Introduction to Git

Course overview

1. Introduction

1. Introduction
2. About your teacher
3. Version control: What and why?

2. Baby steps: Working locally

1. Creating a repository
2. Working with commits
3. DAGs and SHA-1s
4. Local branching and merging

3. Getting distributed

1. Adding remotes
2. Pushing and pulling

4. Getting Awesome

1. The wonder of rebasing
2. Amending existing commits
3. Other useful stuff
4. Closing remark

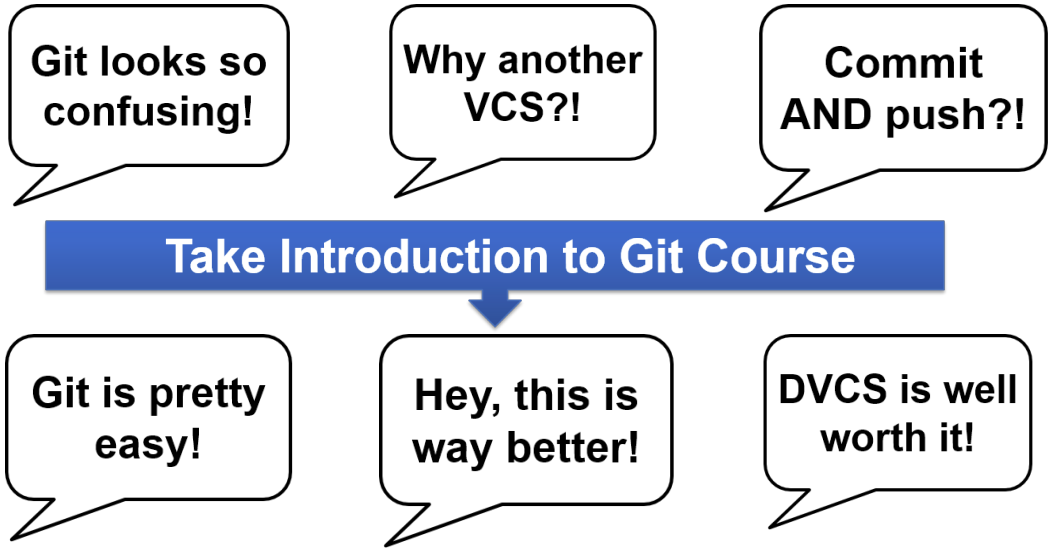
Introduction

Sections in this chapter:

1. Introduction
2. About your teacher
3. Version control: What and why?

1-1. Introduction

Course objectives



Illustrated course objectives

1-1-2

1-2. About your teacher

1-2-1

- Name: Mattias Andersen
- Computer geek
- Wife and two daughters
- Also a bass player
- Teacher/Consultant at Edument

At night time: One of two developers at Decemberborn currently developing the 8-bit game Cathedral

1-2-2

Techniques/languages: LUA-script, C++, Python and some basic pixel art:)

Worked in large projects with:

1-2-3

- React
- AngularJS
- Git, of course :)
- TDD

When I went to school the focus was on .NET and C#

Please interrupt me with your questions!

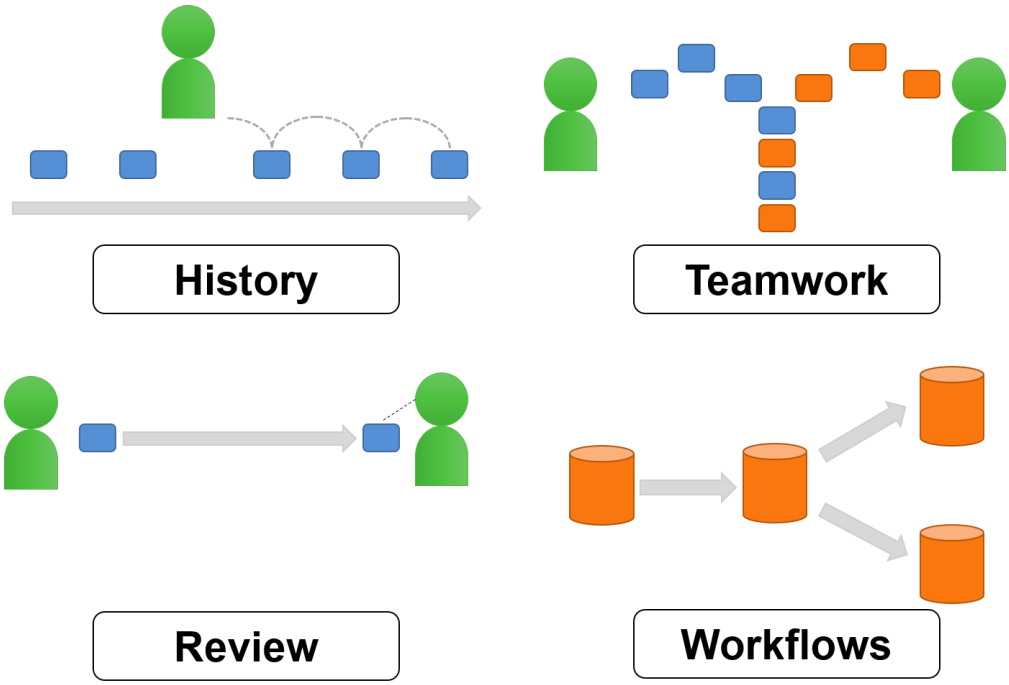
1-2-4

Even questions outside of Git are welcome!

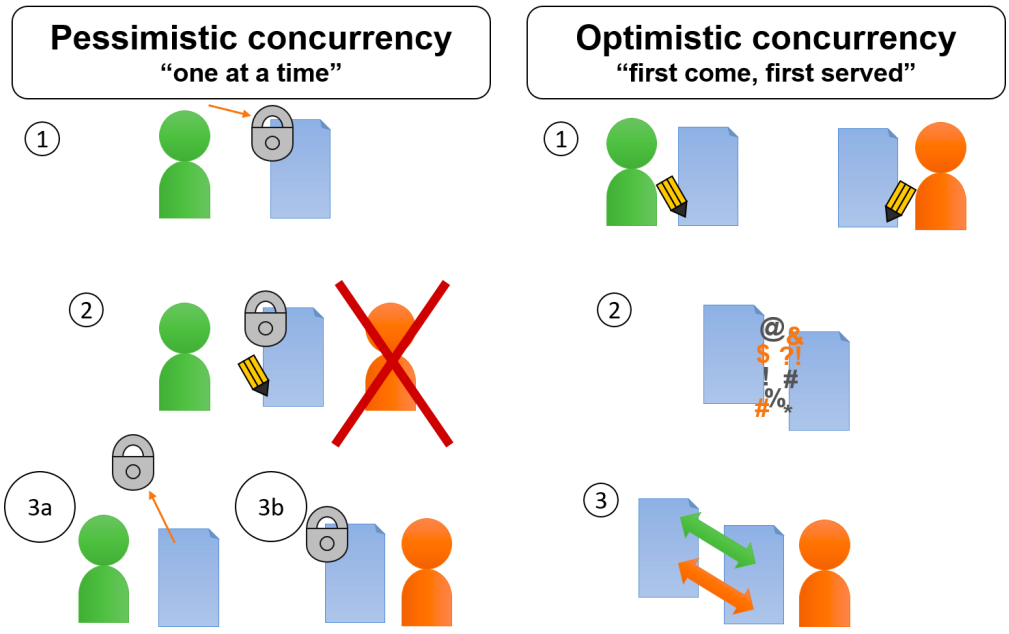
1-3. Version control: What and why?

A little background

Why we use version control

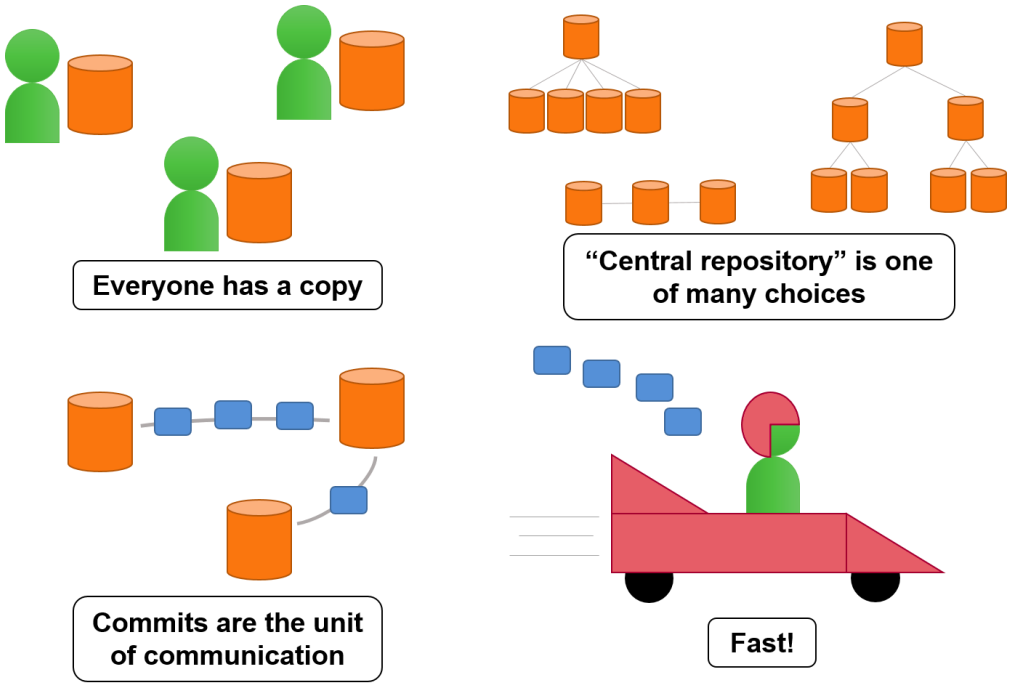


Two ways to handle conflicts



Advantages of distributed

1-3-3



Branches in the DVCS world

1-3-4

If you have never worked with a DVCS before...

...please now **forget everything you think you know** about branches! :)

Especially, *stop believing* all of the following:

1-3-5

- Branches are expensive
- Only create a branch if you really need it
- Merging is to be avoided because it's hard

Effective Git users work with branches every day!

Baby steps: Working locally

Sections in this chapter:

2-1. Creating a repository

For the first part of the course, we will be working with Git in an *entirely local setting* *2-1-1*

This means that you will be *creating a repository* and performing operations on *your own machine* *2-1-2*

Due to its distributed nature, Git allows you to do everything *locally*, from simple things like *commits and history viewing* through to more complex operations such as *branching and merging* *2-1-3*

Very different from centralized version control! *2-1-4*

Creating a Git repository takes about 15 seconds! *2-1-5*

1. Create a directory
   ```
   $ mkdir my-first-git
   ```
2. Change into that directory
   ```
   $ cd my-first-git
   ```
3. Tell Git to create a repository there
   ```
   $ git init
   ```

Q What just happened? *2-1-6*

A The *init* command creates a *.git directory* *2-1-7*

This is where the *entire version history of the repository* will be stored

The .git folder contains everything that matters. Want to give your full version history to a friend? Just zip up .git and send that; the "current version" can be recreated from what's in .git. *2-1-8*

Mini exercise *2-1-9*

Create you own git repo!

Make sure you can *find .git* inside your folder

Consult the previous slides if you get stuck

2-2. Working with commits

With Git, you *make a commit for each change* you wish to record in the system *2-2-1*

Some version control systems keep track of history per file. Git versions the whole tree.

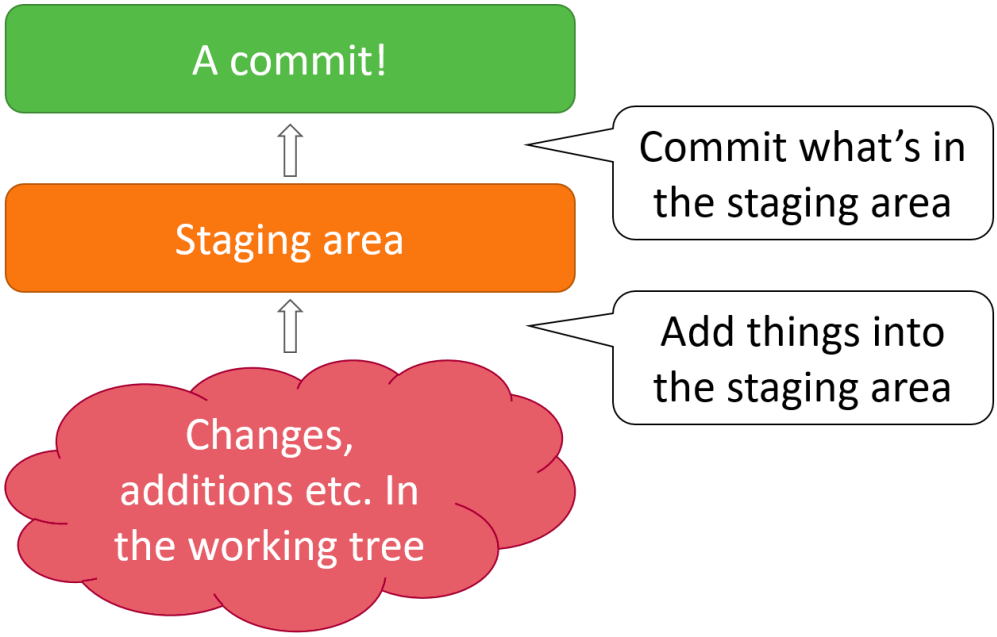
A commit consists of:

2-2-2

a. A snapshot of all the files
b. A commit message
c. An author
d. A date

In Git, committing is a two-step process (It is also possible to do it with a single command)

2-2-3



Adding new files

2-2-4

The `add` command places a file in the staging area Below is given that the README file exists

```
$ git add README
```

The `commit` command then takes the contents of the staging area, and creates a commit

```
$ git commit

[master (root-commit) beb9dfd] Add a README.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Committing changes

2-2-5

With existing files, add places the changes to that file into the staging area, so you can commit changes with two commands:

```
$ git add README
$ git commit
```

(A shorthand for these steps: `git commit README`)

Snapshots

2-2-6

Each time you add a file, you tell git "remember the file as it looks now, because that's how I want to commit it".

Snapshots

You can snapshot a file, and then keep working on it.

```
$ vim README //these changes will go into the commit...
$ git add README
$ vim README //...but not these changes
$ git commit
```

Mini exercise

2-2-8

a. Create a repo (or use the last one you created)
b. Create a file (either in the terminal or in your OS)
c. Add some content to the file
d. Make git aware of the file by adding it
e. Make a snapshot by running the commit command

Directories

2-2-9

If you use add with a file inside a directory, it will retain the directory structure

```
$ git add src/main.c
$ git commit
```

Directories

There is *only one .git* at the top level of the repository, not one inside every directory

Minor limitation: Git doesn't support putting empty directories under version control
If you really want an "empty" folder tracked in your repo you could put an empty .gitignore file inside of it and commit that instead

git status

Gives an overview of what is currently staged, changed in tracked files and untracked

```
# On branch master
# Changes not staged for commit:
#
#       modified:   src/main.c
#
# Untracked files:
#
#       src/util.c
#       src/util.h
```


Modified files are already known to git

Untracked files are not

git status

Tip: If you get fed up of typing status, alias it to st

```
$ git config --global alias.st status
```

Let's add one file and look at the status again... *2-2-13*

```
$ git add src/util.h
$ git status
```

```
# On branch master
# Changes to be committed:
#
#       new file:   src/util.h // Now in the staging area
#
# Changes not staged for commit:
#
#       modified:   src/main.c //Not in staging, but a commit
#                              //of src would include it
#
# Untracked files:
#
#       src/util.c
```


.gitignore

2-2-14

One annoyance is that after building our code, the generated files show up in the status output

```
# On branch master
# Untracked files:
#
#       main.exe
#       main.obj
#       util.obj
```

The solution is to *add a .gitignore* file *2-2-15*

Content for .gitignore

```
*.exe
*.obj
```

Then commit the file

```
$ git add .gitignore
$ git commit
```

Mini exercise

2-2-16

a. Create a file type that you'd like to ignore
b. `git status` to see the new untracked file
c. Create a .gitignore
d. Make sure to ignore the first file you created
e. Use `git status` to confirm the file is ignored
f. Add and commit the .gitignore file

git diff

2-2-17

The status command gives you an overview of your changes; diff gives the details

```
$ git diff
```



```
diff --git a/src/util.c b/src/util.c
index 9fe5927..e826a45 100644
--- a/src/util.c
+++ b/src/util.c
@@ -1,7 +1,7 @@
 #include <stdio.h>

-void print_ten_times(char *msg) {
+void print_n_times(char *msg, int n) {
     int i;
-    for (i = 0; i < 10; i++)
+    for (i = 0; i < n; i++)
         printf(msg);
}
```

git diff

2-2-18

```
-    for (i = 0; i < 10; i++)
+    for (i = 0; i < n; i++)
```

Red and the "-", indicate removed lines

Green and the "+" indicate added lines

By default diff only shows unstaged changes; use the staged option to see the staged changes

2-2-19

```
$ git add src/main.c
$ git diff --staged
```

```
diff --git a/src/main.c b/src/main.c
index 5f8a873..81f2948 100644
--- a/src/main.c
+++ b/src/main.c
@@ -1,5 +1,5 @@
 #include "util.h"

 int main() {
-    print_ten_times("Hello, world!\n");
+    print_n_times("Hello, world!\n", 10);
 }
```


The --stat option will just give you statistics on the changes that have been made to each file

2-2-20

```
$ git diff --stat
```

```
 src/main.c |    2 +-
 src/util.c |    4 ++--
 src/util.h |    2 +-
3 files changed, 4 insertions(+), 4 deletions(-)
```

Has many more options than this; see:

```
$ git help diff
```

Mini exercise

2-2-21

`diff`

a. Create a file. Add and commit it
b. Make a change to the file
c. Try `git diff` to see the changes
d. Add the changes to the staging area
e. Find out why `git diff` now gives empty output

(Use the course material)

git log

2-2-22

Shows a log of all the commits, most recent commit first

```
commit e20962ebed7b0288922320f217a6a3ab9371727c
Author: jnthn
Date:   Wed Apr 18 18:09:02 2012 +0200

  Add a .gitignore.

commit eae16e7a7f34d1208ca8267c2fabbbc1eb8e3640
Author: jnthn
Date:   Wed Apr 18 17:56:55 2012 +0200

  Factor printing out to a utility file.

...
```


git log

2-2-23

Has more options than you can possibly imagine; one fairly useful one is:

```
$ git log --oneline
```

```
e20962e Add a .gitignore.
eae16e7 Factor printing out to a utility file.
887f06c Start le coding!
869cec3 Update README.
8356287 Add a README.
```

To learn about (literally) dozens more, see:

```
$ git help log
```

git show

2-2-24

Shows you the full details of a commit

```
$ git show 869cec3
```

```
commit 869cec3f4e200ded3e9b7d27823cfd4da8cd086d
Author: jnthn
Date:   Wed Apr 18 17:34:10 2012 +0200

  Update README.

diff --git a/README b/README
index 7062ca7..79a326b 100644
--- a/README
+++ b/README
@@ -1 +1 @@
-This project will be awesome!
+This project will be awesome! REALLY awesome!
```


Deleting files

2-2-25

If you use the `rm` command, then the deletion of the file is immediately placed into the staging area

```
$ git rm README
$ git commit
```

Deleting files

2-2-26

Alternatively, you can just delete the file in your OS or IDE; a commit of the directory that contains it will then do the right thing

```
$ del README
$ git commit .
```

Moving files

2-2-27

Once again, two options; the `mv` command which immediately stages the rename:

```
$ git mv src\main.c src\hi.c
$ git commit
```

Moving files

2-2-28

Or move it with your OS or IDE, and git add the new file; it will figure out it's the same file!

```
$ move src\main.c src\hi.c
$ git add src\hi.c
$ git commit src
```

The staging area

2-2-29

To repeat, after the first `add` to make git care about a file, you can always `commit` things directly by naming a file or a directory.

So why even have a staging area?

For commits that *only affect a single* file, the staging area isn't useful. Indeed, this is where committing directly is common.

For *larger changes*, the staging area allows us to *structure the commit*, piece by piece.

EXERCISE 1

2-2-30

This exercise is aimed at getting you comfortable with the commands that we have covered so far

It includes...

a. Creating a new repository
b. Adding new files
c. Committing changes
d. The status command
e. Looking at diffs and logs
f. Moving and deleting files

2-3. DAGs and SHA-1s

A peek inside Git

SHA-1 Hashes

2-3-1

Every commit is identified by a SHA-1 hash; we came across these when using `log` and `show`

```
commit e20962ebed7b0288922320f217a6a3ab9371727c <- The SHA-1
Author: jnthn
Date:   Wed Apr 18 18:09:02 2012 +0200

    Add a .gitignore.
```

The hash is *derived from the content of the commit* along with the commit that came before it

2-3-2

This means that the SHA-1 is not only unique locally, *but unique over all copies of a repository*!

2-3-3

Writing SHA-1 Hashes

2-3-4

We have also seen SHA-1 hashes show up in an *abbreviated form*

```
e20962e Add a .gitignore.
eae16e7 Factor printing out to a utility file.
887f06c Start le coding!
869cec3 Update README.
8356287 Add a README.
```

Usually, the first six characters are enough to *uniquely identify a commit* in a repository

2-3-5

You can provide as few or as many characters as you wish, provided they identify a *single commit*

Q Why not a revision number?

2-3-6

A Using numbers would imply there is one unique ordering of commits *2-3-7*

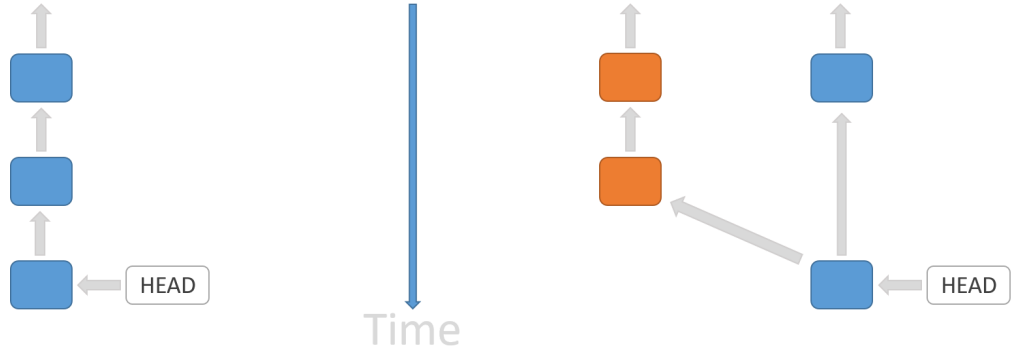
This is *fine if you have a centralized version control system*, since the server can decide who gets to commit first if there is competition

This doesn't make any sense in a distributed world, where different local copies will have commits that have not been shared yet *2-3-8*

SHA-1 is always content-unique

The DAG (Directed Acyclic Graph) *2-3-9*

The version history is represented by a DAG of commits. *Each commit points to the commit that came before it* (or "commits" when there is a merge)



The simple, linear case *With a merge commit*

And that's Git! *2-3-10*

Just about everything in Git boils down to...

a Commits that are identified by a SHA-1

b Manipulating the DAG of commits

The rest of this course will show you a whole array of different commands and patterns *2-3-11*

However, everything we will cover is really just about commits and placing them in graphs

The underlying model of Git is simple :)

Creating and listing branches

2-4-1

Creating a branch in Git is easy, and also an extremely cheap operation

```
$ git branch calc-fibs
```

You can now get a list of branches that exist

```
$ git branch

  calc-fibs
* master
```


Notice the current branch is still master!

Switching to a branch

2-4-2

The `checkout` command is used to switch from one branch to another

```
$ git checkout calc-fibs
```

We can use the branch command again, to verify that we have indeed switched branch

```
$ git branch

* calc-fibs
  master
```

Committing to the branch

We make a change, and check the status

```
$ git status
```

```
# On branch calc-fibs
# Changes not staged for commit:
#
#       modified:   src/util.c
#
```

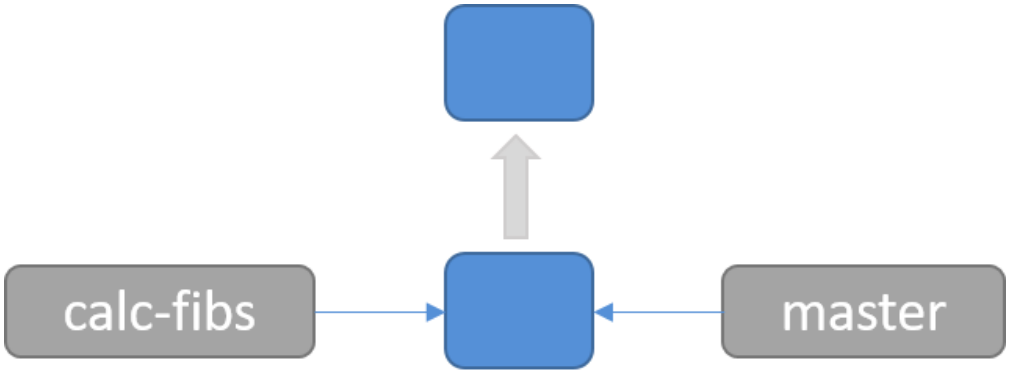
It notes we are on the calc-fibs branch. We now go ahead and commit the usual way...

```
$ git add src/util.c
$ git commit
```


So, what is a branch?

A branch is really just a reference to a commit

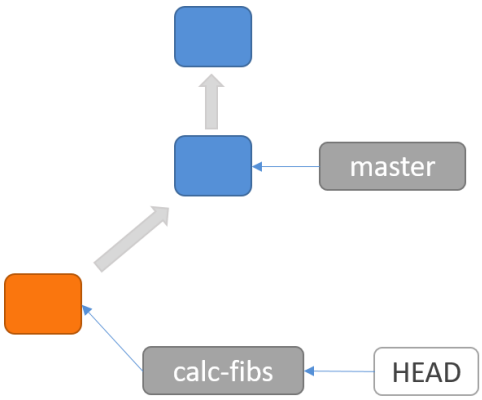
Given every commit knows its predecessor, then knowing the latest commit in a branch is enough to know the full history of the branch



HEAD

HEAD always refers to the current commit

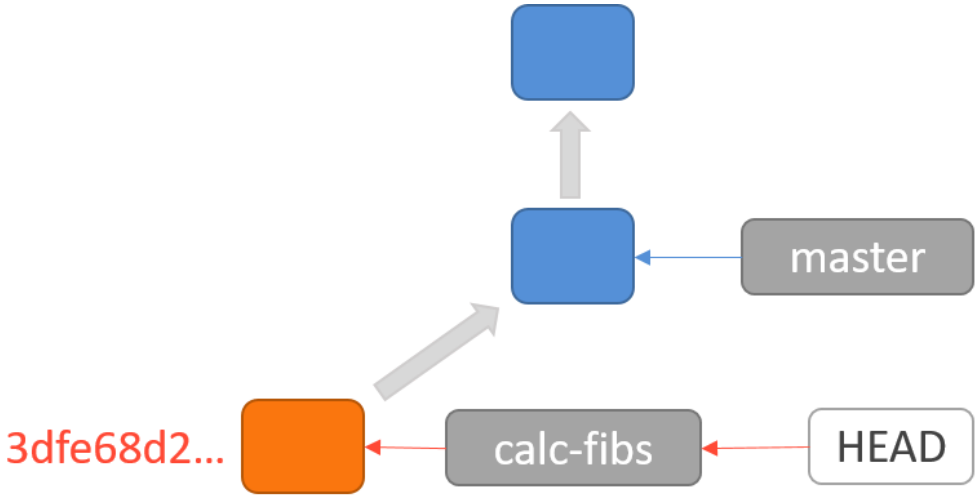
It does this by referring to the current branch, which contains the SHA-1 of the latest commit



What a commit does (1)

First, HEAD is used to find the current branch (provided there is one; there is also a detached HEAD mode, which we'll pass over for now)

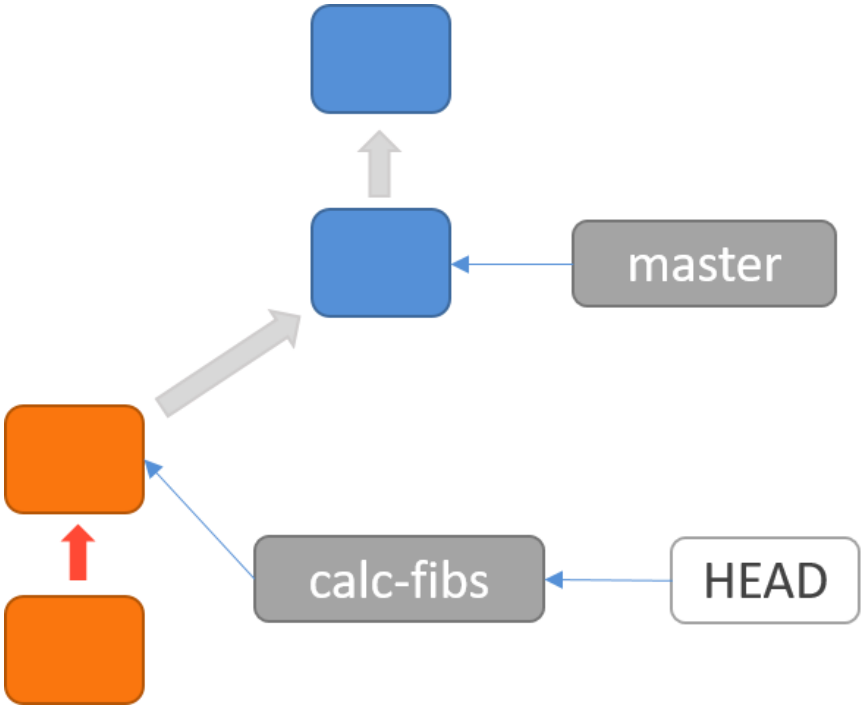
This in turn is used to get the SHA-1 of the current commit



What a commit does (2)

A new commit is created, with its parent set to the current commit, which we just located

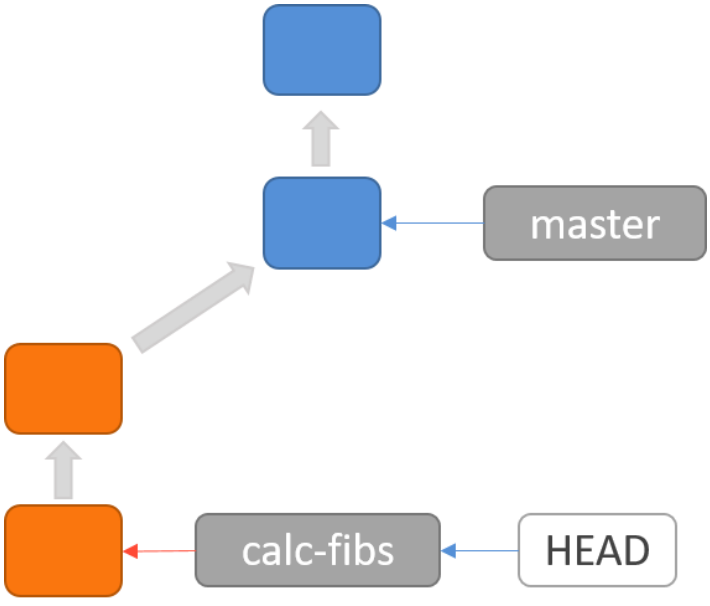
It is SHA-1'd and stored by Git



What a commit does (3)

Finally, the branch alias is updated to point to the new commit, so it becomes the branch's latest

Note that HEAD itself is not updated; it need not be as it points to the branch



Branch switching is "in place"

2-4-9

In some version control systems, different branches are stored in different directories, so moving between them is a directory change

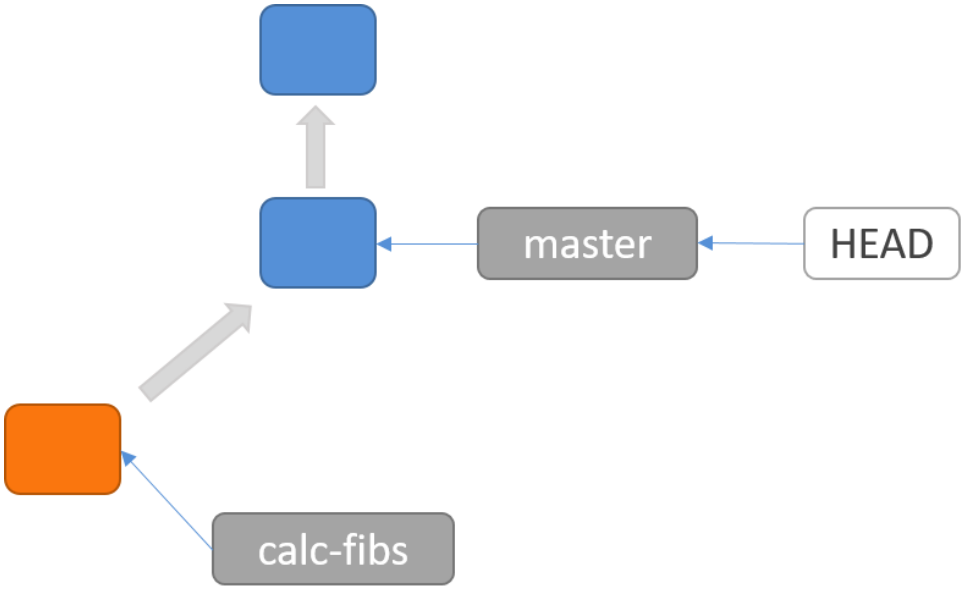
With Git, *using `checkout` to move to a different branch changes the contents of the repository directory in place*, altering the files to represent the state they are in in the branch you switch to

Generally, this is a win; you can create a branch and start working in it without having to do a fresh build of your system!

Fast-forward "merges" (1)

2-4-10

Consider the DAG at this point in time



We have made no commits to master since we started the branch

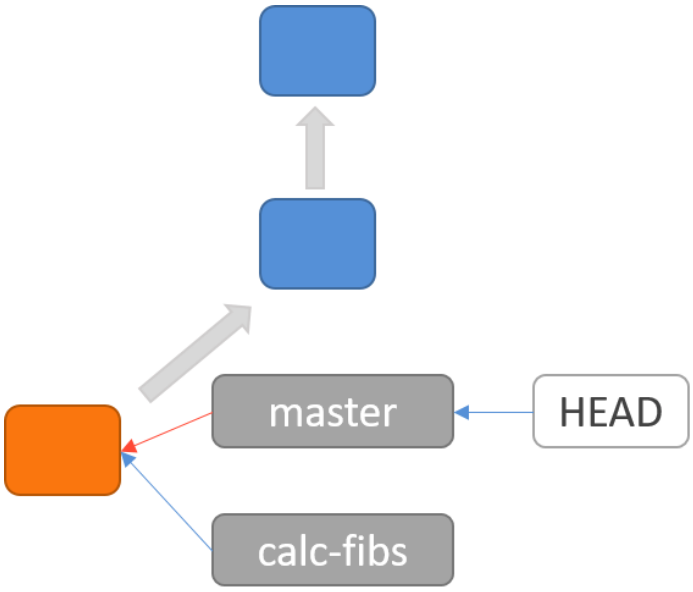
This means we can do a "fast-forward" merge

Fast-forward "merges" (2)

2-4-11

In a fast-forward merge, we don't actually have to do any merging in the traditional sense

Instead, we simply update the master branch's current commit to alias the latest commit in the fib-calc branch - and we're done!



Fast-forward "merges" (3)

2-4-12

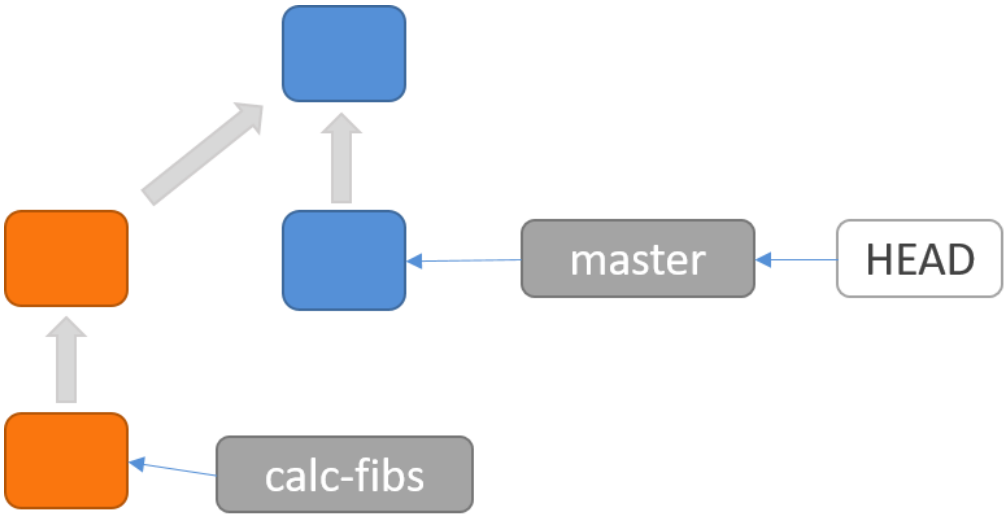
Fast-forward merges are cheap and completely painless - there's never going to be a conflict
Doing development work in a local branch can be useful if you need to urgently task-switch back to master for doing a bug fix, or you discover a need to work on some other prerequisite first

TIP: If you're unsure whether you should do some work in a branch - Just Do It. If you turn out to need it, it's right there for you. If you don't need it, you get a painless, instant fast-forward merge.

Merge commits (1)

2-4-13

In many situations, there will have been commits in both branches; here we can see that master has a commit and fib-calc has a couple



A fast-forward is simply not possible here

Merge commits (2)

2-4-14

Before the merge, we may want to inspect how the two branches differ. There are two questions to answer here. First, what happened on the calc-fibs branch?

```
$ git log master..calc-fibs
```

Second, what happened on the master branch?

```
$ git log calc-fibs..master
```

Now we know what commits we're merging together

Merge commits (3)

The output of a merge will look a little different:

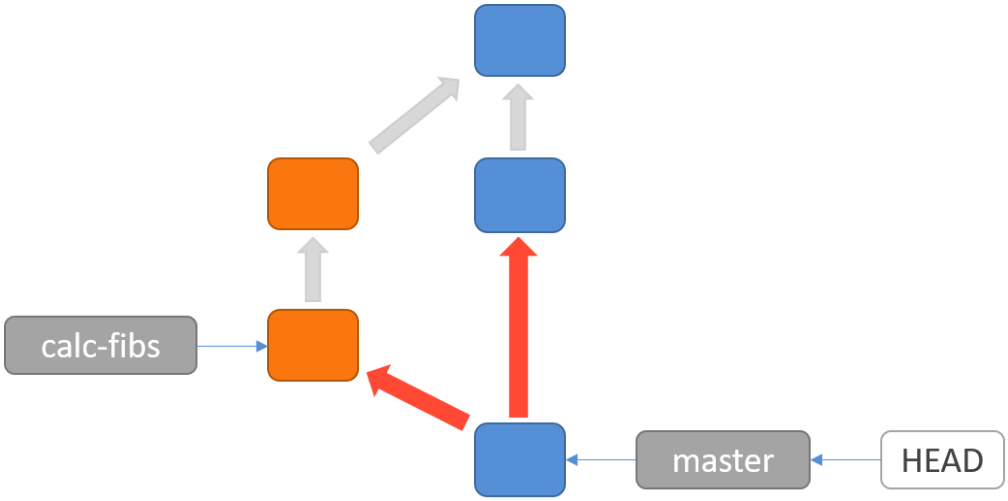
```
Auto-merging src/util.c
Merge made by recursive.
 src/hi.c   |    3 +++
 src/util.c |    7 +++++++
 src/util.h |    1 +
 3 files changed, 11 insertions(+), 0 deletions(-)
```

It doesn't mention fast-forward, but rather that there was a merge (by the recursive algorithm)

Additionally, we are informed that both branches changed src/util.c, but Git handled that by itself

Merge commits (4)

This kind of merge also produces a merge commit, which has a couple of parents

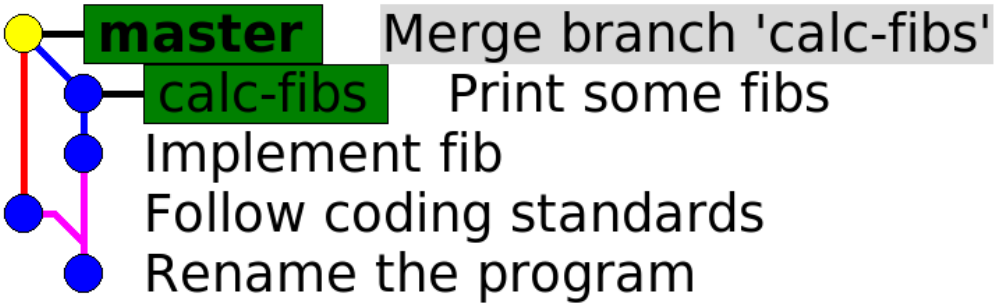


gitk

As well as the command line tool, Git ships with a couple of small, but useful GUI applications

One of them is *gitk*, which provides a nice way to explore the commit history

It also gives a visual representation of the DAG



Coping with conflicts (1)

There are some cases where a merge commit is needed, but Git can't completely work it out

For example, imagine calc-fibs has this commit:

```
 int main() {
     print_n_times("Hello, world!\n", 10);
+    printf("Fib 5 = %d\n", fib(5));
+    printf("Fib 10 = %d\n", fib(10));
 }
```

And master has this one:

```
 int main() {
     print_n_times("Hello, world!\n", 10);
+    return 0;
 }
```

Coping with conflicts (2)

2-4-19

When we issue the merge command, we get some slightly scarier output:

```
Auto-merging src/hi.c
CONFLICT (content): Merge conflict in src/hi.c
Auto-merging src/util.c
Automatic merge failed; fix conflicts and then commit the result.
```

This demands...manual intervention!

Coping with conflicts (3)

2-4-20

We open the conflicting file(s) in our editor; the conflicts are indicated as follows:


```
int main() {
    print_n_times("Hello, world!\n", 10);
<<<<<<< HEAD
    return 0;
=======
    printf("Fib 5 = %d\n", fib(5));
    printf("Fib 10 = %d\n", fib(10));
>>>>>>> calc-fibs
}
```

The markers tell us which part came from which branch; we just edit the file to how it should be

Coping with conflicts (4)

2-4-21

After editing, the file looks like this:

```
int main() {
    print_n_times("Hello, world!\n", 10);
    printf("Fib 5 = %d\n", fib(5));
    printf("Fib 10 = %d\n", fib(10));
    return 0;
}
```

If we try and commit now, however, it won't work:

```
$ git commit

U       src/hi.c
fatal: 'commit' is not possible because you have unmerged files.
```

Coping with conflicts (5)

2-4-22

We have to inform Git that we have *resolved the conflict* to our satisfaction; this is done using the `add` command:

```
$ git add src/hi.c
```

At this point, the `commit` command can be issued, and this time it will be successful.

```
$ git commit
```

EXERCISE 2

2-4-23

This exercise will give you a chance to practice *working with branches locally*

Most importantly, it will give you a chance to get a feel for the *different types of merge*...

a. Fast-forwards
b. Merges that are fully automatic
c. Merges that need manual intervention

Do ask questions if you're unsure; *branching and merging are key Git skills* to master!

Getting distributed

Sections in this chapter:

1. Adding remotes
2. Pushing and pulling

3-1. Adding remotes

Remotes

3-1-1

Earlier on, it was mentioned that you can copy a Git repository, complete with history, by copying the contents of the .git directory

A remote is another copy of the current repository

It may have some commits we don't have
It may lack some commits we have

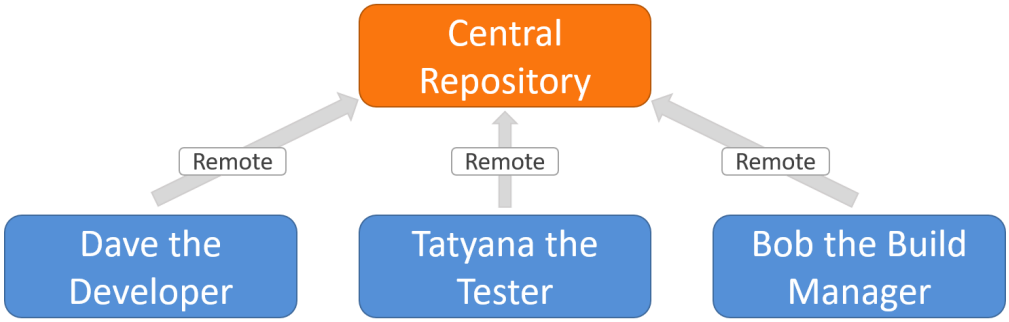
3-1-2

It should always be possible to trace back from the current latest commits locally and in the remote to a common ancestor

A central repository

3-1-3

The most common use of remotes is to set up a central repository



It's the central repository because everyone agrees it is. To Git, it's just another copy.

Hosted Git

3-1-4

There are a range of hosted Git services out there, which can host your central repository for you

GitHub is the largest today, offering free hosting for public repositories (used by thousands of open source projects) and private hosting for individuals and organizations

Bitbucket and *GitLab* are known alternatives; some projects have chosen them over GitHub

3-1-5

You can also host it yourself, with a bit of setup

This course: GitHub

3-1-6

We'll use repositories on GitHub for the purpose of practicing with remotes

All of the commands and techniques we learn are completely transferable to working with any other remote, whether hosted internally or elsewhere. Having a GitHub account also opens the door to contributing to thousands of open source projects; it may well come in useful to you far beyond this course. :)

Adding a remote

3-1-7

After creating a repository on GitHub, we need to tell our local Git repository about it by adding it as a remote

```
$ git remote add origin https://github.com/<user>/<repo>.git
```

Since you can have multiple remotes, you have to specify a name as well as the address

The convention used by almost all Git users is to call the central repository remote origin

Mini mission/exercise

3-1-8

a. Goto Github
b. Create an account (It's free)
c. Start a new repository
d. Add the github repo as a remote in one of your repos

3-2. Pushing and pulling

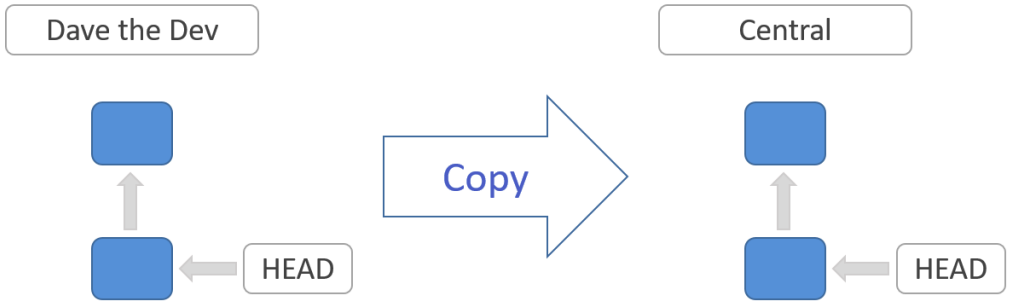
The first push

3-2-1

Pushing is taking commits we have locally and copying them to a remote

If we have a new, empty, central repository then our first push should use the -u flag

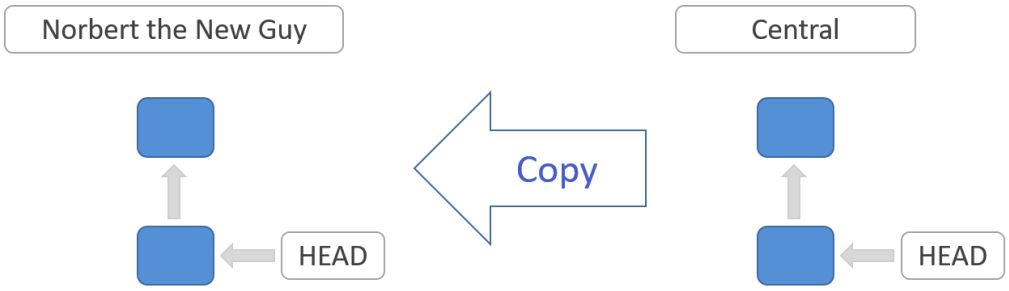
```
$ git push -u origin master
```



Cloning

The `clone` command is used when you want to *get a local copy of a remote repository*; it also sets up origin to point to the remote for you

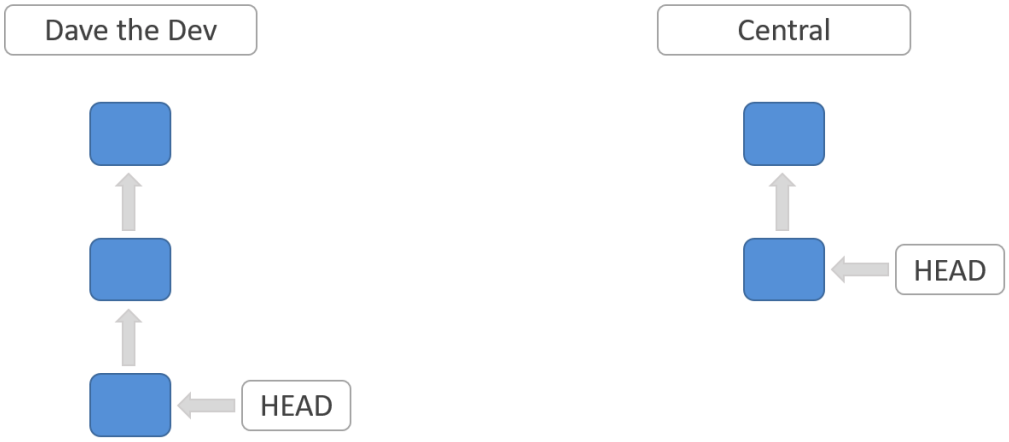
```
$ git clone https://github.com/<user>/<repo>.git
```



You use `clone` once to get an initial copy of the remote; later you'll be using `pull` to synchronize.

Sharing changes (1)

After you have done some work locally, you will have one or more commits that the central repository does not have



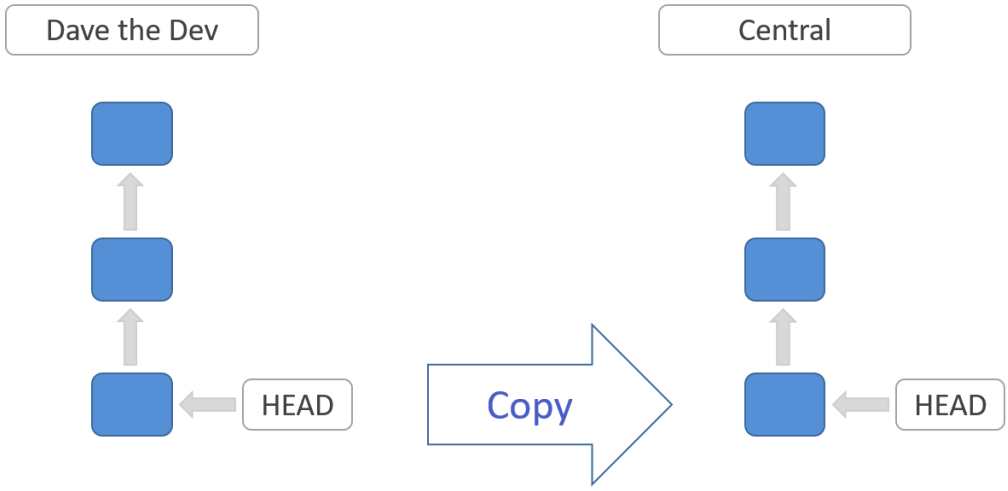
In this case, we need to copy the one extra, new commit over to the central server

Sharing changes (2)

3-2-4

The push command is used to do this:

```
$ git push origin master
```

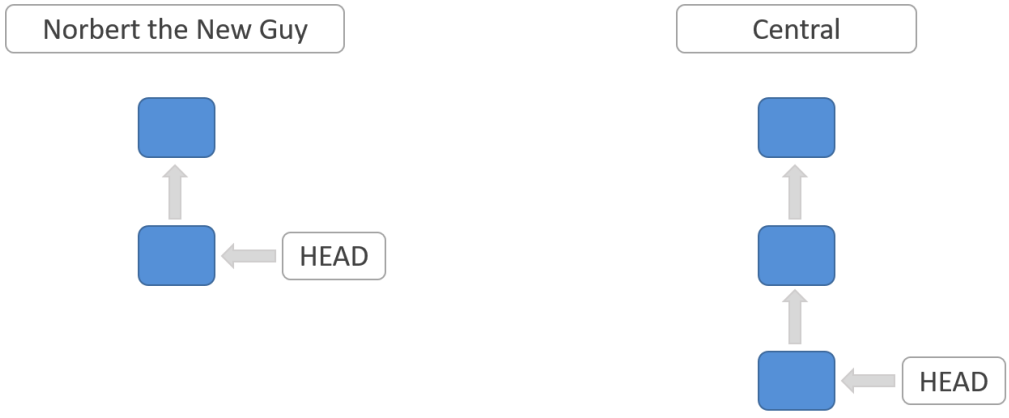


You don't need to tell Git which commit(s) it needs to copy. It always works that out for you. :)

Getting the latest changes (1)

3-2-5

At this point, the central repository has a commit that Norbert the New Guy doesn't have in his local copy of the repository



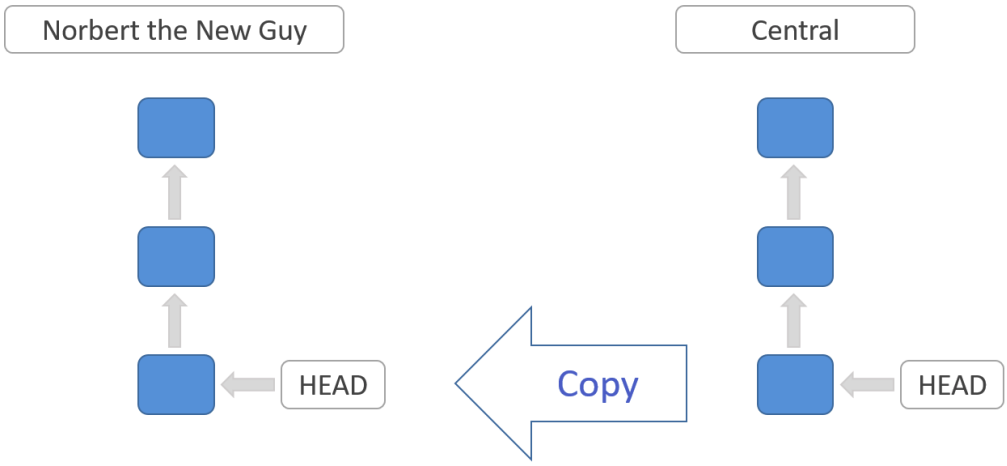
We need to copy that new commit from the central repository into the local copy

Getting the latest changes (2)

3-2-6

The `pull` command is used to fetch the latest changes and merge them into our local copy

```
$ git pull
```



Getting the latest changes (3)

3-2-7

`pull`
really just calls two other commands for us:

`fetch`
This command fetches all of the changes from a remote that we don't know about locally

`merge`
We already saw this one! :)

The exact same mechanism we used to merge local branches is also used to merge commits from a remote. Neat, huh? :)

When pulling gets tricky

3-2-8

Since a pull involves a merge, all of the cases of merging we saw earlier can come up again

Fast-forward
If you have no local commits, you always get a straightforward fast-forward merge

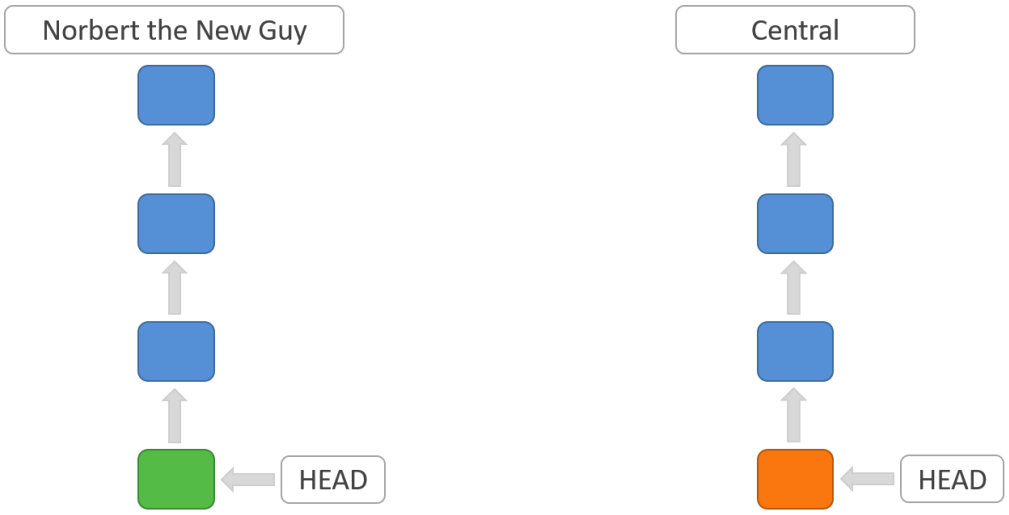
Merge Commit *3-2-9*

When you have local commits and also bring in remote ones, the DAG looks just as it does in a branch situation. A merge commit is needed. Sometimes, manual intervention may be needed.

When pushing gets tricky (1)

3-2-10

Sometimes, both you and the remote repository may have extra commits

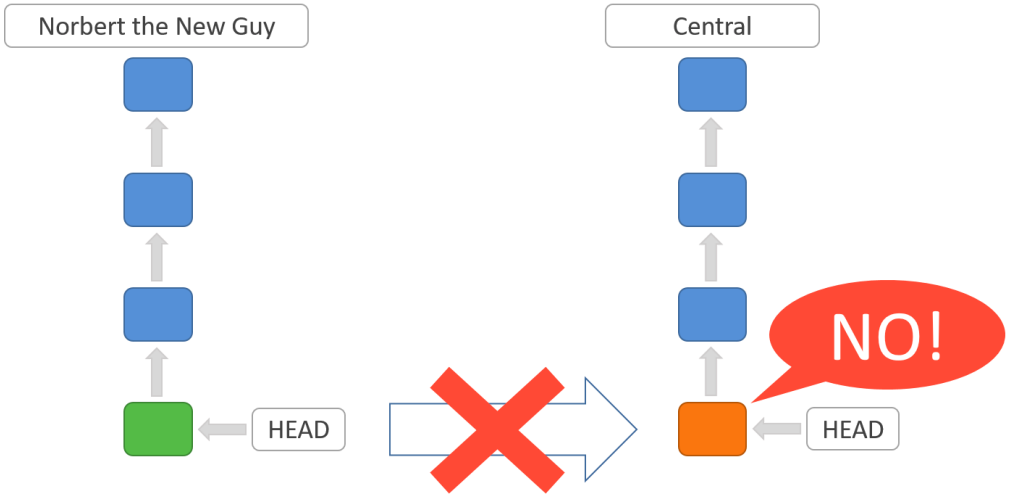


When pushing gets tricky (2)

3-2-11

If Norbert tries to push, it will fail

```
$ git push origin master
```



When pushing gets tricky (3)

3-2-12

The solution is to pull first

```
$ git pull
```

This will result in a merge commit being created locally; we then push it along with our commit

```
$ git push origin master
```

The underlying principle here is that all merging takes place locally

Exercise 3

3-2-13

In this exercise, you will work in small groups of two or three, practicing working with a remote repository hosted on GitHub

The exercise gets you to...

a. Add a remote and push
b. Clone that pushed repository
c. Make local commits, then push/pull them

You will also create conflicting changes for each other, and practice resolving them

Getting Awesome

Sections in this chapter:

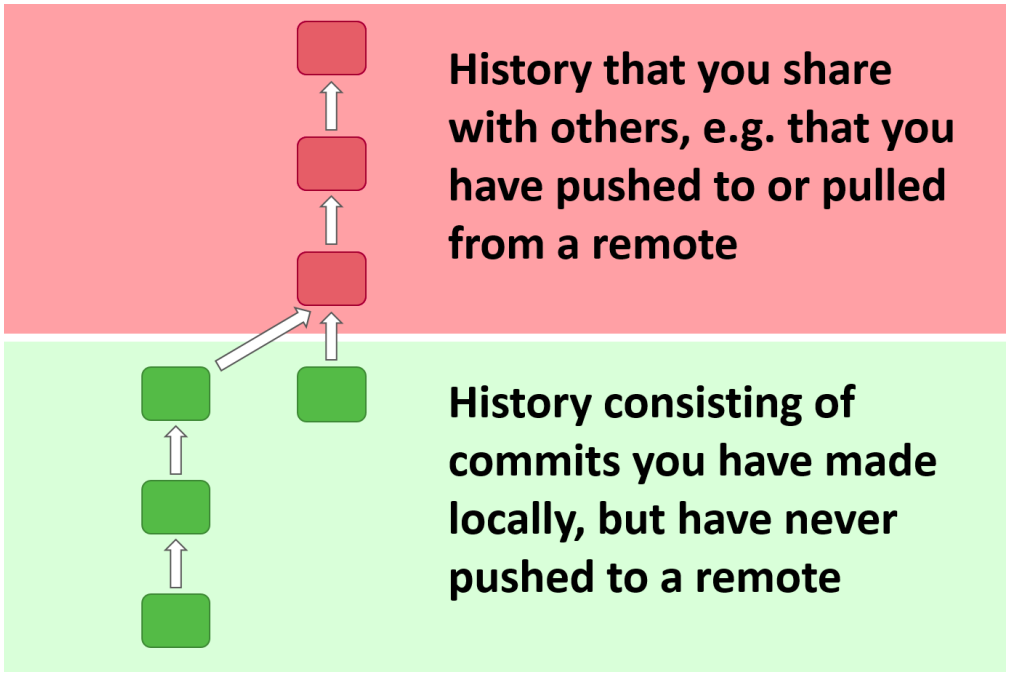
1. The wonder of rebasing
2. Amending existing commits
3. Other useful stuff
4. Closing remark

4-1. The wonder of rebasing

Re-writing history (1)

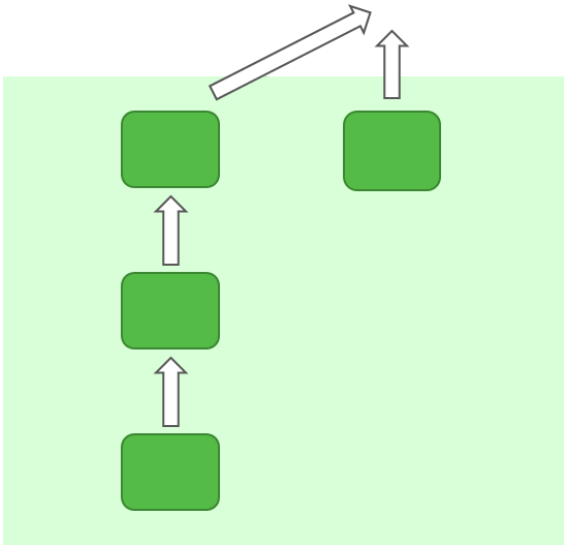
4-1-1

When working with Git and you have remotes, your version history falls into two categories



Re-writing history (2)

4-1-2



Since nobody has seen your local commits, you are free to manipulate that part of the history without causing any problems

Very Important Warning

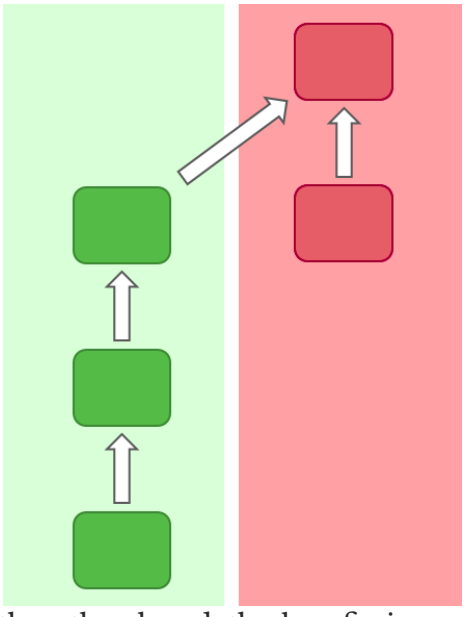
You must be very careful to never change history that you have already shared. The big hint you've got it wrong is that a push will fail, and Git may suggest using --force. DO NOT DO THIS; you will cause problems for others.

Rebasing (1)

4-1-3

You started a branch to work on a feature, and have made a few commits. Then you switched back to master to do a bug fix. You pushed the bug fix, and now return to your feature branch.

Your work on the feature branch is local - you did not push it to a remote



On the other hand, the bug fix is now pushed; it is shared history

Rebasing (2)

4-1-4

A couple of things are now less than ideal...

Your feature branch is missing the bug fix :(
You'd really like to have the bug fix in your branch too; it may affect your feature, and you'd like to be able to test the two in combination

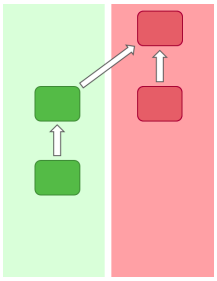
You won't be able to do a fast-forward merge :(
This isn't a huge problem, but merge commits do create a little clutter in the version history; many Git users prefer fast-forward merges

Rebasing (3)

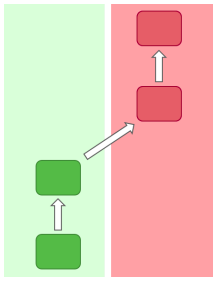
4-1-5

The `rebase` command can help here. In the feature branch, we can run:

```
$ git rebase master
```

In a rebase, Git *takes the commits in your branch, then "replays" them as if the branch had started at the master branch's current commit*, so it has all master does and can fast-forward!



Pulling with rebasing

4-1-6

We've seen that merge commits can also appear when you have local commits, then pull commits from a remote repository

It is often desirable to avoid these merge commits; pulling is such a common scenario that this will quickly clutter the history

Thankfully, it's as simple as:

```
$ git pull --rebase
```

4-2. Amending existing commits

Q Ever made a commit...then instantly realized it had a mistake - perhaps even in the commit message?

4-2-1

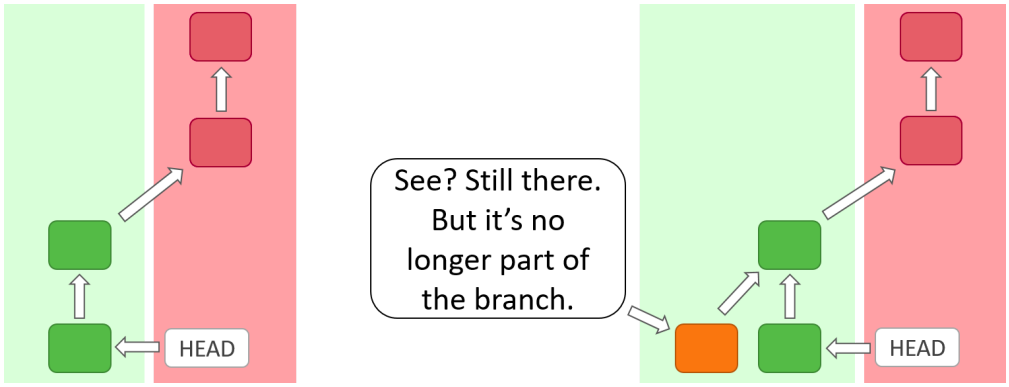
Ⓐ Provided you did not push the commit, this is easy to fix locally; correct any files, and then use the amend flag: *4-2-2*

```
$ git commit --amend
```

This will take the previous commit, incorporate the current changes into it, and use the new commit message you specify also

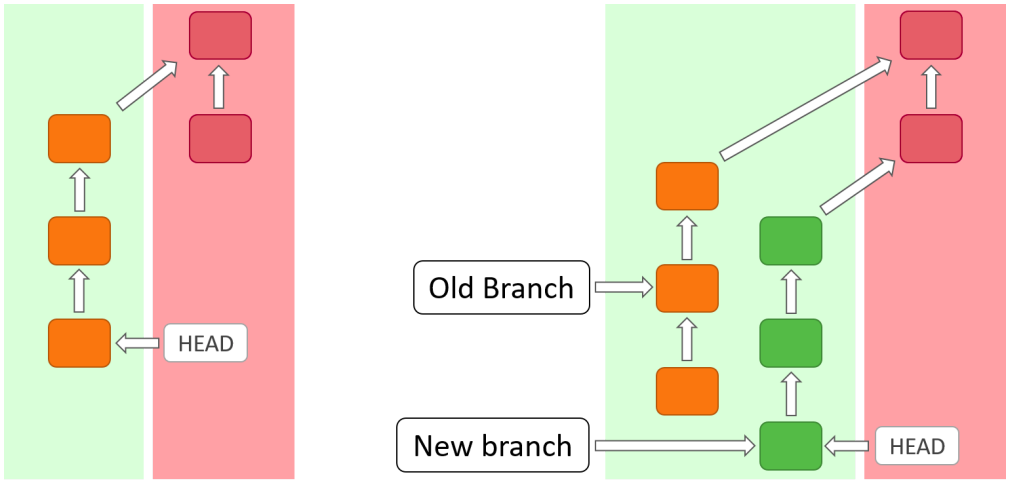
When you amend, old commits don't go away. They're put away to the side and ignored, as if you never made them. *4-2-3*

```
$ git commit --amend
```



When you rebase, commits don't get destroyed during a rebase either. They're just put away to the side. *4-2-4*

```
$ git rebase master
```



Exercise 4

4-2-5

This exercise focuses on re-writing local history

First, you will use the `--amend` option with a commit - it's too useful not to have to hand!

After this, you will look at the difference you get in the DAG when you use rebasing compared to created merge commits

You will be able to visualize the DAG in `gitk`

4-3. Other useful stuff

stash

Git provides a "stack" that you can push your current changes on to...

```
$ git stash
```

Giving you a clean repository again. Later, you can pop the changes from the stack:

```
$ git stash pop
```

Note you can switch branch in the meantime.

When might stash be useful?

cherry-pick

4-3-2

What if you have...

- A master branch
- A feature branch
- A bug fix in the feature branch

You don't want to merge, you just want to take the single commit. Look up the commit ID, then:

```
$ git cherry-pick abc123
```

Done. :)

reset

4-3-3

Another command with loads of different possibilities - check out the help page!

If you want to throw away all of your uncommitted changes (including staged ones):

```
$ git reset --hard
```

You can also use it to clear the staging area:

```
$ git reset
```

Line and hunk level commits

4-3-4

So far, we have seen that we can move files into the staging area, then commit them

If you end up with two changes tangled in a single file, you can also just stage individual lines or hunks rather than the whole file

This is easiest done with another GUI tool:

```
$ git gui
```

Bonus Exercise (If Time)

4-3-5

Review some of the topics covered in this final section of the course.

Pick one of them, and explore it by yourself or, if you prefer, in a small group.

If you're unsure which to choose, then stashing is perhaps the most invaluable one to practice.

If you have time and are comfortable, you may wish to explore a second topic also.

4-4. Closing remark

A paradigm shift

4-4-1

Distributed version control is more a step of *revolution rather than evolution*

Branching becomes lightweight, natural and cheap - both in terms of branch creation and especially thanks to efficient merging

Working locally makes things fast

More complex workflows can be implemented since all copies of a repository are on an equal footing; you give them their place and value

Remember...

4-4-2

In the end, *it's all about commits and the DAG*

Branches are a central part of effective Git usage. It's better to create a branch you then merge right away, than to not create one and wish you had

Rebasing and amending commits are powerful techniques - but you must *only apply them to local history* that has not been shared

Look at your desired development process, and *shape your Git usage* around it - it's malleable!

Evaluation

4-4-3

Please complete the evaluation for this course

What did you like most?

Was anything not as clear as it could have been?

Was there anything that you think should have been done differently?

We take all feedback into account, and use it to help us improve and optimize the course

Thank you! :)

Some other interesting courses offered by Edument directly or through our partners:

Advanced Git
Software Architecture
Applied Domain Driven Design (.Net or Java)
C# Master Class
JavaScript Foundations
Angular2
TDD (.Net or Java)

You can contact us at

info@edument.se

External links

3-1-8 Github: https://www.github.com