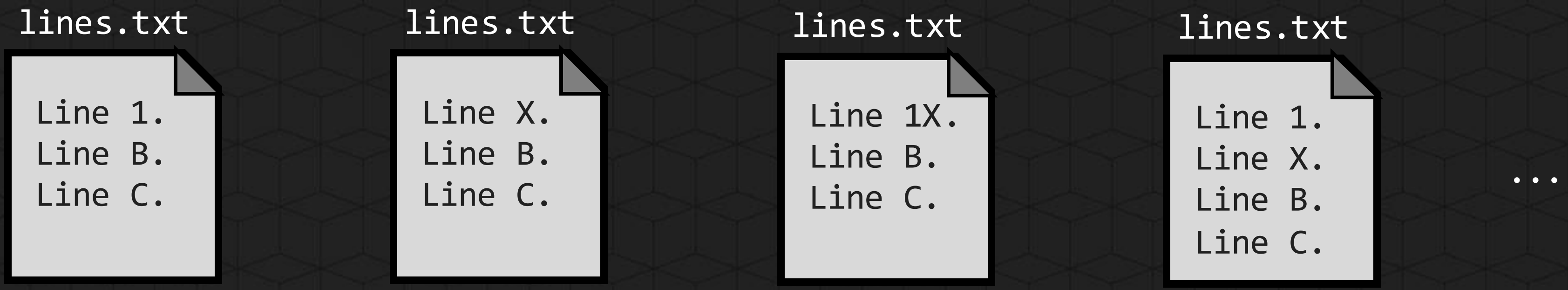
Merging branches: diverged history

(with conflicting changes)

Three-way merge

Resolve the conflict

by manually making a resolution.



lines.txt
```
Line 1.
Line B.
Line C.
```

lines.txt
```
Line X.
Line B.
Line C.
```

lines.txt
```
Line 1X.
Line B.
Line C.
```

lines.txt
```
Line 1.
Line X.
Line B.
Line C.
```

...

Merging branches: diverged history

(with conflicting changes)

Three-way merge

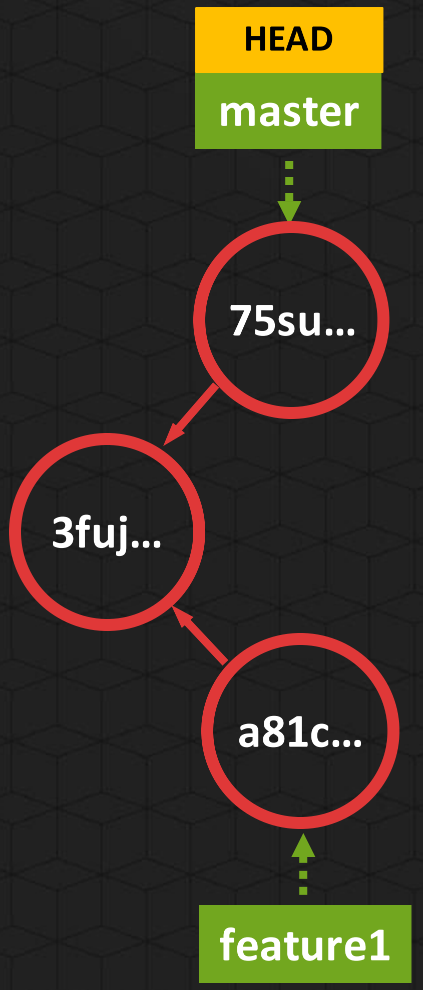
```
git add lines.txt
```

```
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   lines.txt
```

```
git commit -m “merged”
```



lines.txt

```
Line 1X.
Line B.
Line C.
```

We have manually resolved the conflict of our file and saved its changes to the disk.

Merging branches: diverged history

(with conflicting changes)

Three-way merge

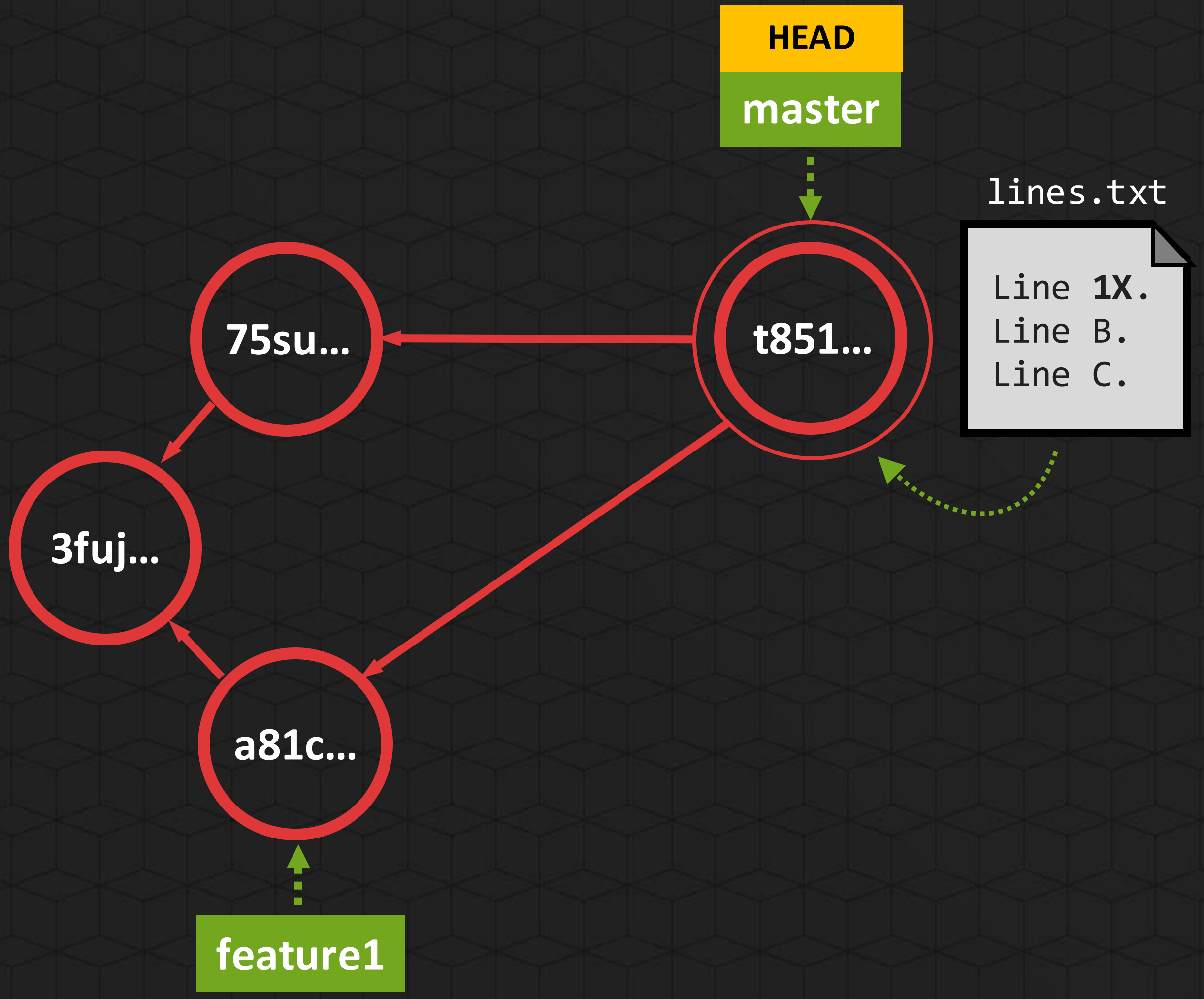
```
git add lines.txt
```

```
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   lines.txt
```

```
git commit –m “merged”
```



Remote branches

Chapter 3

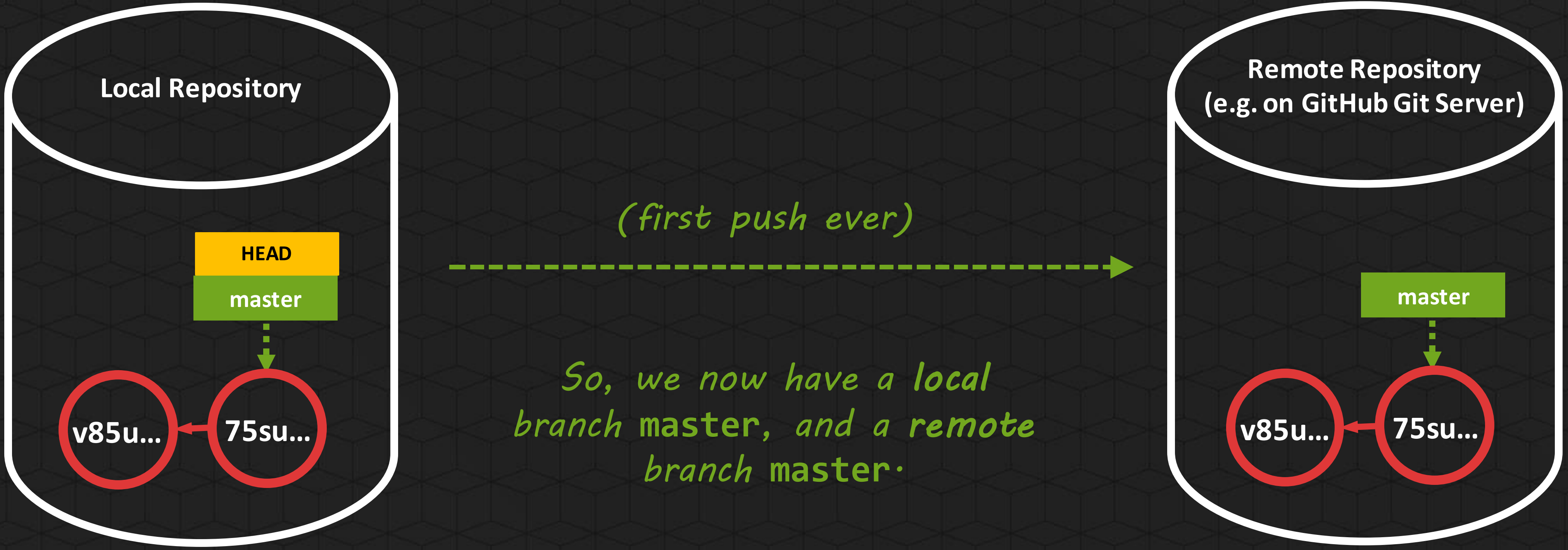
Overview

Remote branches

Introduction

Remember how a **branch** is nothing more than a **pointer / reference to a commit**.

When we **push our local changes to a remote**, **references,** such as **branches, can** be pushed as well.



*So, we now have a **local** branch* `master`*, and a **remote** branch* `master`*.*

Remote branches

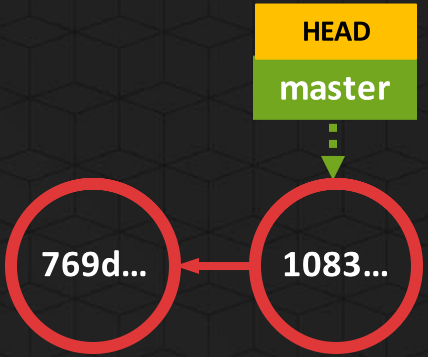
Remote-tracking branches

Locally, Git **keeps track** of **all branches**:

- ✓The **local branches.** *for which we can directly move the pointer by creating new commits*
  - ✓A local branch simply takes the form of its name, e.g. **`master`**
- ✓The **remote branches** by means of **pointers called remote-tracking branches.** *These pointers are only moved by Git when we fetch updates from the remote.*
  - ✓A remote-tracking branch takes the form of **`<remote>/<branch-name>`**, e.g. **`origin/master`**

Let’s start by looking at a **scenario** in which we have **initialized a new git repository.**

Local repository (of developer X)



Option -r shows the remote-tracking branches

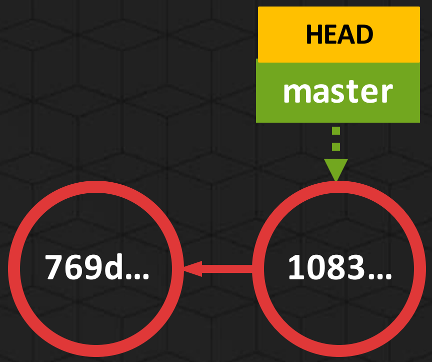
```
git branch
```

```
* master
```

```
git branch -r
```

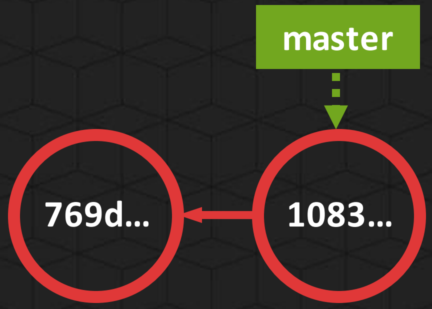
Remote repository (configured as origin)

Local repository (of developer X)

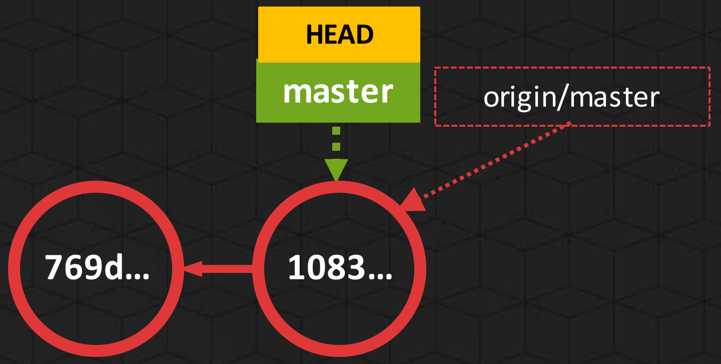


```
git push origin master
```

Remote repository (configured as origin)



Local repository (of developer X)



Option -r shows the remote-tracking branches

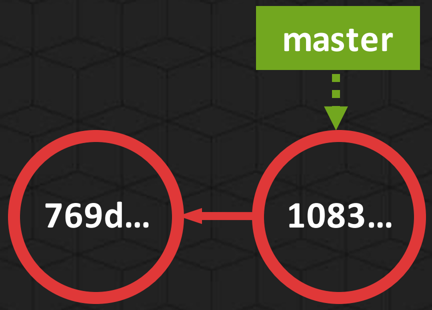
```
git branch
```

```
* master
```

```
git branch -r
```

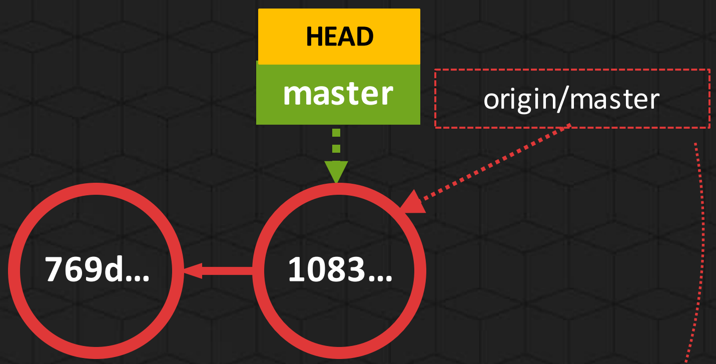
```
origin/master
```

Remote repository (configured as origin)



Remote-tracking branches

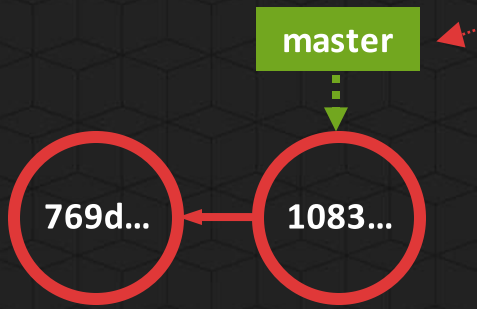
Local repository (of developer X)



```
> git log --all --decorate --oneline --graph
* 1083751 (HEAD -> master, origin/master) second commit
* 769d6ff initial commit
```

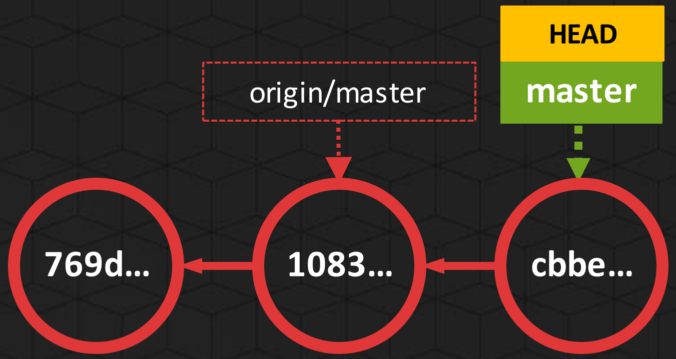
Shows, on your local repository, the position of remote branch **master** *(on remote* **origin***)*

Remote repository (configured as origin)



Remote-tracking branches

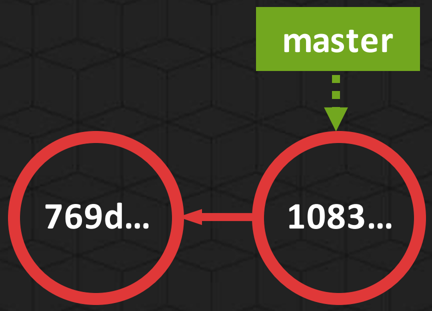
Local repository (of developer X)



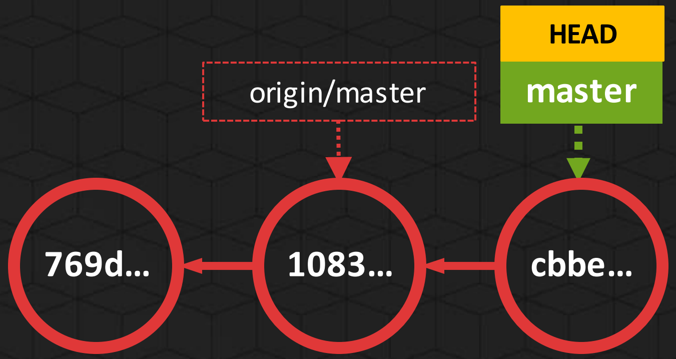
Now, imagine we made some changes and created a new commit

```
> git log --all --decorate --oneline --graph
* cbbe1f6 (HEAD -> master) third commit
* 1083751 (origin/master) second commit
* 769d6ff initial commit
```

Remote repository (configured as origin)

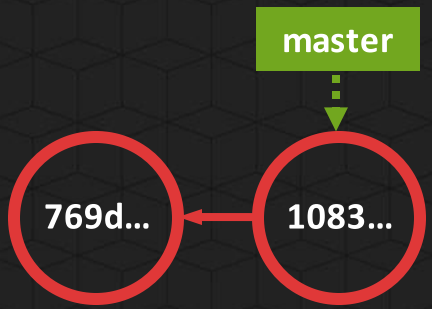


Local repository (of developer X)



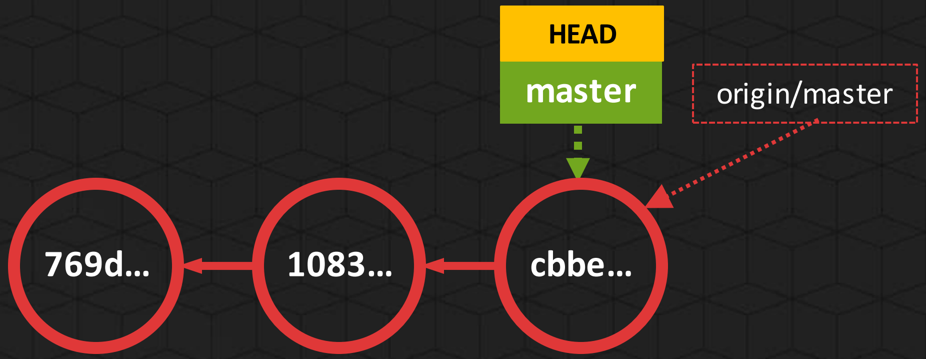
```
git push origin master
```

Remote repository (configured as origin)



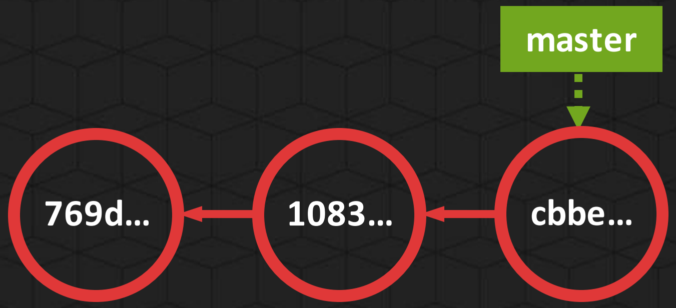
Remote-tracking branches

Local repository (of developer X)

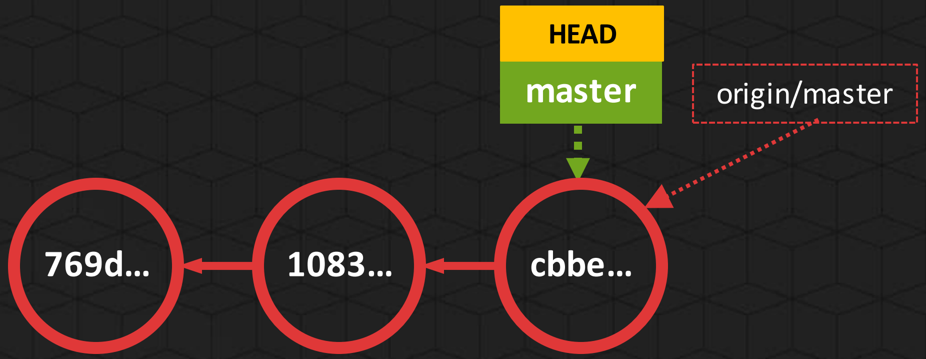


```
git push origin master
```

Remote repository (configured as origin)

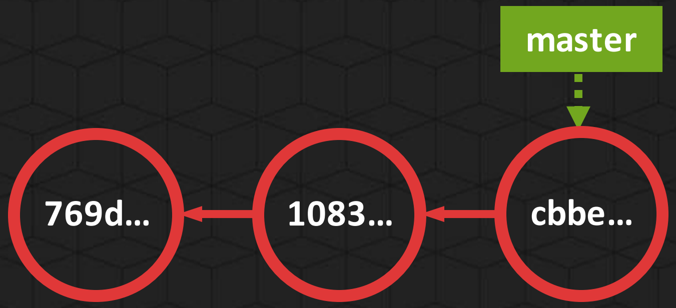


Local repository (of developer X)



```
> git log --all --decorate --oneline --graph
* cbbe1f6 (HEAD -> master, origin/master) third commit
* 1083751 second commit
* 769d6ff initial commit
```

Remote repository (configured as origin)



Remote-tracking branches

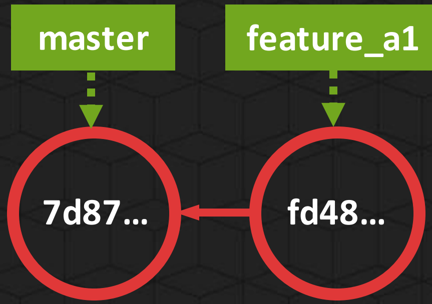
Now, let’s look at a **different scenario**. One in which we **clone an already existing remote repository** that has **multiple branches**.

Remote-tracking branches

Local repository (of developer X)

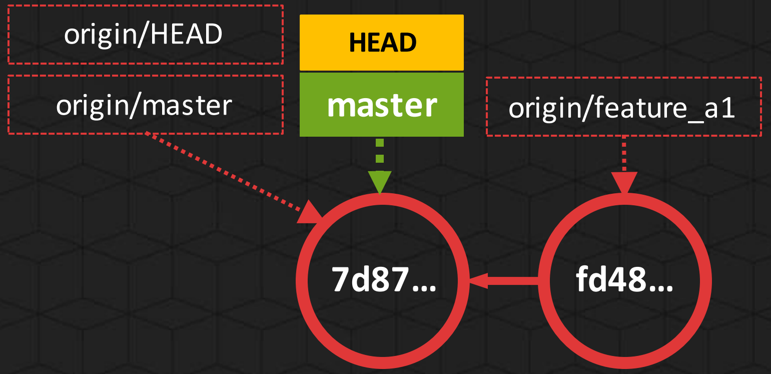
Each time there's communication between the local repository and a remote repository (from remote → local: cloning, pulling, fetching), the remote-tracking branches get updated / synchronized (thus possibly moved) by git.

Remote repository (configured as `origin`)



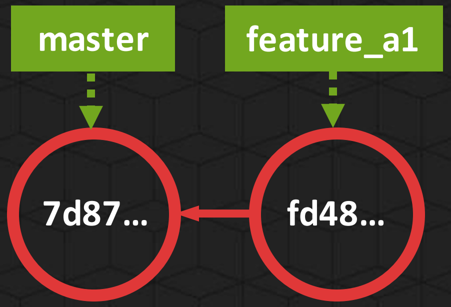
Remote-tracking branches

Local repository (of developer X)

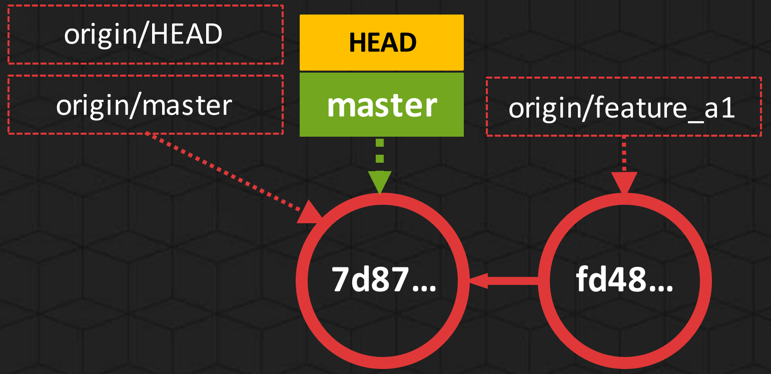


```
git clone <remote-repo-url>
```

Remote repository (configured as origin)



Local repository (of developer X)



Option -r shows the remote-tracking branches

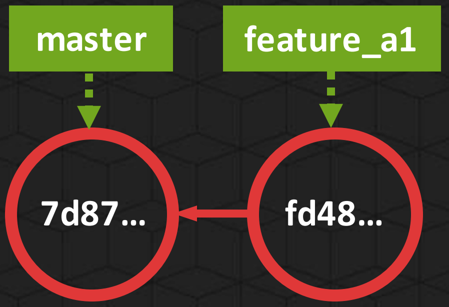
```
git branch
```

```
* master
```

```
git branch -r
```

```
  origin/HEAD -> origin/master
  origin/feature_a1
  origin/master
```

Remote repository (configured as origin)

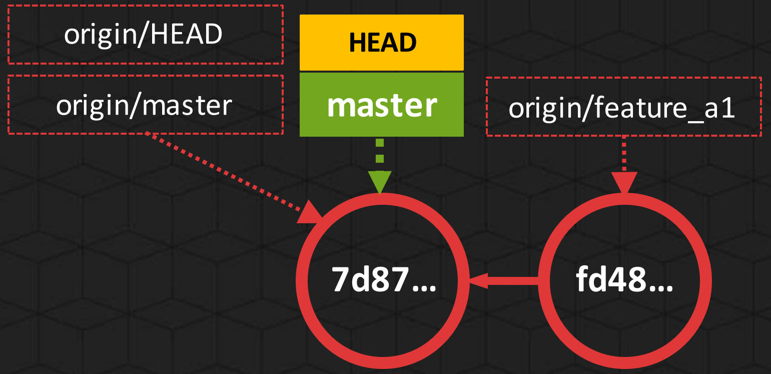


Locally, after cloning, only a local branch of the remote branch to which **`origin/HEAD`** *is pointing at, is created. By default, this is the* **`master`** *branch.*

We don't yet have a new local branch **`feature_a1`**. *We only have the remote-tracking branch* **`origin/feature_a1`** *which we can't modify.*

Remote-tracking branches

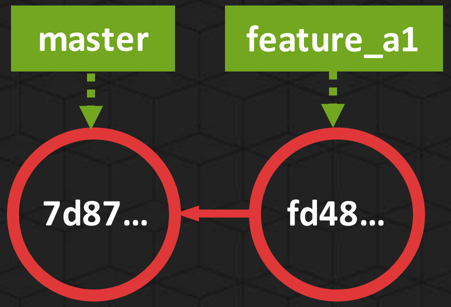
Local repository (of developer X)



```
git checkout -b feature_a1 origin/feature_a1
```

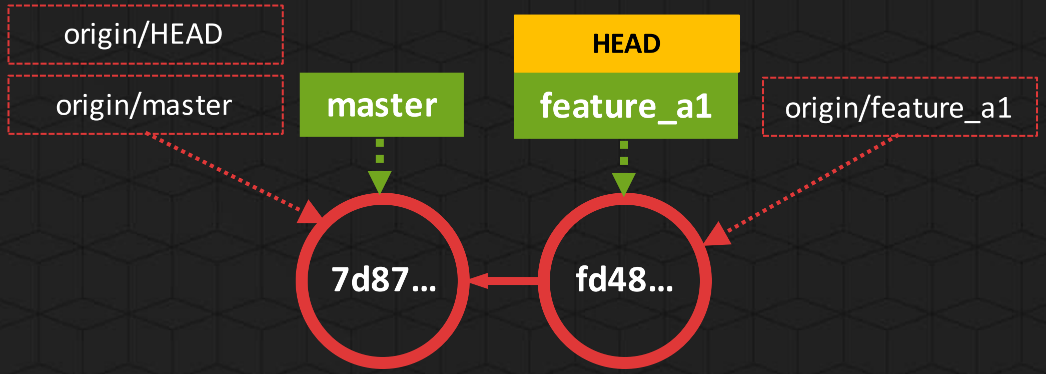
```
Switched to a new branch 'feature_a1'
Branch 'feature_a1' set up to track remote
branch 'feature_a1' from 'origin'.
```

Remote repository (configured as origin)



Remote-tracking branches

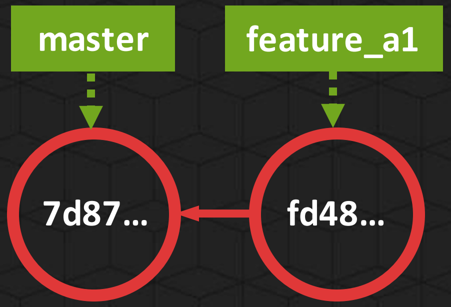
Local repository (of developer X)



```
git checkout -b feature_a1 origin/feature_a1
```

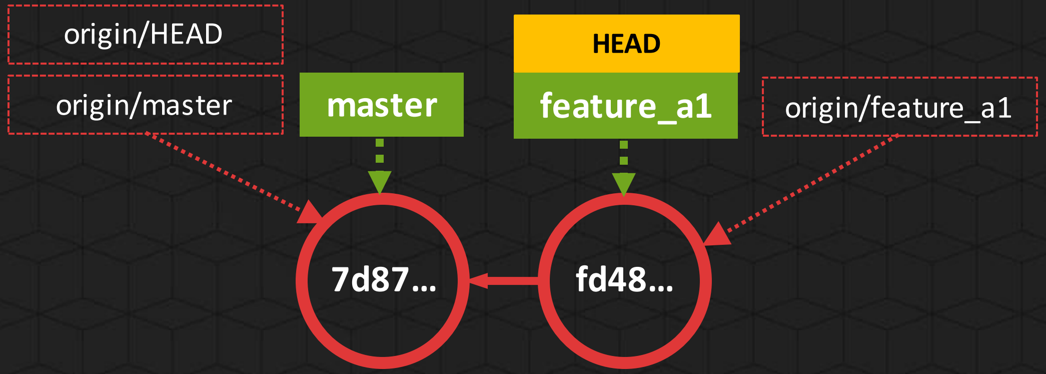
```
Switched to a new branch 'feature_a1'
Branch 'feature_a1' set up to track remote
branch 'feature_a1' from 'origin'.
```

Remote repository (configured as origin)



Remote-tracking branches

Local repository (of developer X)

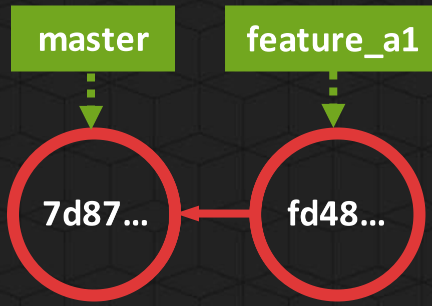


```
git checkout -b feature_a1 origin/feature_a1
```

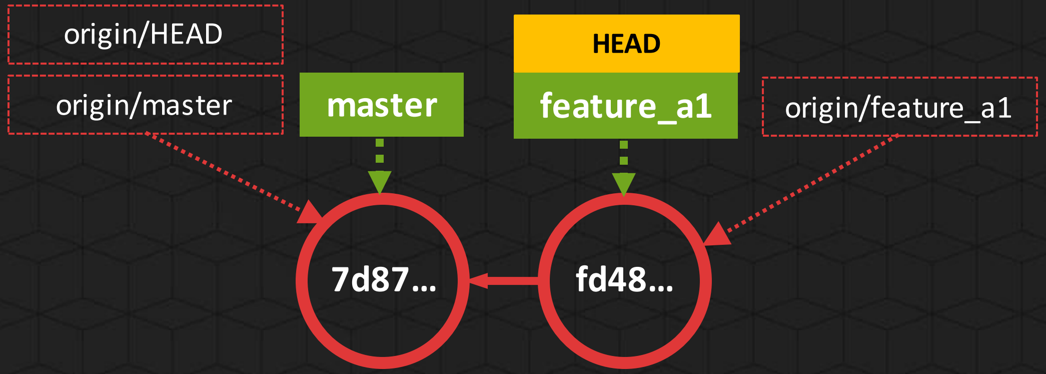
```
git checkout feature_a1
```

This simplified command does the exact same thing ***if*** *the branch does not yet exist locally* ***and if*** *its name matches that of (only) one remote-tracking branch.*

Remote repository (configured as origin)



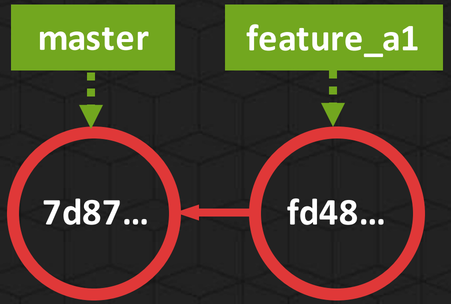
Local repository (of developer X)



```
git branch
```

```
* feature_a1
  master
```

Remote repository (configured as origin)



Remote-tracking branches

Remote branches

Tracking branches

Almost always, a **local branch** should have a **direct, one-on-one relationship to a remote branch**.

In Git, we can explicitly **set that relationship** by **creating a tracking branch** out of the **local branch**. The **remote branch being tracked** is called the **upstream branch.**

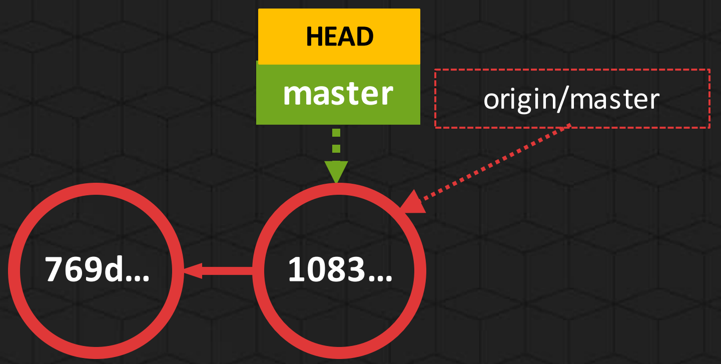
Creating tracking branches comes with a few benefits!

Tracking branches

Let’s start with a **scenario** in which we **initialized a local repository**, made 2 commits and pushed them once to the **`origin`** remote.

Tracking branches

Local repository (of developer X)



Option -r shows the remote-tracking branches

Option -vv shows info about the branch + the set upstream branch (if any)

```
git branch
```

```
* master
```

```
git branch -r
```

```
origin/master
```

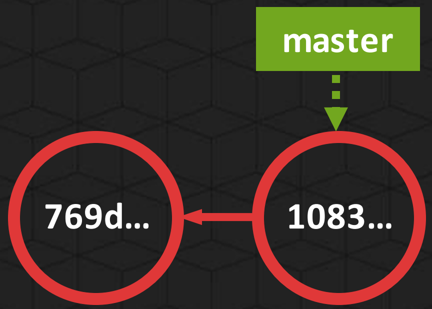
```
git branch -vv
```

```
* master 1083568 second commit
```

```
git status
```

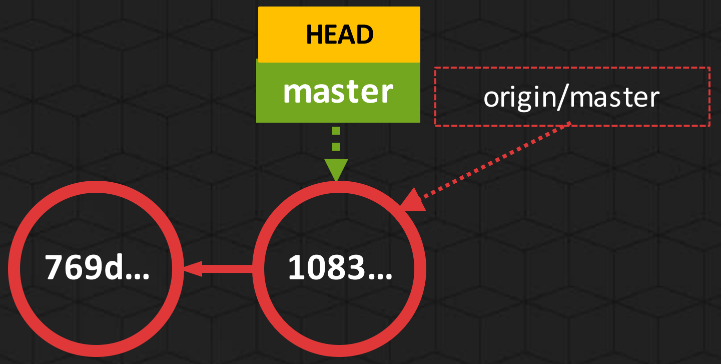
```
On branch master
nothing to commit, working tree clean
```

Remote repository (configured as origin)



Without tracking branch

Local repository (of developer X)

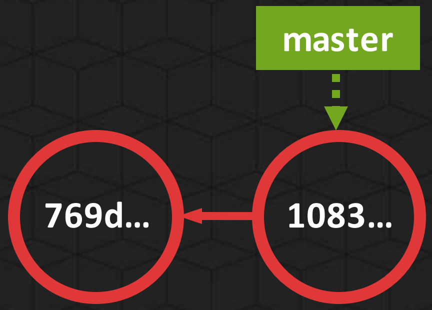


Sets up local branch master to track remote branch master on origin. (option --set-upstream-to is equivalent to -u)

```
git branch -u origin/master
```

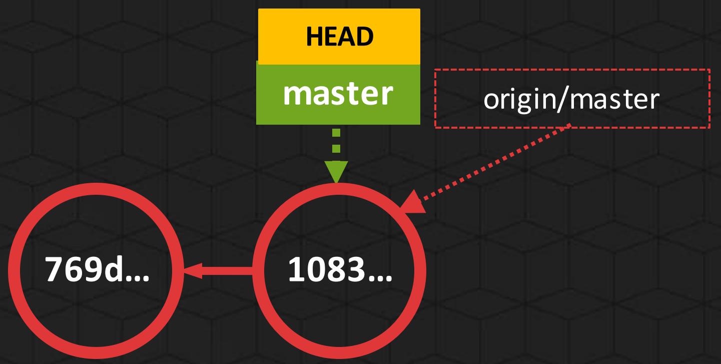
```
Branch 'master' set up to track remote branch
'master' from 'origin'.
```

Remote repository (configured as origin)



Setting up a tracking branch

Local repository (of developer X)



Option -vv shows info about the branch + the set upstream branch

```
git branch -vv
```

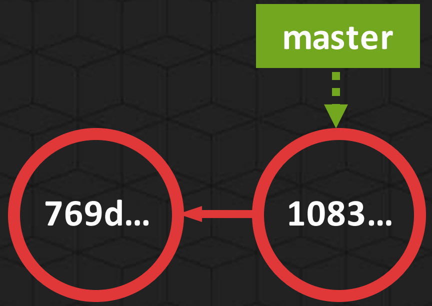
```
* master 1083568 [origin/master] second commit
```

```
git status
```

```
On branch master
Your branch is up to date with
'origin/master'.

nothing to commit, working tree clean
```

Remote repository (configured as origin)



With tracking branch

Tracking branches

Without tracking branch

```
git branch -vv
```

```
* master 1083568 second commit
```

```
git status
```

```
On branch master

nothing to commit, working tree clean
```

```
git push origin master
```

```
git pull origin master
```

With tracking branch

```
git branch -vv
```

```
* master 1083568 [origin/master] second commit
```

```
git status
```

```
On branch master
Your branch is up to date with
'origin/master'.

nothing to commit, working tree clean
```

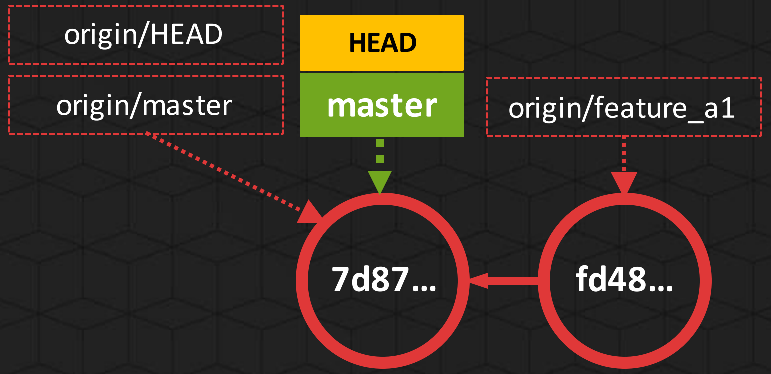
```
git push
```

```
git pull
```

Tracking branches

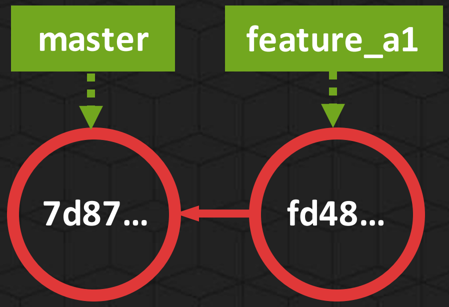
When **cloning** an already existing repository, a **tracking branch is automatically created** for the checked out branch (by default: `master`)

Local repository (of developer X)



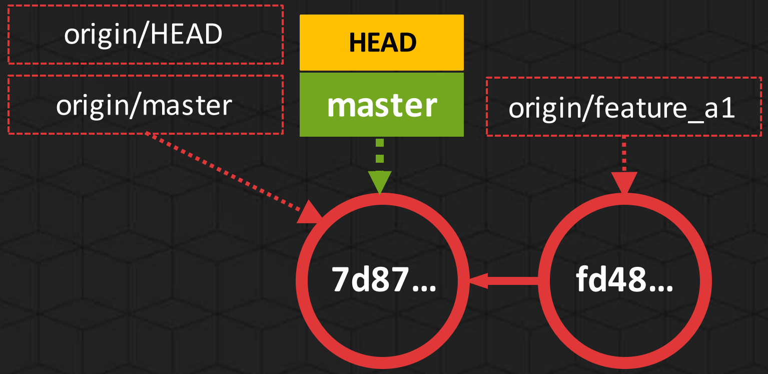
```
git clone <remote-repo-url>
```

Remote repository (configured as origin)



Tracking branches

Local repository (of developer X)

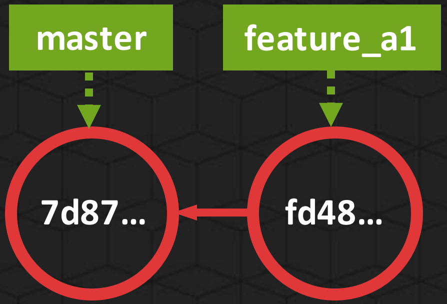


Option -vv shows info about the branch + the set upstream branch

```
git branch -vv
```

```
* master 7d87fea [origin/master] calculations added
```

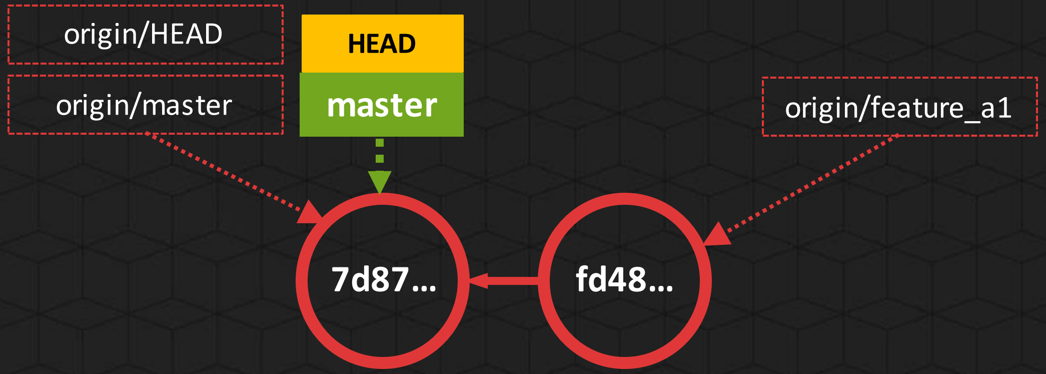
Remote repository (configured as origin)



Tracking branches

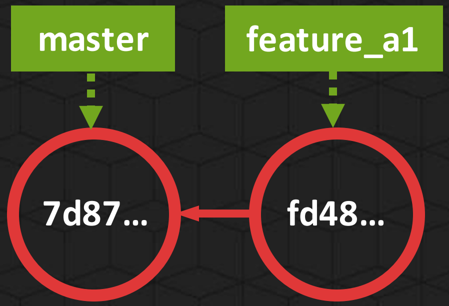
Creating a local branch based of a **remote-tracking branch** will **automatically create** a **tracking branch**.

Local repository (of developer X)



```
git checkout -b feature_a1 origin/feature_a1
```
or
```
git checkout --track origin/feature_a1
```
or
```
git checkout feature_a1
```

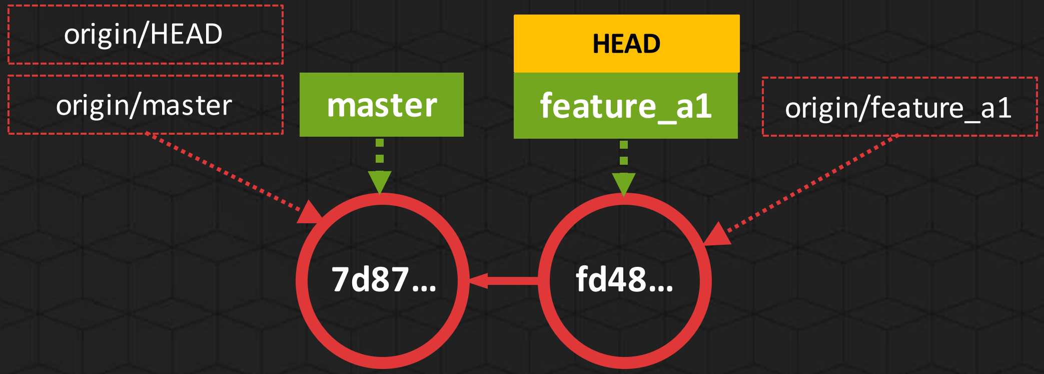
Remote repository (configured as origin)



Remote branches

Tracking branches

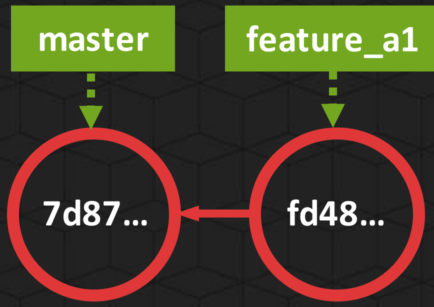
Local repository (of developer X)



```
git branch -vv
```

```
* feature_a1 f48bad9 [origin/feature_a1] feature implemented
  master     7d87fea [origin/master] calculations added
```

Remote repository (configured as origin)



Remote branches

Deleting a remote branch

We already saw how to **delete a local branch**

```
git branch -d <branch-name>
```

```
git branch -D <branch-name>
```

All this does is removing the selected branch (which is nothing more than a pointer that points to a certain commit).

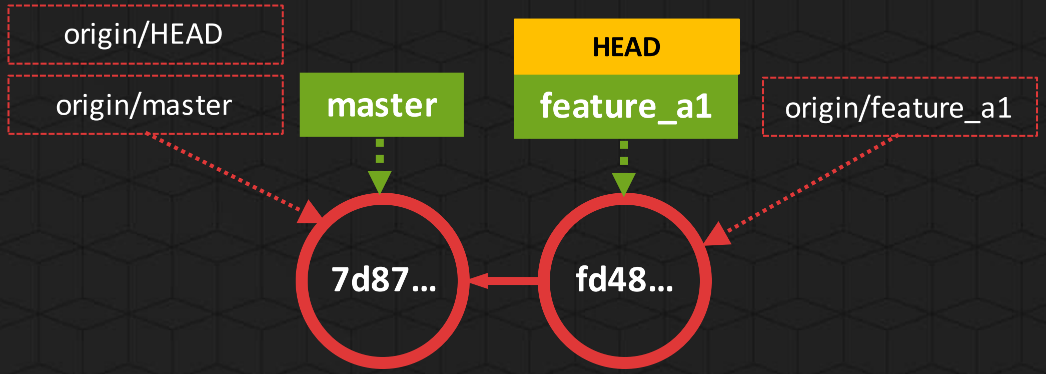
To **delete a remote branch**,
you use the following command

```
git push origin --delete <branch-name>
```

All this does is removing the selected remote branch (which is nothing more than a remote pointer that points to a certain commit) from the specified remote repository

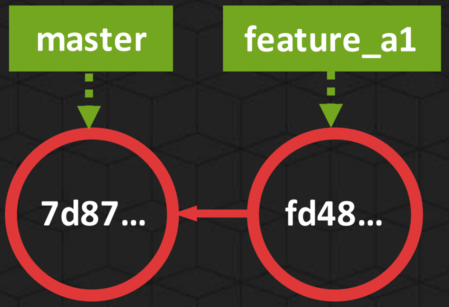
Deleting a remote branch

Local repository (of developer X)

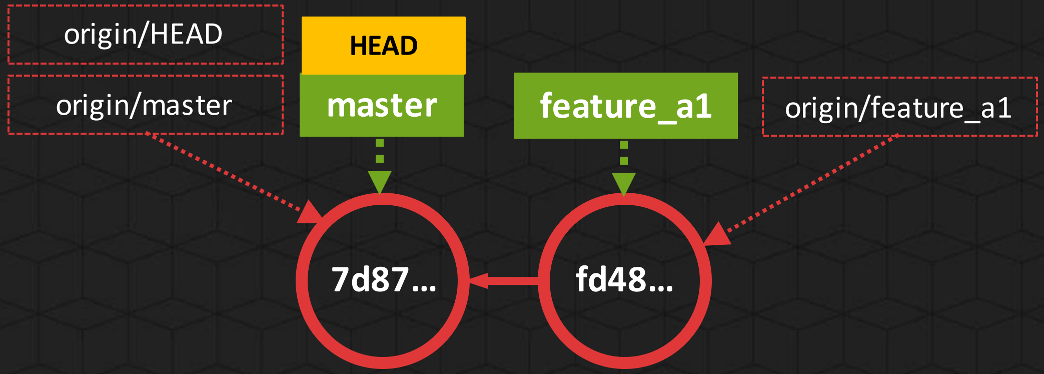


```
git checkout master
```

Remote repository (configured as origin)



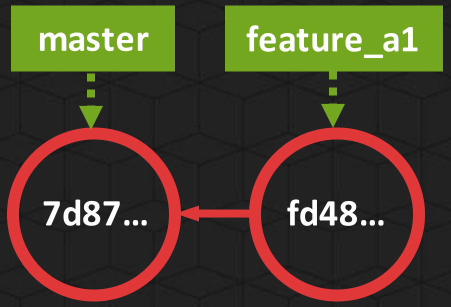
Local repository (of developer X)



```
git checkout master
```

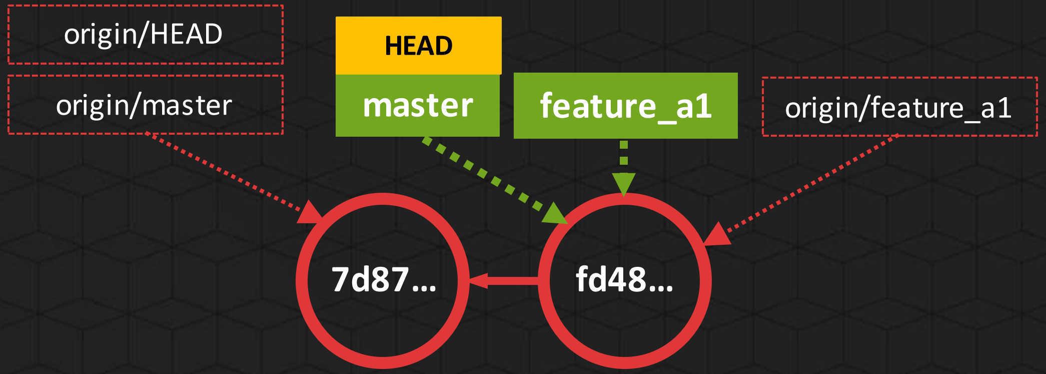
```
git merge feature_a1
```

Remote repository (configured as origin)



Deleting a remote branch

Local repository (of developer X)

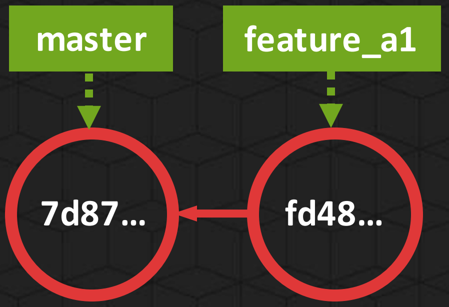


```
git checkout master
```

```
git merge feature_a1
```

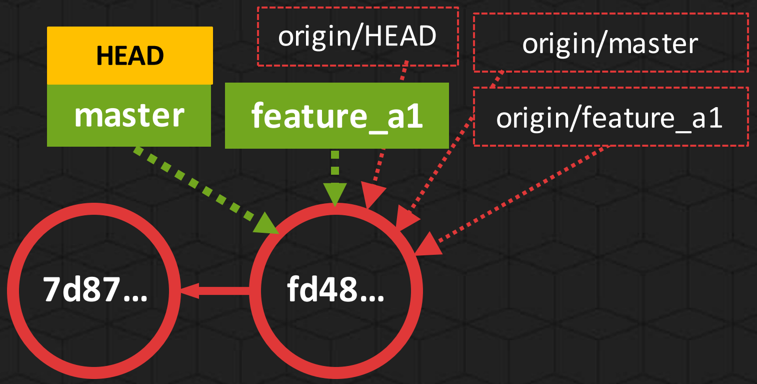
```
git push origin master
```

Remote repository (configured as origin)



Deleting a remote branch

Local repository (of developer X)

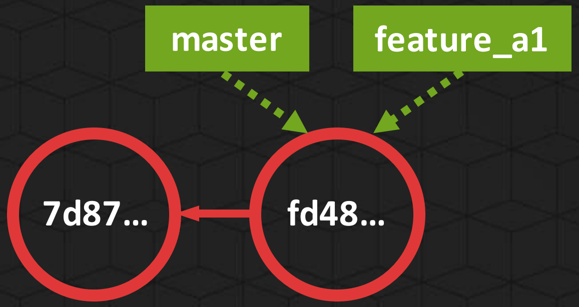


```
git checkout master
```

```
git merge feature_a1
```

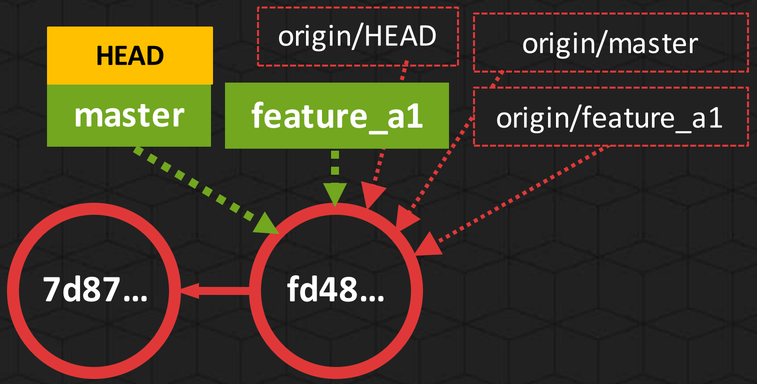
```
git push origin master
```

Remote repository (configured as origin)



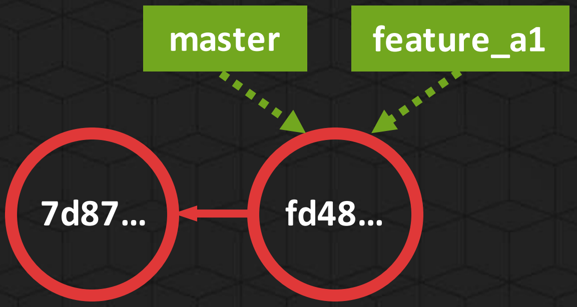
Deleting a remote branch

Local repository (of developer X)



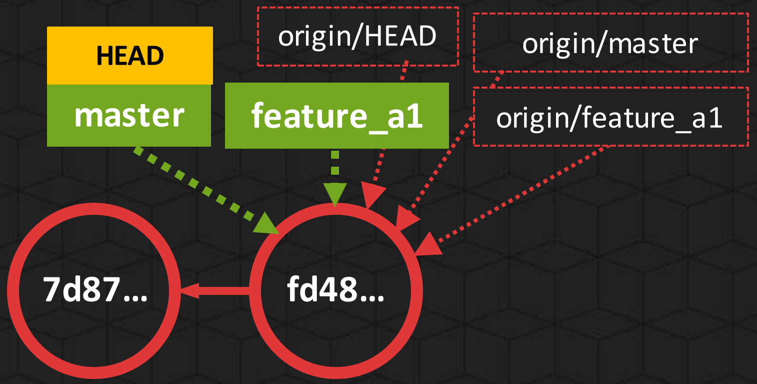
```
> git log --all --decorate --oneline --graph
* f48bad9 (HEAD -> master, origin/master, origin/feature_a1,
          origin/HEAD, feature_a1) feature implemented
* 7d87fea calculations added
```

Remote repository (configured as origin)



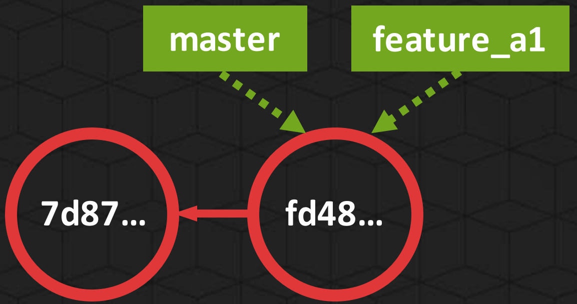
Deleting a remote branch

Local repository (of developer X)



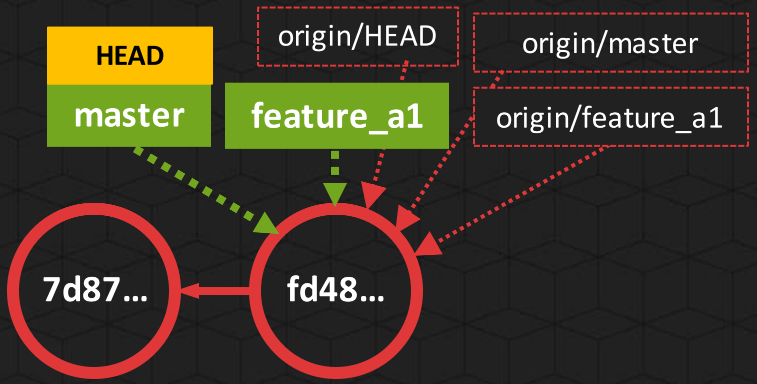
All of our changes made on the **`feature_a1`** branch are merged into our **`master`** branch. Now imagine we no longer need that **`feature_a1`** branch as the feature it contained is completely finished and merged into the **`master`**. We, nor any other developers will work on the **`feature_a1`** branch anymore. Thus, we should/could remove it: both locally and remotely.

Remote repository (configured as `origin`)



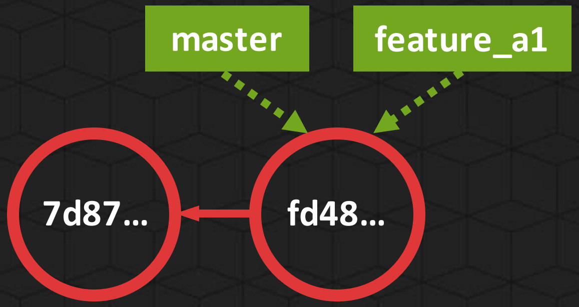
Deleting a remote branch

Local repository (of developer X)



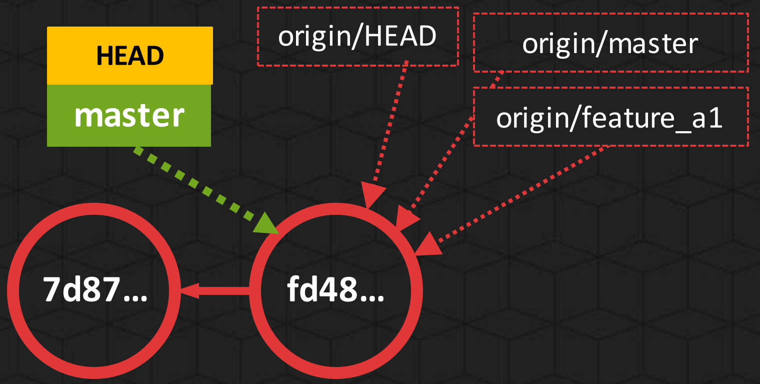
```
git branch -d feature_a1
```

Remote repository (configured as origin)



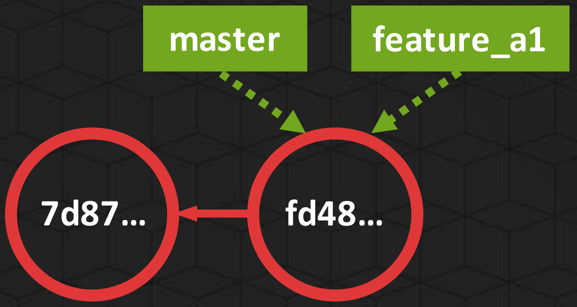
Deleting a remote branch

Local repository (of developer X)



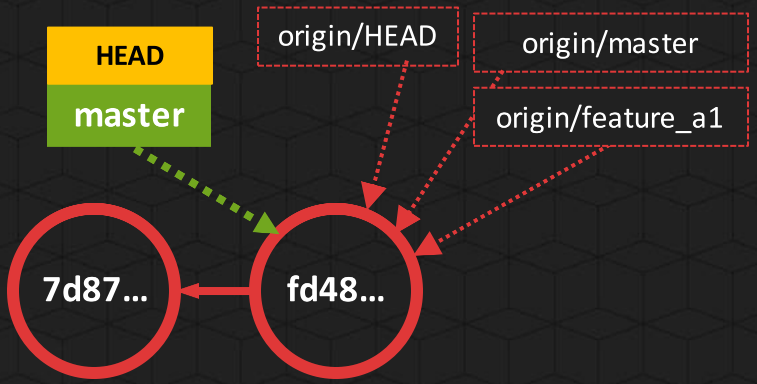
```
git branch -d feature_a1
```

Remote repository (configured as origin)



Deleting a remote branch

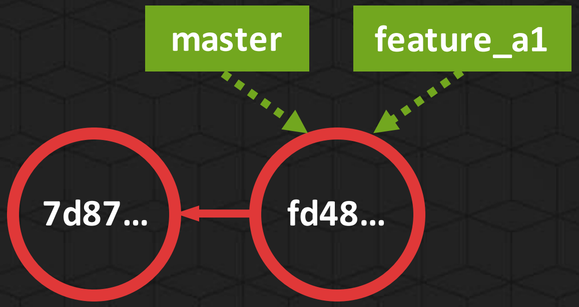
Local repository (of developer X)



```
git branch -d feature_a1
```

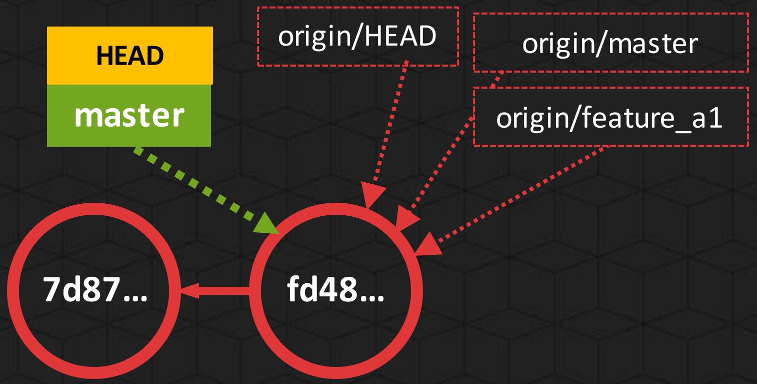
```
git push origin --delete feature_a1
```

Remote repository (configured as origin)



Deleting a remote branch

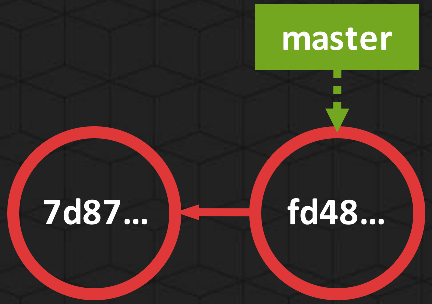
Local repository (of developer X)



```
git branch -d feature_a1
```

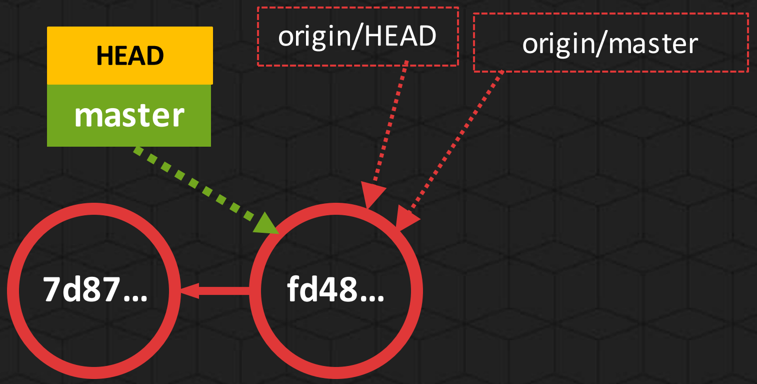
```
git push origin --delete feature_a1
```

Remote repository (configured as `origin`)



Deleting a remote branch

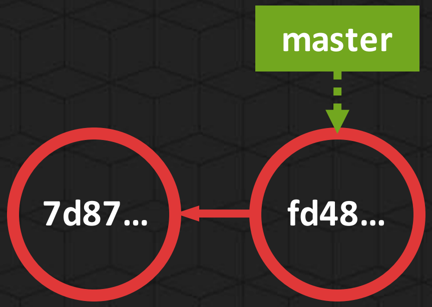
Local repository (of developer X)



```
git branch -d feature_a1
```

```
git push origin --delete feature_a1
```

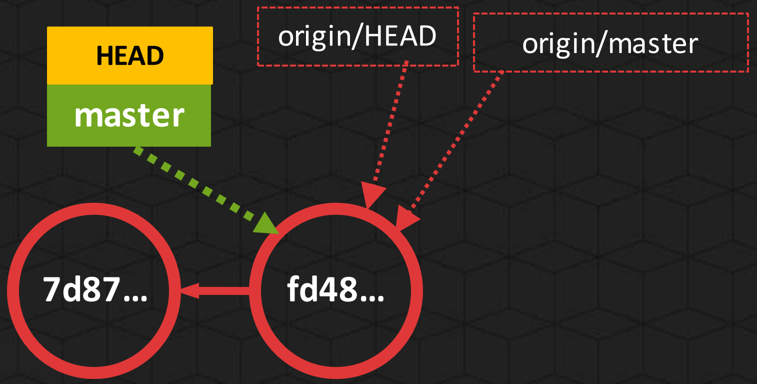
Remote repository (configured as `origin`)



Remote branches

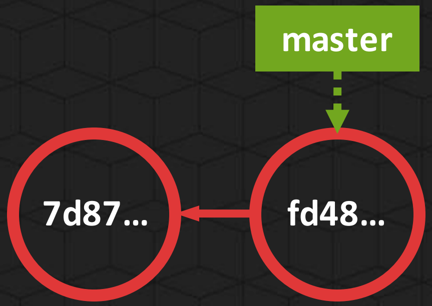
Deleting a remote branch

Local repository (of developer X)



```
> git log --all --decorate --oneline --graph
* f48bad9 (HEAD -> master, origin/master, origin/HEAD) feature implemented
* 7d87fea calculations added
```

Remote repository (configured as origin)



Deleting a remote branch

Push & Pull revisited

Chapter 4

Overview

Push & Pull revisited

Git push revisited

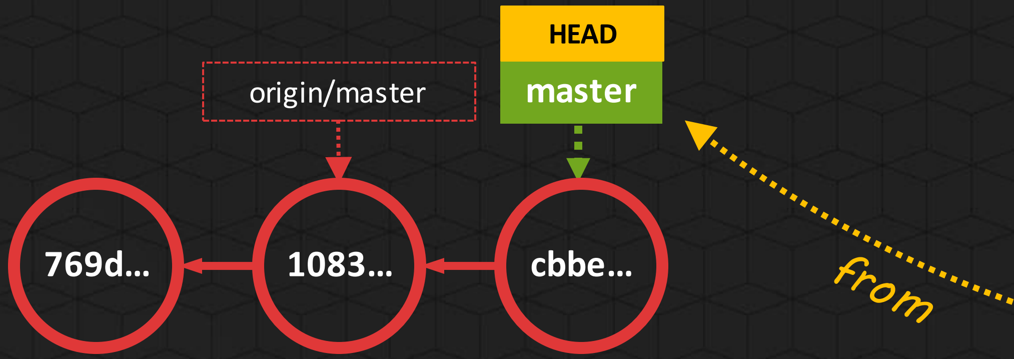
The **Push command**

```
git push [<remote> [<src>][:<dst>]]
```

Pushes the specified branch along with the commits (+ the newly created blob-objects) to the specified remote repository.

- Argument `<remote>` is the name of the remote repository we want to push to.
- Argument `<src>` specifies from which local **branch** we want to push
- Argument `:<dst>` specifies to which remote **branch** we want to push
  - Both are part of the **`<refspec>`** option. Leaving out `:<dst>` will push from the local branch to the remote branch with the same name.

Local repository (of developer X)

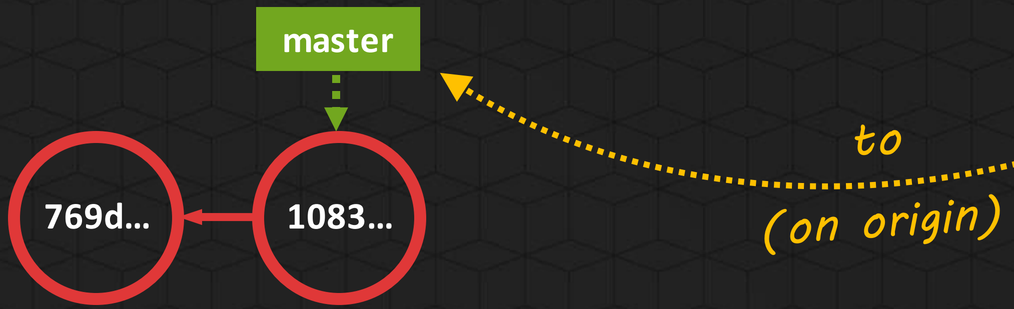


```
git push origin master
```

```
git push origin master:master
```

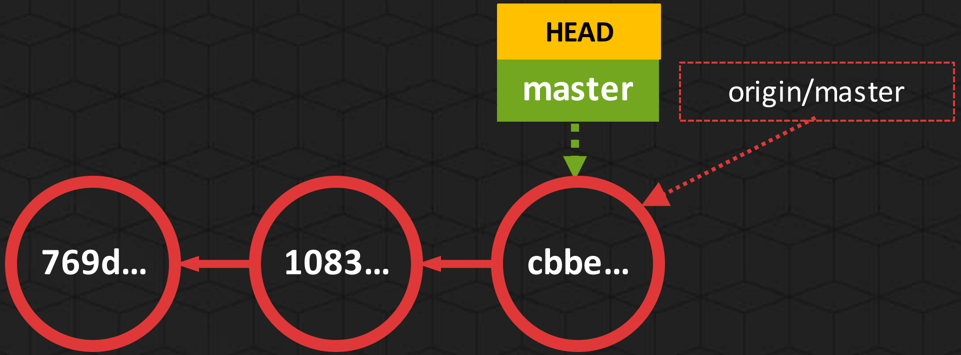
Both commands do the exact same thing

Remote repository (configured as origin)



Git push revisited

Local repository (of developer X)

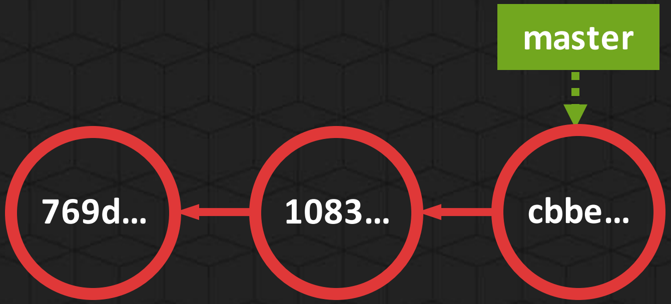


```
git push origin master
```

```
git push origin master:master
```

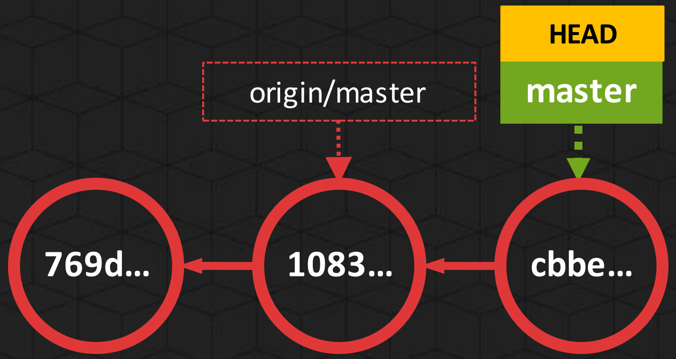
Both commands do the exact same thing

Remote repository (configured as origin)



Git push revisited

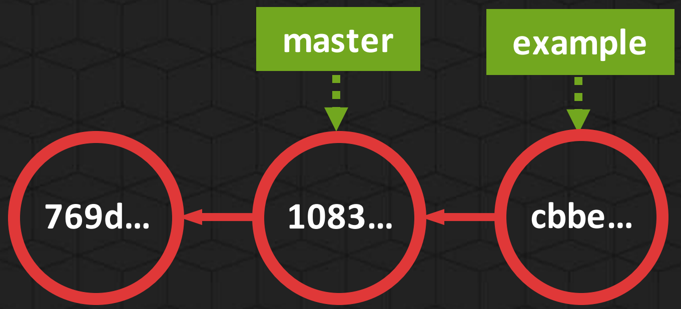
Local repository (of developer X)



```
git push origin master:example
```

Thus, it's possible to push to a remote branch with a different name. The branch will be created if it doesn't exist.

Remote repository (configured as origin)



Disclaimer: not often a valid use case!

(e.g. you made a commit on the wrong local branch, it contains valid changes but they should have been made on another branch. You could then push the changes to the correct remote branch, then reset your current local branch to its previous state)

Git push revisited

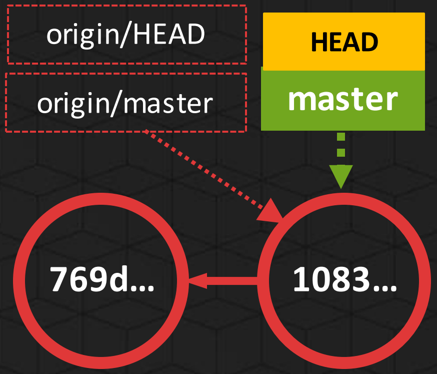
Push & Pull revisited

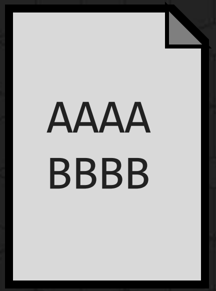
Git pull revisited

`git pull` does **2 things**: first it `fetches` changes from a (specified) remote (and a specified branch). Then, it `merges` the (selected) changes into the current (checked out) branch.

Git pull revisited

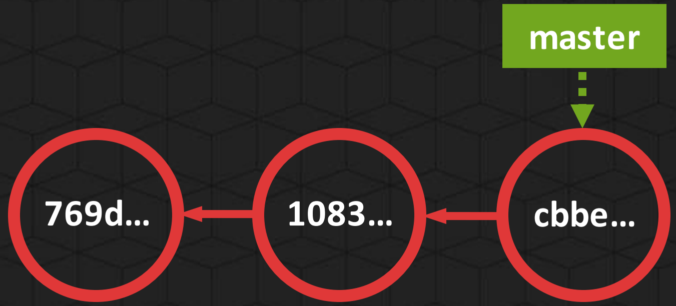
Local repository (of developer X)



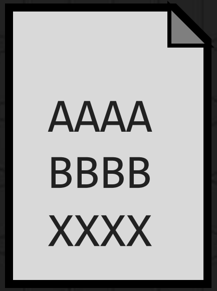


fileA.txt
(as in 1083…)
Shown in the working directory (HEAD)

Remote repository (configured as origin)



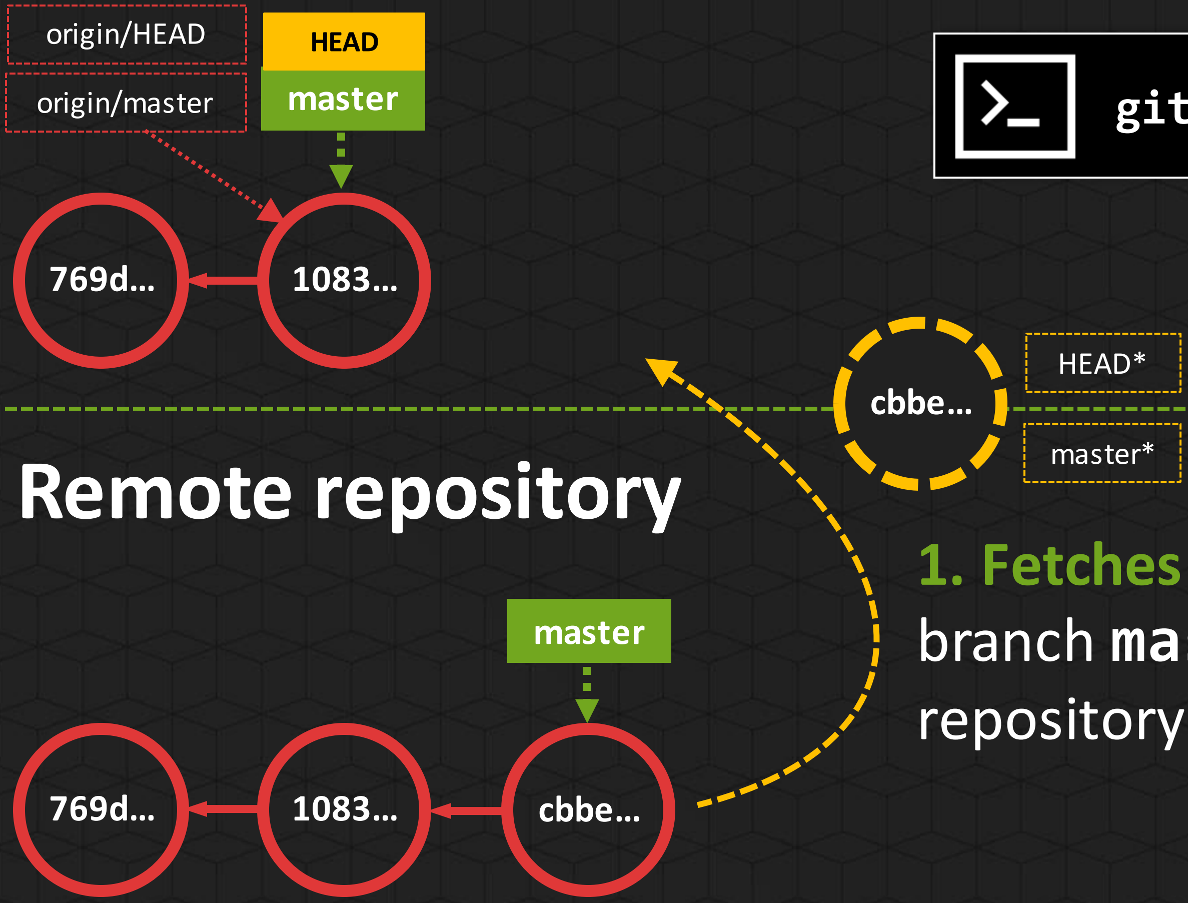
Another developer made a new commit and pushed it to the master branch on the remote repository. Developer X's local repository is not yet aware of this new commit.



fileA.txt
(as in cbbe…)

Git pull revisited

Local repository (of developer X)

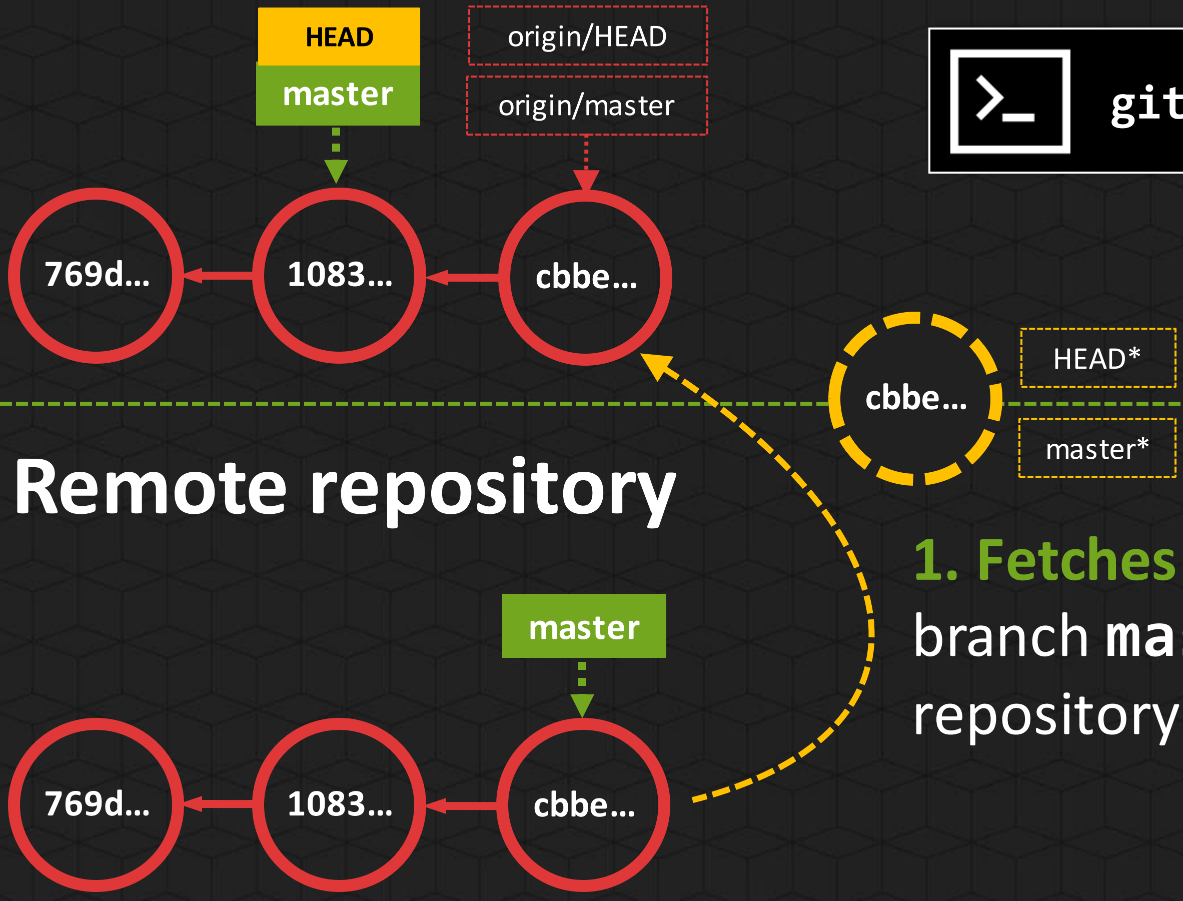


```
git pull origin master
```

1. Fetches the changes from the remote branch **`master`** on the remote repository called **`origin`**

Git pull revisited

Local repository (of developer X)



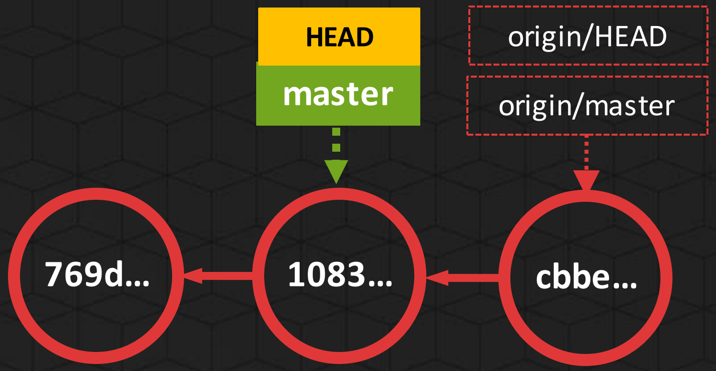
Remote repository

```
git pull origin master
```

1. **Fetches** the changes from the remote branch **`master`** on the remote repository called **`origin`**

Git pull revisited

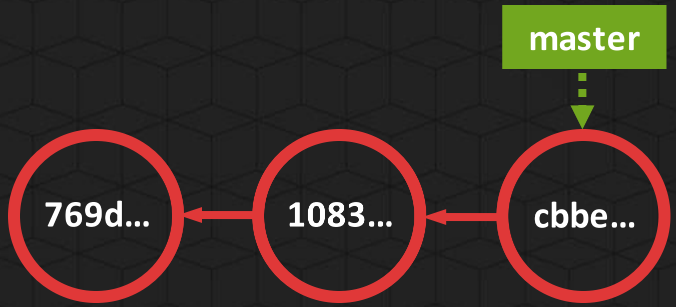
Local repository (of developer X)



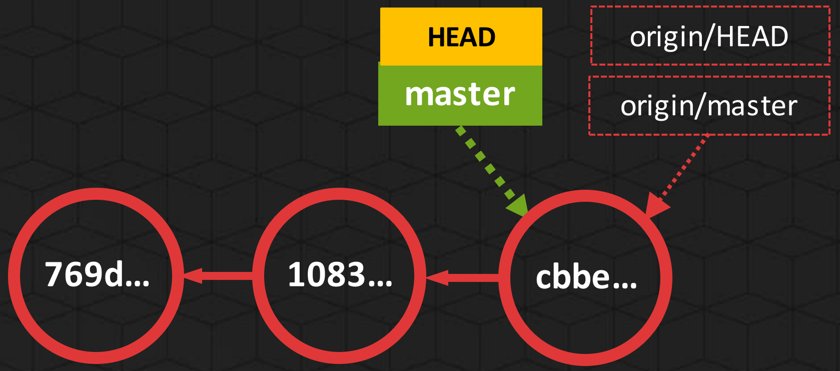
```
git pull origin master
```

2. **Merges** the changes into the current branch

Remote repository

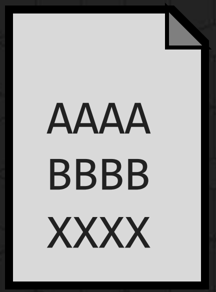


Local repository (of developer X)



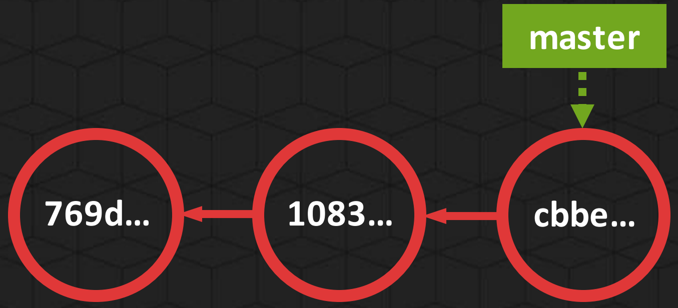
```
git pull origin master
```

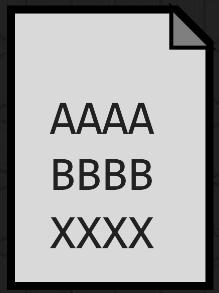
2. Merges the changes into the current branch



fileA.txt
(as in cbbe…)
Shown in the working directory (HEAD)

Remote repository





fileA.txt
(as in cbbe…)

Git pull revisited

`git pull` is nothing more than a shorthand for the combination of commands **`git fetch`** and **`git merge`**

Git pull revisited

The **Pull command**

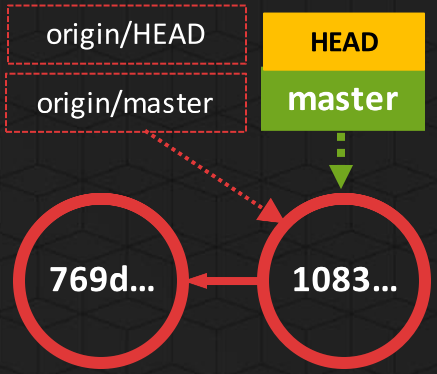
```
git pull [<remote> [<refspec>]]
```

Fetches, from the specified remote branch on the specified remote repository, the commits (+ the other newly created blob-objects + other refs) **and** automatically **merges** the changes into the local branch.

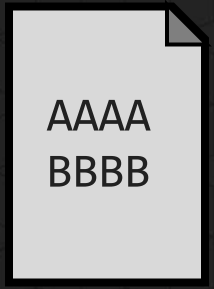
- Argument `<remote>` is the name (or URL) of the remote repository (to fetch from)
- Argument `<refspec>` is normally used to specify the remote branch

In it's default mode, **`git pull`**, is the combination of **`git fetch`** and **`git merge FETCH_HEAD`**

Local repository (of developer X)

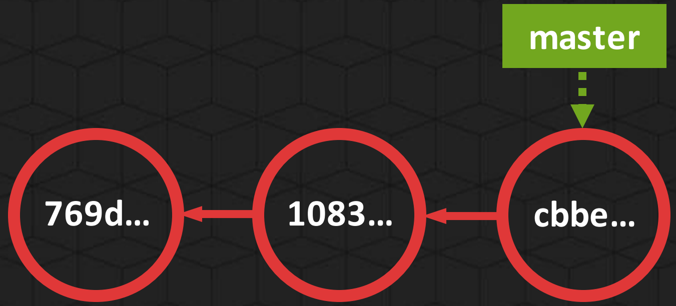


Let's retake our previous example. However, this time, we'll not use the pull command, but fetch followed by merge

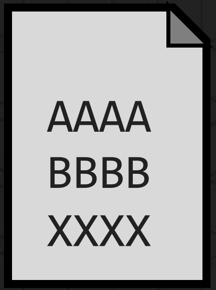


fileA.txt
(as in 1083…)
Shown in the working directory (HEAD)

Remote repository (configured as origin)



Another developer made a new commit and pushed it to the master branch on the remote repository. Developer X's local repository is not yet aware of this new commit.

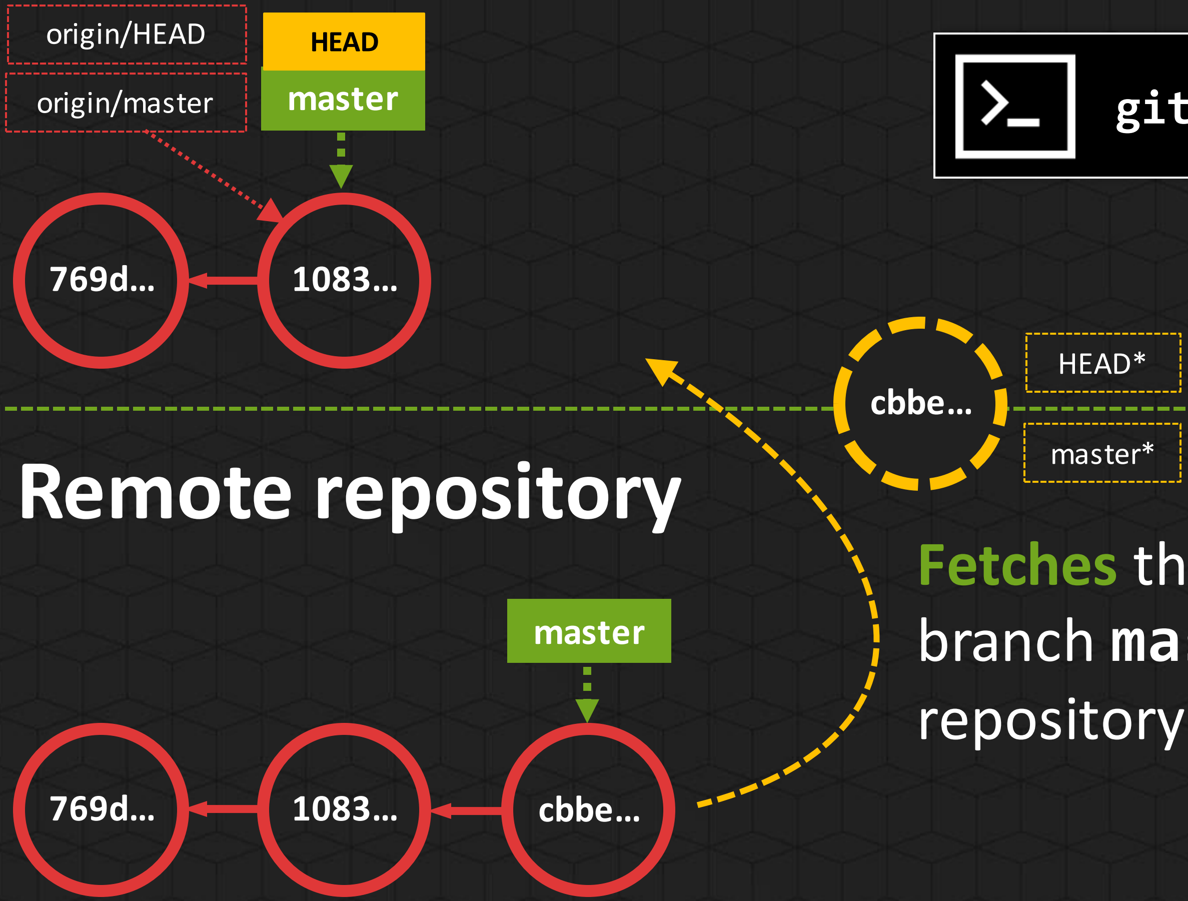


fileA.txt
(as in cbbe…)

Git pull revisited

Local repository (of developer X)

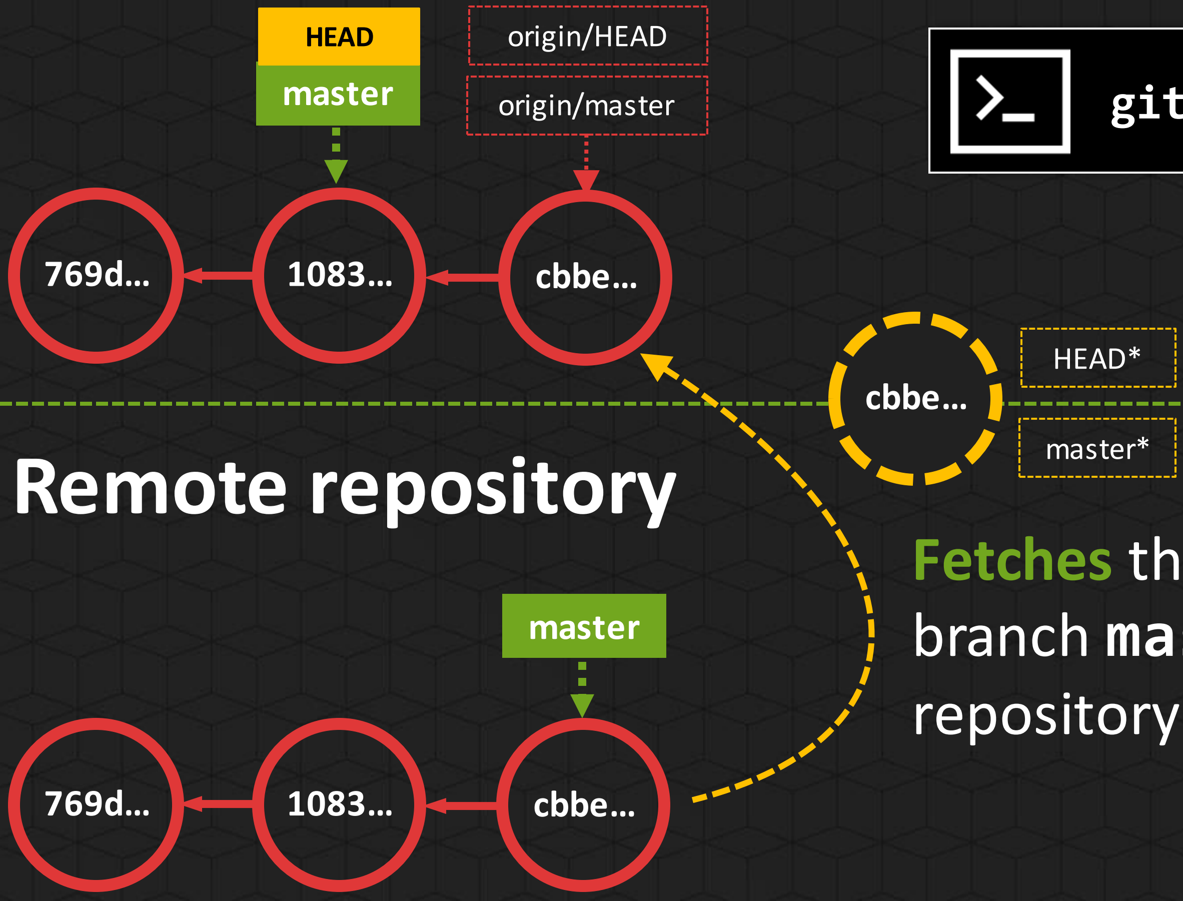
```
git fetch origin master
```



Remote repository

Fetches the changes from the remote branch **`master`** on the remote repository called **`origin`**

Local repository (of developer X)

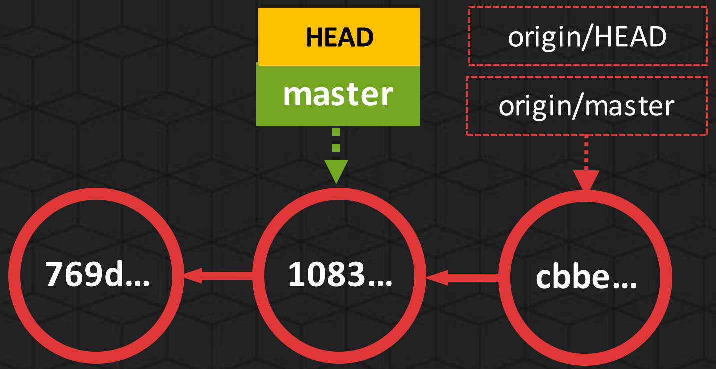


```
git fetch origin master
```

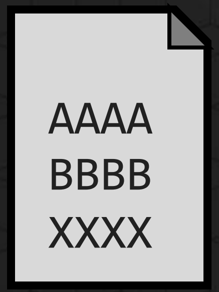
Fetches the changes from the remote branch `master` on the remote repository called `origin`

Git pull revisited

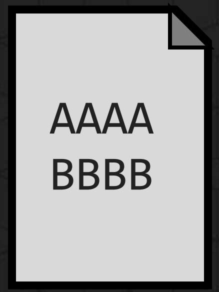
Local repository (of developer X)



The fetched changes are not yet incorporated into our checked-out (**`master`**) branch

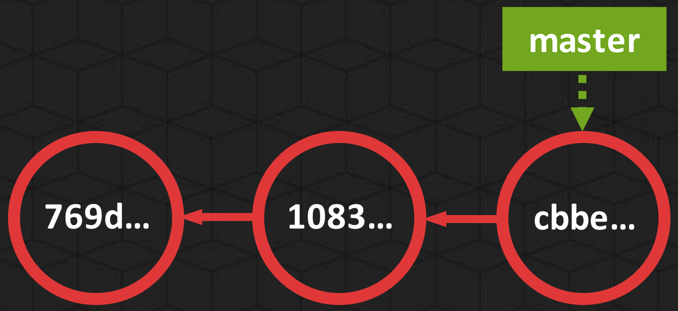


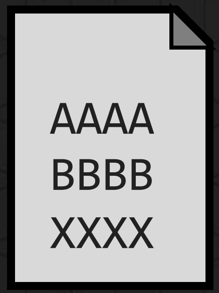
`fileA.txt (as in cbbe…)`



`fileA.txt (as in 1083…)`
`Shown in the working directory (HEAD)`

Remote repository

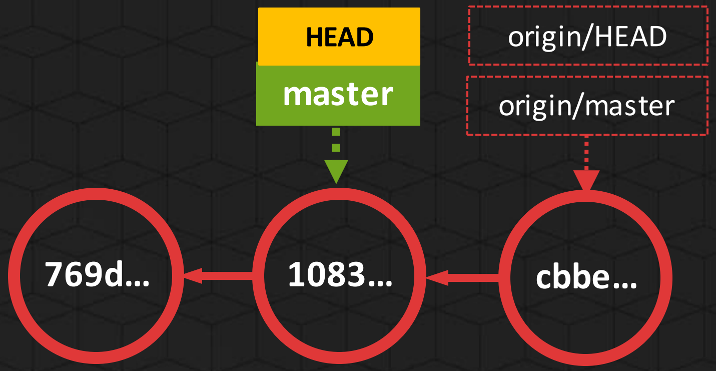




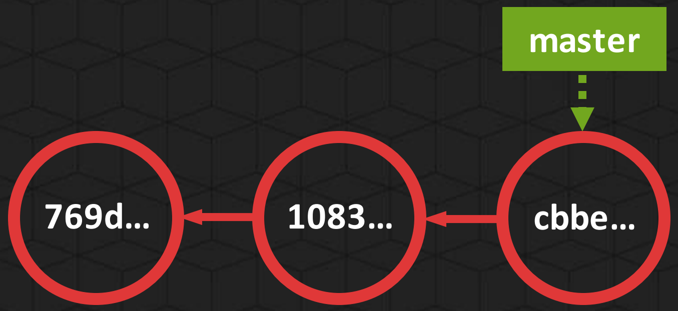
`fileA.txt (as in cbbe…)`

Git pull revisited

Local repository (of developer X)



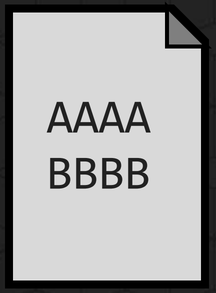
Remote repository



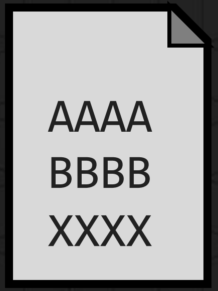
We can ask Git to show us all the **differences** between 2 branches (in this case: our local master branch with remote-tracking branch origin/master)

```
git diff origin/master
```

```
diff --git a/fileA.txt b/fileA.txt
index f0e5787..035f2bf 100644
--- a/fileA.txt
+++ b/fileA.txt
@@ -1,3 +1,2 @@
 AAAA
 BBBB
-XXXX
```

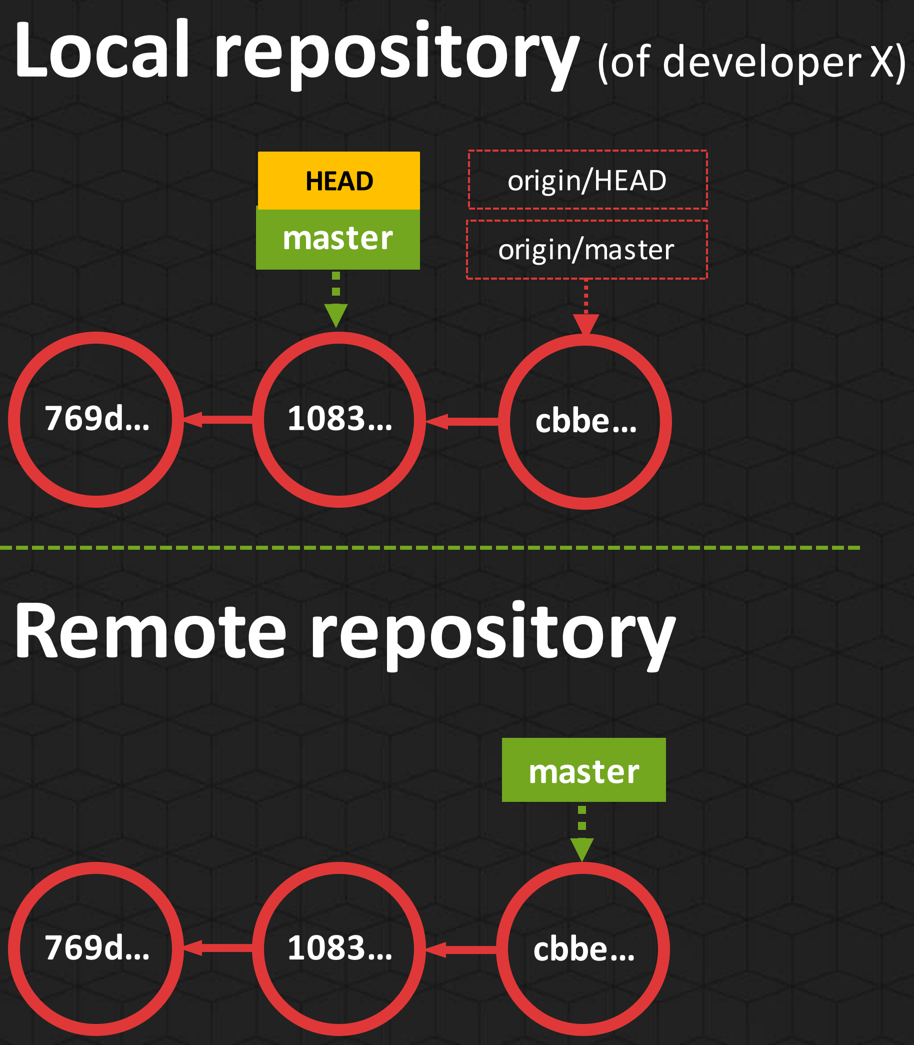


fileA.txt
(as in 1083…)
Shown in the working directory (HEAD)



fileA.txt
(as in cbbe…)

Git pull revisited

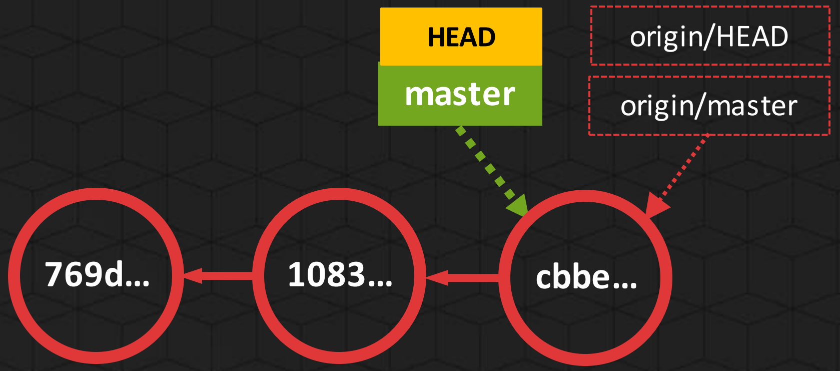


```
git merge origin/master
```

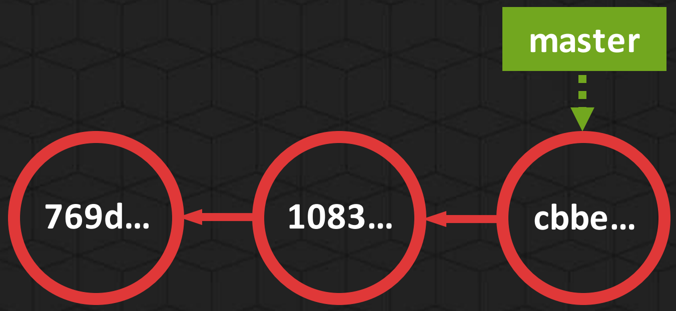
Merges the changes of the remote-tracking branch `origin/master` into the current branch (the branch to which HEAD is pointing)

Git pull revisited

Local repository (of developer X)

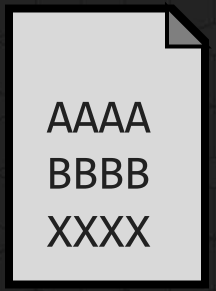


Remote repository

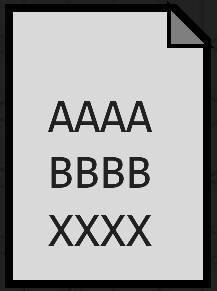


```
git merge origin/master
```

Merges the changes of the remote-tracking branch **`origin/master`** into the current branch (the branch to which HEAD is pointing)



fileA.txt
(as in cbbe…)
Shown in the working directory (HEAD)



fileA.txt
(as in cbbe…)

Git pull revisited

If you want to **fetch** (thus download without incorporating) **all** of the **branches, commits, objects,…** from a **remote,** use the **fetch command** as follows:

```
git fetch origin
```

Leaving out a remote, will (normally) default to the `origin` *remote. If you want to fetch everything from all configured remotes, use* `git fetch --all`

In essence, you **synchronize your local repository with the remote repository**

(without merging the downloaded remote changes)

Additional topics

Chapter 5

Overview

Additional topics

Git Workflows

"A **Git Workflow** is a **recipe** or recommendation for how to **use Git to accomplish work in a consistent and productive manner**. Git workflows encourage users to leverage Git effectively and consistently."

https://www.atlassian.com/git/tutorials/comparing-workflows

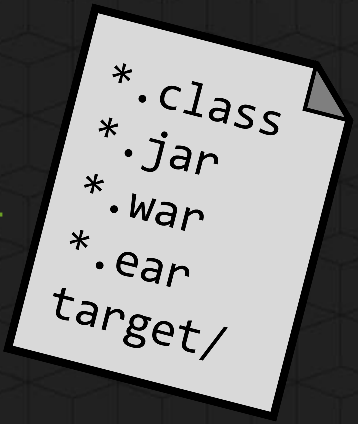
1. **Centralized workflow**
   https://www.atlassian.com/git/tutorials/comparing-workflows
2. **Feature-branch workflow**
   https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
3. **Gitflow workflow**
   https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
4. **Forking workflow**
   https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

Additional topics

.gitignore file

“A **.gitignore** file **specifies intentionally untracked files that Git should ignore.**”

https://git-scm.com/docs/gitignore



```
*.class
*.jar
*.war
*.ear
target/
```

Additional Topics

.gitignore file

Additional topics

Rebasing

Git rebase solves the same problem as **Git merge**, but **in another way**. The **primary reason for rebasing** is to maintain a **linear project history** (a three-way merge introduces parallel history, which can become less readable/usable over time)

- ✓ **https://git-scm.com/book/en/v2/Git-Branching-Rebasing**
- ✓ **https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase**
- ✓ **https://www.atlassian.com/git/tutorials/merging-vs-rebasing**

- ✓ https://blog.carbonfive.com/2017/08/28/always-squash-and-rebase-your-git-commits/

Rebasing

Additional topics

Cherry pick

Git cherry-pick allows to **select one** (or more) **commit**(s) **from one branch** and **apply it** (as a patch, thus a new (duplicate) commit) to **another branch**

- ✓ https://git-scm.com/docs/git-cherry-pick

Cherry pick

Additional topics

Revert / Reset / Checkout

“The **git reset**, **git checkout**, and **git revert** commands are some of the most useful tools in your Git toolbox. They all let you **undo some kind of change in your repository**, and the **first two commands** can be used **to manipulate either commits or individual files**. ”

https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting

There are some codelabs available for these commands in the 29-git-advanced-additional-topics submodule of our git module

Revert / Reset / Checkout

Additional topics

Git stash

“The **`git stash`** command takes your **uncommitted changes** (both staged and unstaged), **saves them away** (on a stack) **for later use**, and then reverts them from your working copy ”

https://www.atlassian.com/git/tutorials/saving-changes/git-stash

Git stash

Additional topics

Git tagging

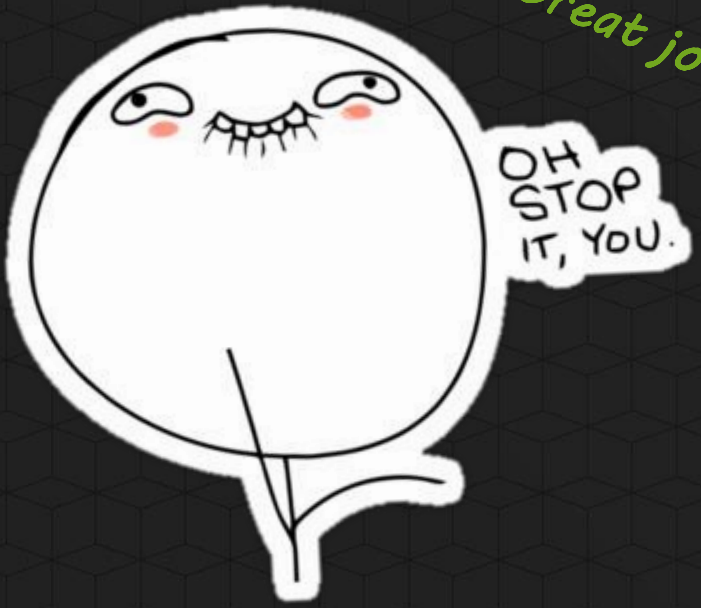
“**Git** has the ability to **tag specific points in history as being important**. Typically people use this functionality to mark release points (v1.0, and so on).”

https://git-scm.com/book/en/v2/Git-Basics-Tagging

Git tagging

1. √ Branches
2. √ Merging
3. √ Remote Branches
4. √ Pull & Push revisited
5. √ Additional topics

git commit -m "Great job done!"



Overview

Switchfully

Essentials| Version Control

Git Essentials: Advanced

www.switchfully.com