TopHat activities

- Please join and indicate that you have done so:

- https://app.tophat.com/login
  Join: **147651**

MORE ADVANCED GIT TOPICS

CO1106 Requirements Engineering and **Professional Practice**

Dr Matthias Heintz , Prof. Shigang Yue
(mmh21@leicester.ac.uk) (sy237@Leicester.ac.uk)

Schedule

Week	Start Date	Monday / Wednesday - Lecture	Thursday / Friday - Surgery	Assessment
26	15/01/2024	Introduction & Why Requirements?	Icebreaker activity for groups & work on Project Description	
27	22/01/2024	Requirements gathering (Quan. & Qual. User Studies)	Work on requirements gathering for Assessment 1	
28	29/01/2024	Functional Requirements	Work on building list of funct. requirements for Assessment 1	
29	05/02/2024	Non-Functional Requirements	Work on building list of non-funct. requirements for Assessment 1	
30	12/02/2024	Overview of UML; Use Case diagrams and descriptions	Work on Use Case diagram and Use Case description for Assessment 1	
31	19/02/2024	Basics of git version control	Checkout and setup group git repository and set up Weekly Log .md file	Assessment 1 (50%)
32	26/02/2024	More advanced git topics	Work on reworked list of functional requirements	
33	04/03/2024	Class Diagrams	Work on Class diagram	
34	11/03/2024	Class Modelling	Rework Class diagram	
35	18/03/2024	Sketching and Lo-fi prototyping	Work on wireframes/lo-fi prototypes	
36	25/03/2024	Software Laws & Professionalism	**none**	Effective use of Git (10%)
37-40	01/04/2024	break	break	
41	29/04/2024	none	none	Blackboard Test (40%)

Matthias
Shigang

Session objectives

- At the end of the lecture you will be able to:
  - *Explain what ‘branching’ in git is and why we would want to do it*
  - *Explain what it means to ‘merge’ branches*
  - *Be able to resolve ‘content conflicts’ when merging two branches*

Group coursework in CO1106

- Main group project
  - ***Part 1 (50% - due 23rd February)***
    - Project description (10%)
    - Quantitative and qualitative studies (10%)
    - Written requirements (20%)
    - Use Case UML Diagram and Use Case Description (10%)
  - ***Part 2 (10% - due 27th March)***
    - Effective usage of git version control

Group Coursework Part 2 – Effective use of git version control

- Your group will utilise a git repository in order to manage/submit any files produced as part of the second part of the group project for CO1106 (details of how to access the repository will be provided to you in the tutorial of Week 6).
- Groups should make frequent usage of their group repository - any shared files that you work on (for example, the .md files containing functional requirements and use case descriptions) should be added to the repository as soon as they are made, with regular changes being committed by group members until that particular file is finished. Each group will be responsible for coordinating their git usage.

Group Coursework Part 2 – Instructions

- A maximum of 3 marks are available, depending on how effectively your group used git. We will decide the number of marks you receive by inspecting the contents of your repository as well as the commit history of your repository:
  - For 1 mark, at least one member of your group needs to make a commit each week; no advanced features (i.e., branching/merging) have been used, and commit descriptions (included when you made the commit) may be non-descriptive and not give a good idea of the changes included in a particular commit.
  - For 2 marks, multiple commits should be made each week, and an initial (possibly incomplete) version of the artefact worked on during each tutorial session needs to be submitted in the week it was worked on. Commit messages must be descriptive (but succinct) and give a good idea of the changes that have been committed.
  - For 3 marks, you must satisfy all points from the previous two bullets, and it should be apparent from your commit log that all members of the group have

Group Coursework Part 2 – Weekly log

- Each group must also produce **a 'Weekly Log' (.md format)** that is updated with the following contents in each of the weeks 6-10 (5 weeks in total):
  - A 'beginning of week' entry containing a summary of the work that the group plans to complete during that week, along with a breakdown of which members will complete which tasks. The beginning of week entry for each week should be produced by the group; for example, during your first groupwork meeting of that week.
  - An 'end of week' entry which lists the tasks that each member of the group has completed; any outstanding work; and any other additional information that your group feel is relevant to add .

Group Coursework Part 2 – Weekly log (continued)

- Each group will be responsible for designing the Markdown structure of their Weekly Log, ensuring that it is easy to read and maintained properly. The entries in the Weekly Log will be checked on a weekly basis (the entry for Week X will be checked by us during Week X+1). For the entries of Week X, there is a maximum of 0.75 marks available (up to a total of 3 marks for weekly updates):
  - 0.25 marks depending on whether both the 'beginning of week' and 'end of week' entries have actually been added to the log (if either one is missing, you receive 0 marks for that week)
  - 0.5 marks will be awarded depending on the quality of the entry (is it descriptive enough? does it contain all the information listed above?)
  - An additional mark out of 4 will be awarded based on the readability/quality of the Weekly Log document.

Marking rubrics

Markdown Usage, Marking Rubric					
Fail	Poor	Requires Improvements	Satisfactory	Good	Excellent
0	0.5	1	2	3	4
No serious attempt is made.	Required information is contained in the document but no Markdown syntax has been utilised in order to improve readability.	A limited amount of basic Markdown syntax has been used to produce a document that is readable but not aesthetically pleasing or easy to navigate. There may be some syntactical errors in the Markdown usage that cause the document to look untidy.	Basic Markdown syntax has been used to reasonable effect. The produced document is navigable and its contents is clearly displayed. Most audiences should have little to no problem with understanding the document, but it is not particularly aesthetically pleasing.	Same as Satisfactory, but with a number of more advanced Markdown features used in order to improve the readability of the document. Any member of the intended audience would be able to navigate the document easily.	Same as Good, but a clear effort to make the document aesthetically pleasing as well as easily navigable has been made.

- **GitLab** Repositories **will be set up for each group**; before this week's tutorial sessions, where you will clone them for the first time.
- You'll be required to commit a number of components (Class Diagrams, Sketches, Prototypes) to your group repository, including a 'Weekly Log'
- To create the weekly logs and other text-based files, you'll be using a very simple markup language called **Markdown**

GIT RECAP

Retrieving an old file version

- To retrieve an old version of a file we need the unique ID of the commit that contains that version. To find that out we use ‘`git log` ...’

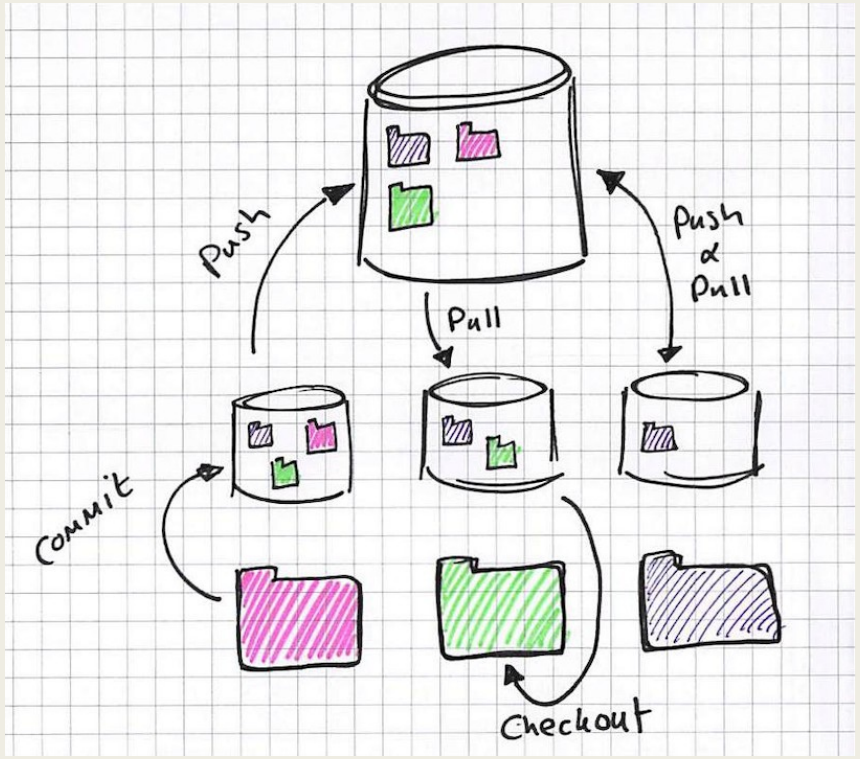
```
jakob@jakob-T430:~/A$ git log --oneline
7b6393e (HEAD -> master) Modified files A, B and C
23e7dab Modified files A, B, C
455f978 Modified fileB.txt
e36c65b Modified fileB.txt
ca3add3 changes
297b095 Fixed bug in fileB.txt
```

- The command provides a list of all commits in the repository history, and we can select the one containing the version of the file we wish to revert to:

```
jakob@jakob-T430:~/A$ git checkout 7b6393e fileA.txt
jakob@jakob-T430:~/A$ git commit -m "reverted fileA.txt to version 7b6393e"
[master cfcd484] reverted fileA.txt to version 7b6393e
 1 file changed, 1 insertion(+), 1 deletion(-)
```

GitLab

- A **repository** is hosted on a central server (e.g., on the internet)
- Collaborators download the repository, then make changes to files **locally**
- When changes are complete, the collaborator **pushes** the changes to the remote repository
- Other collaborators can then **pull** the changes to incorporate them into their own local copy



SIMPLE FILE SHARING

When simple file sharing becomes a problem

Worker 1 maintains a .html file for his business called ‘users-report.html’ which tracks the number of daily users/hits/sales made on their e-commerce website:

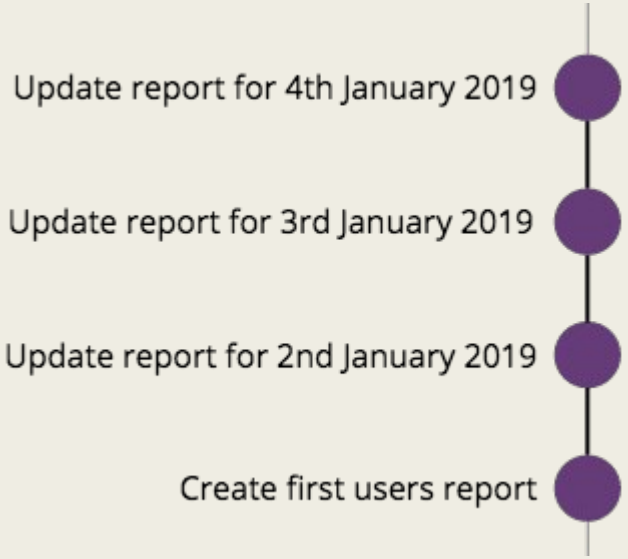
```
Users Report

Date: 14-Sep-2018

Users: 23

Hits: 98

Sales: 12
```

```html
<html>
  <body>
    <p>Users Report</p>
    <p>Date: 14-Sep-2018</p>
    <p>Users: 23</p>
    <p>Hits: 98  </p>
    <p>Sales: 12</p>
  </body>
</html>
```

He changes each value in the file everyday to the most up-to-date figures, and saves it to a shared drive on a central server

Note: Git is **NOT** being used by the company

After updating the file every day for a while, the worker ends up with a file history that resembles this:



It is a **linear sequence** of ‘file versions’, but each version overwrites the previous version.

Recap question: How could git be used in order to ensure that the data from previous dates is not lost?

https://app.tophat.com/login
Join: **147651**

Two concurrent projects on the same file...

In the meantime, Worker 2 begins work on improving the look of the reports, aiming for something like:

Users Report

Date 📅

14-Sep-2018

Users 👩‍👧

23

Hits 🏹

98

Sales 🎁

12

Users Report

Date

4th January

Users

23

Hits

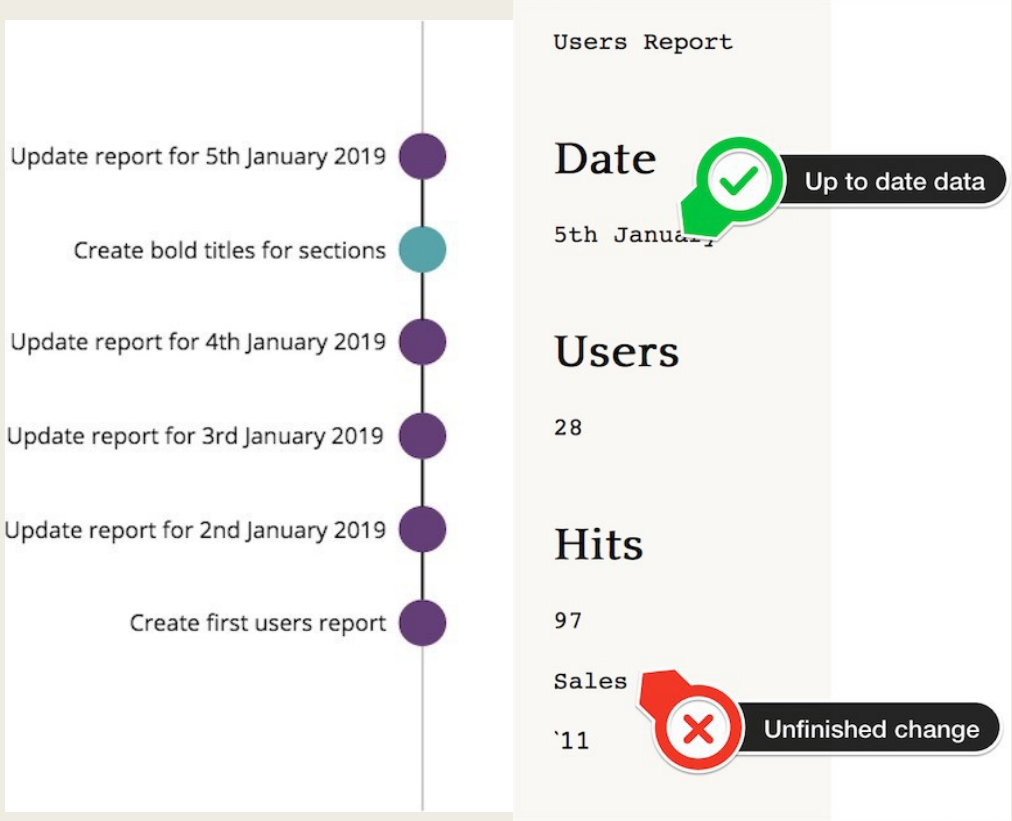
98

Sales

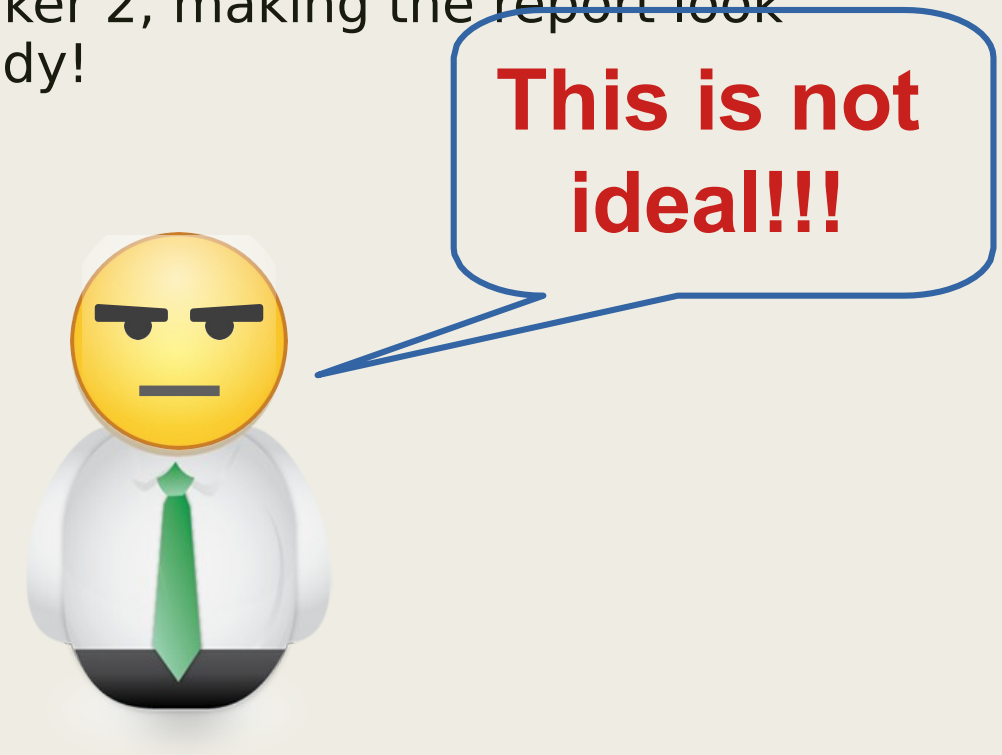
`12

She begins stylising the headings but gets as far as ‘Hits’ before realising she has a meeting and saves the file:

But what about the state of the next daily report?



Worker 1 can now see incomplete changes made by Worker 2, making the report look untidy!



Why is it a problem?

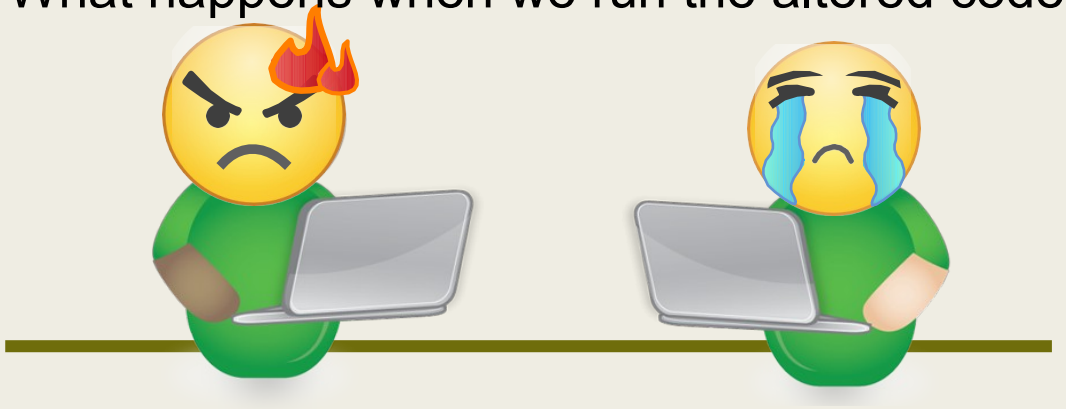
- In our toy example, the mix of content from the old/new versions is annoying, but not fatal
- In bigger projects, conflicts such as this can be a major problem **and lead to broken code**
- What if two developers are working on the same file at different times, and one changes a variable name **without refactoring the existing code**?

Clashes with …

```
#variables
x = 100

# Peter's old code
for i in range(x):
        print(x)
```

File before the change...

What happens when we run the altered code?



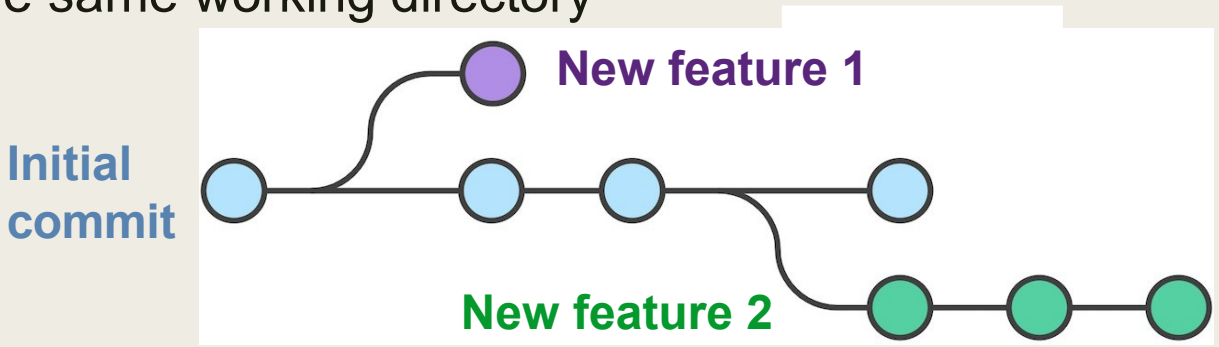
```
#variables
size = 100

# Peter's old code
for i in range(x):
        print(x)

#Dave's new code
y = 0
for j in range(size):
        y += j*3
print(y)
```

File after the change...

GIT

How can we avoid this?
with ‘git branching’…

- Share changes with other collaborators, but control **when** and **which** changes to integrate into our local version of the repository.
- **Branching** lets us create *‘divergence points’*, from which we can maintain different histories of the same working directory



- We start with an initial commit, then create a new **branch** each time there is a feature/change that we would like to work on **in isolation**

Creating a branch

- Just as each commit has a unique ID, each branch has a unique name too
- The **‘default branch’** is known as **master.** You choose the name of subsequently created branches.
- We can create a new branch using the command:

‘`git branch <name_of_branch>`’:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/B$ git branch new_branch
```

- We can use ‘`git branch`’ (without specifying a branch name) to verify:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/B$ git branch
* master
  new branch
```

Checking out (i.e., switching between) branches

- At any point in time, a collaborator is considered to be **'on'** a single branch:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/B$ git branch
* master
  new_branch
```

Current branch is highlighted in green

- We can switch to another branch using '`git checkout <name_of_branch>`';
- Your WD becomes the **most recent commit on the branch called `<name_of_branch>`**:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/B$ git checkout new_branch
Switched to branch 'new_branch'
```

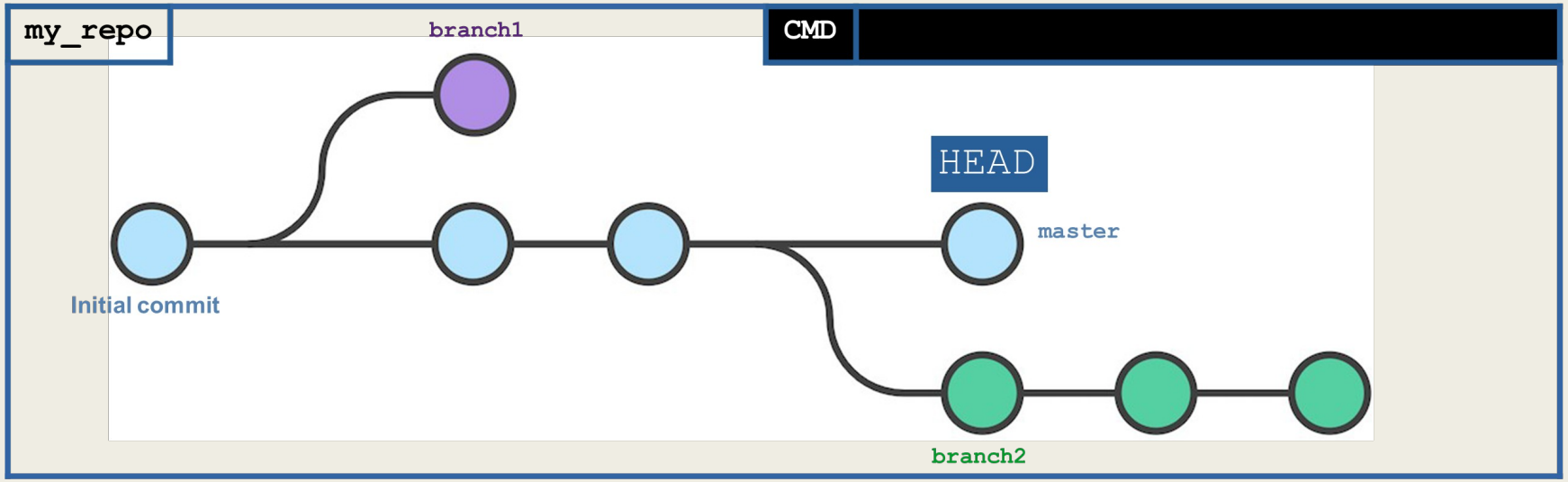
- We then use '`git branch`' again to verify that we've made the switch

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/B$ git branch
  master
* new_branch
```

- Technically speaking, a 'branch name' is a pointer that points to the newest commit on the repositories history path

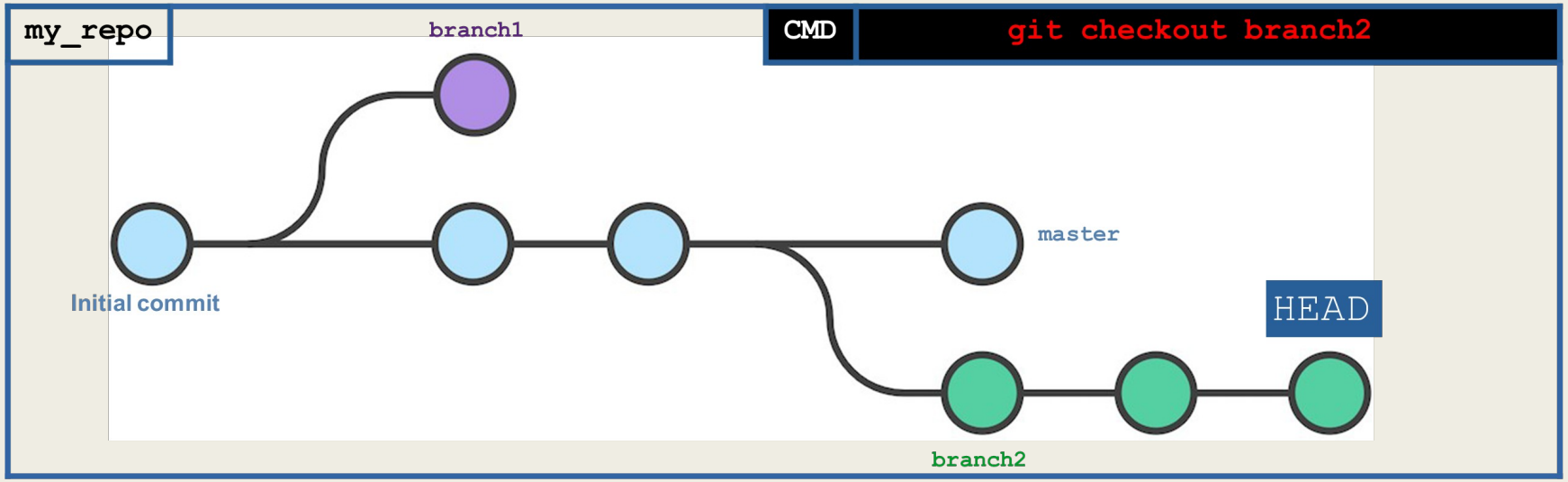
Working on a branch

- Git also maintains a pointer (called ‘`HEAD`’) to the commit that represents our **current working directory**
- When you checkout a branch, you checkout the most recent commit on that branch and `HEAD` is updated



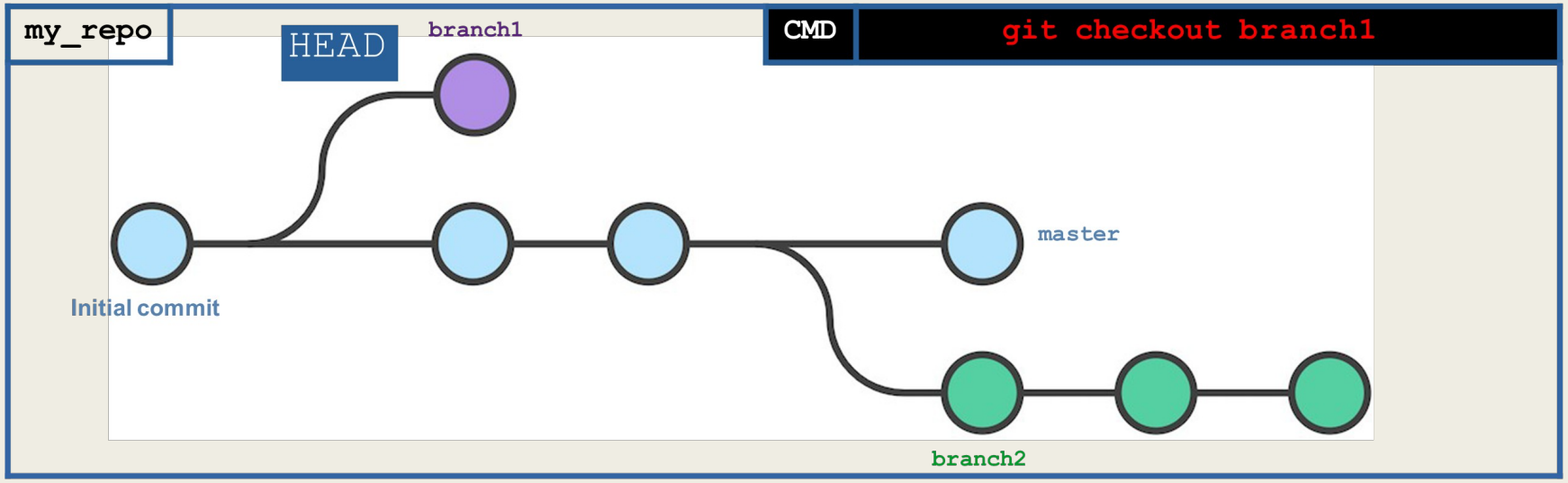
Working on a branch

- Git also maintains a pointer (called ‘`HEAD`’) to the commit that represents our **current working directory**
- When you checkout a branch, you checkout the most recent commit on that branch and `HEAD` is updated



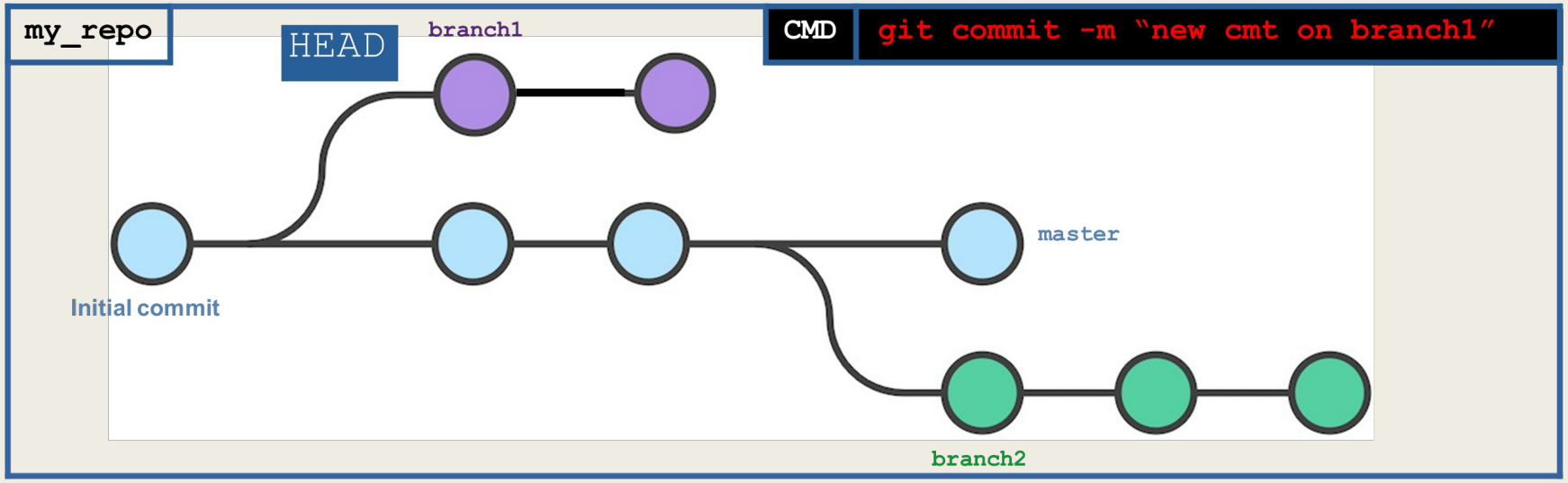
Working on a branch

- Git also maintains a pointer (called ‘`HEAD`’) to the commit that represents our **current working directory**
- When you checkout a branch, you checkout the most recent commit on that branch and `HEAD` is updated



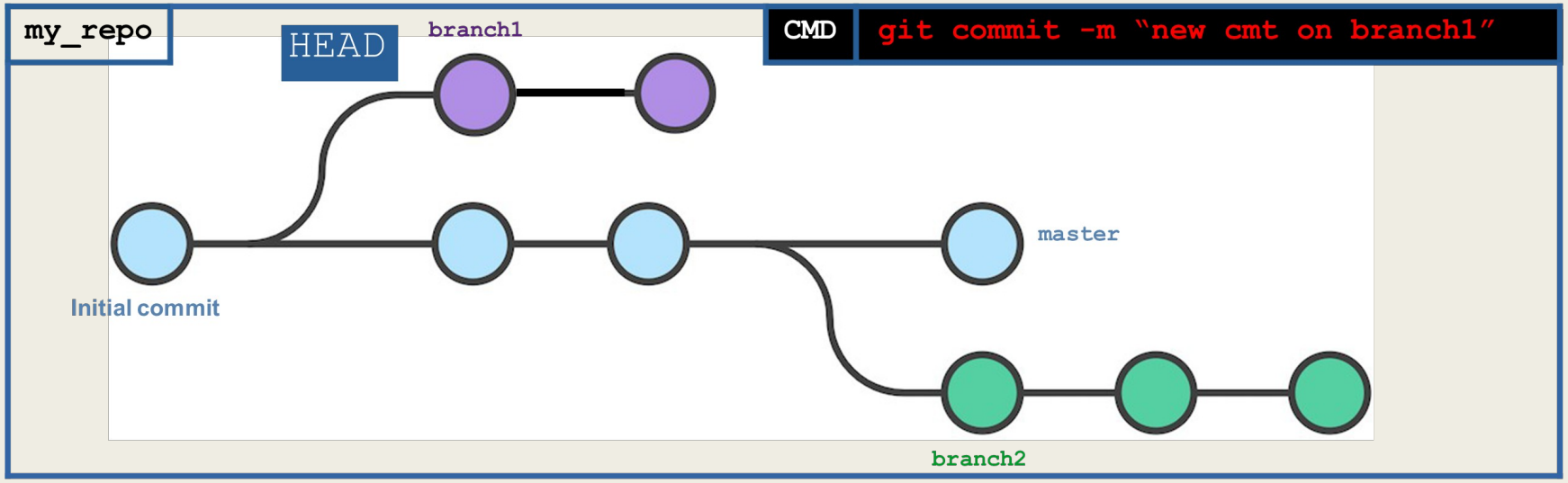
Working on a branch

- If we now make changes on branch1 and commit, these changes are recorded as part of the history of branch1, and **no other branch**



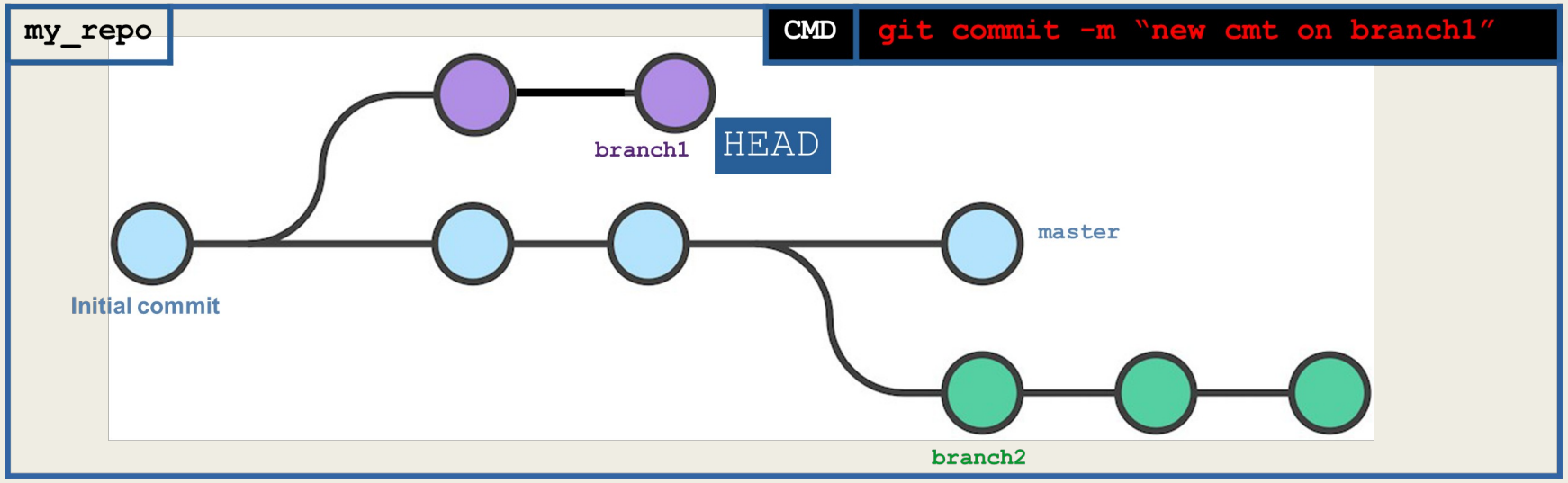
Working on a branch

- If we now make changes on branch1 and commit, these changes are recorded as part of the history of branch1, and **no other branch**



Working on a branch

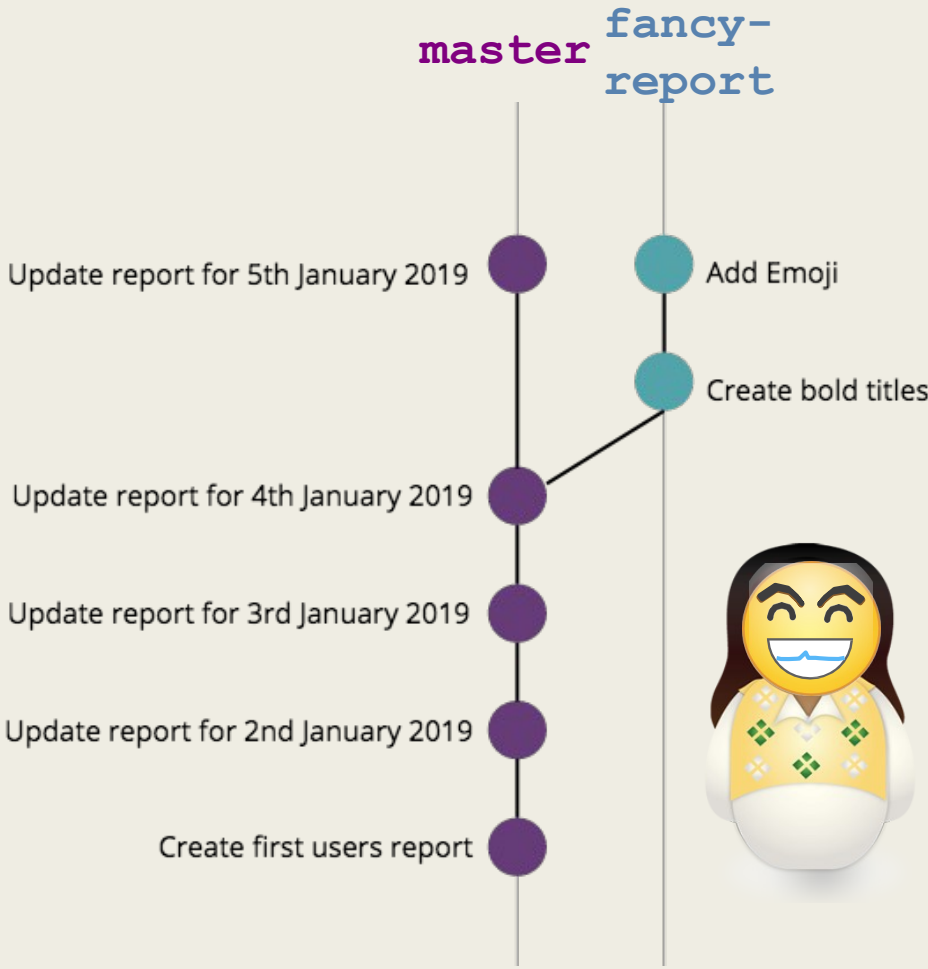
- If we now make changes on branch1 and commit, these changes are recorded as part of the history of branch1, and **no other branch**
- Additionally, the branch1 pointer is updated to point to the newest commit on branch1 (and so is the HEAD pointer, since that commit is our current WD)



Back to our first example...

Worker 1 and 2 could set up their shared file in a git repo:

- Worker 1 uses the `master` branch to make his daily updates
- Worker 2 creates a branch called `'fancy-report'` and keeps her unfinished report design work there until it is complete



- **This should solve the problem for a while...**

DRAW-ALONG EXERCISE

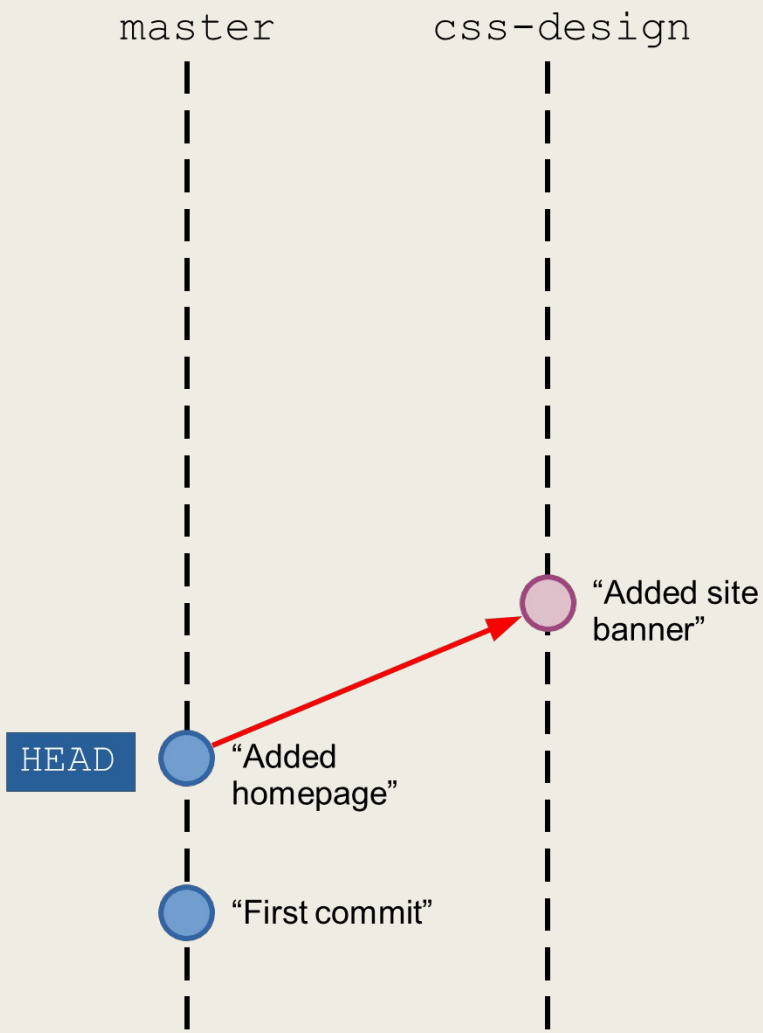
Draw-along exercise ✍️

Using the diagram style from the previous slide, try and map out the following sequence of commits/branch creations related to a website project:

1. Clone a repo with a single commit on master “First commit”
2. Make a commit to master with the message “Add homepage”
3. Create and checkout a branch called ‘**`css-design`**’ and make a commit called ‘Added site banner’.
4. Checkout master again.

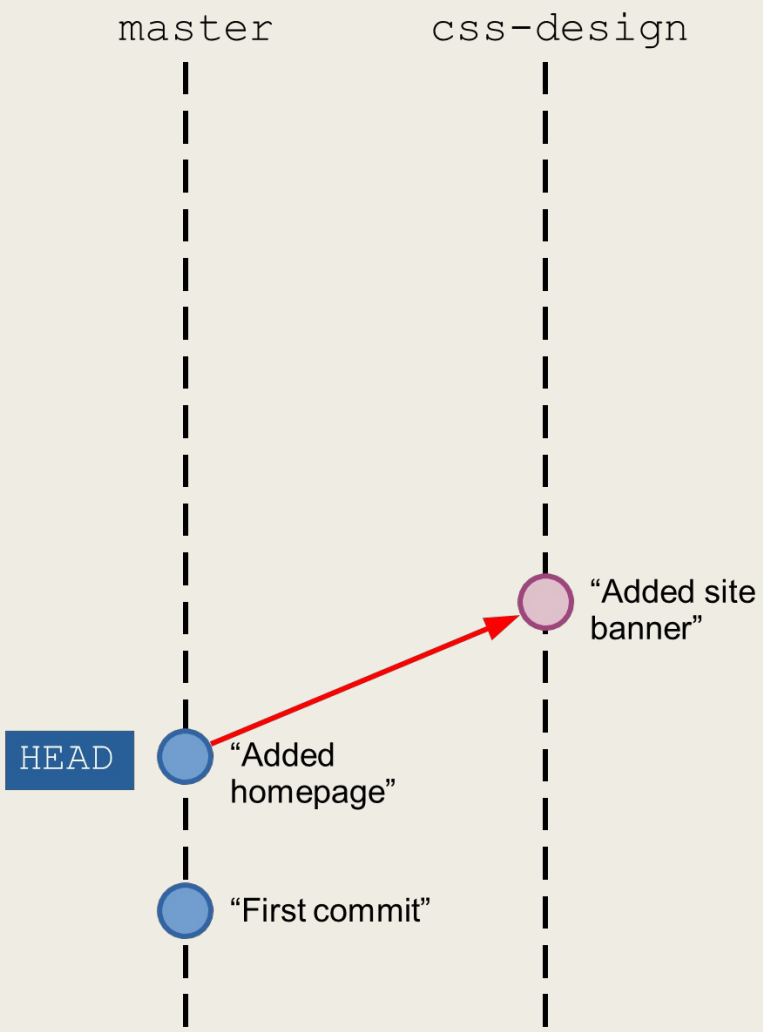
Have a go for 5 minutes, then we’ll look at the solution and add a few more steps to try

After step (4) we have:



1. Clone a repo with a single commit on master called "First commit"
2. Make a commit to master with the message "Add homepage"
3. Create and checkout a branch called '**`css-design`**' and make a commit called 'Added site banner'.
4. Checkout master again.

After step (4) we have:

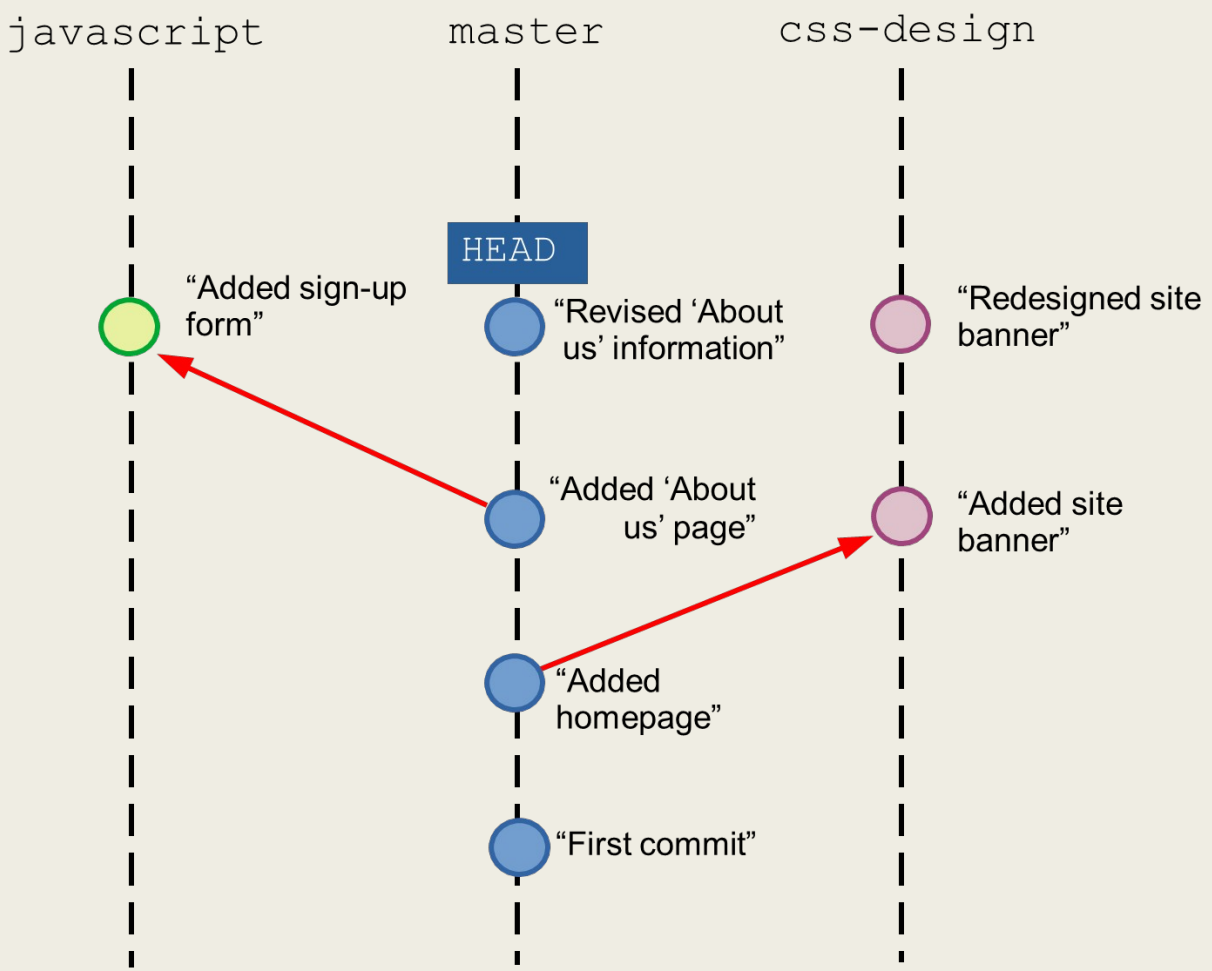


- ☑ 1. Clone a repo with a single commit on master called "First commit"
- ☑ 2. Make a commit to master with the message "Add homepage"
- ☑ 3. Create and checkout a branch called '**`css-design`**' and make a commit called 'Added site banner'.
- ☑ 4. Checkout master again.

Some additional changes:

- ☐ 5. Make a commit called "Added 'About us' page"
- ☐ 6. Checkout reviews "css-design" branch and make a commit called "Redesigned site banner"
- ☐ 7. Checkout master, then create/checkout a branch called '**`javascript`**' and make a commit called 'Added sign-up form'.
- ☐ 8. Checkout master again and make a commit called "Revised 'About Us' information"

The finished repository:

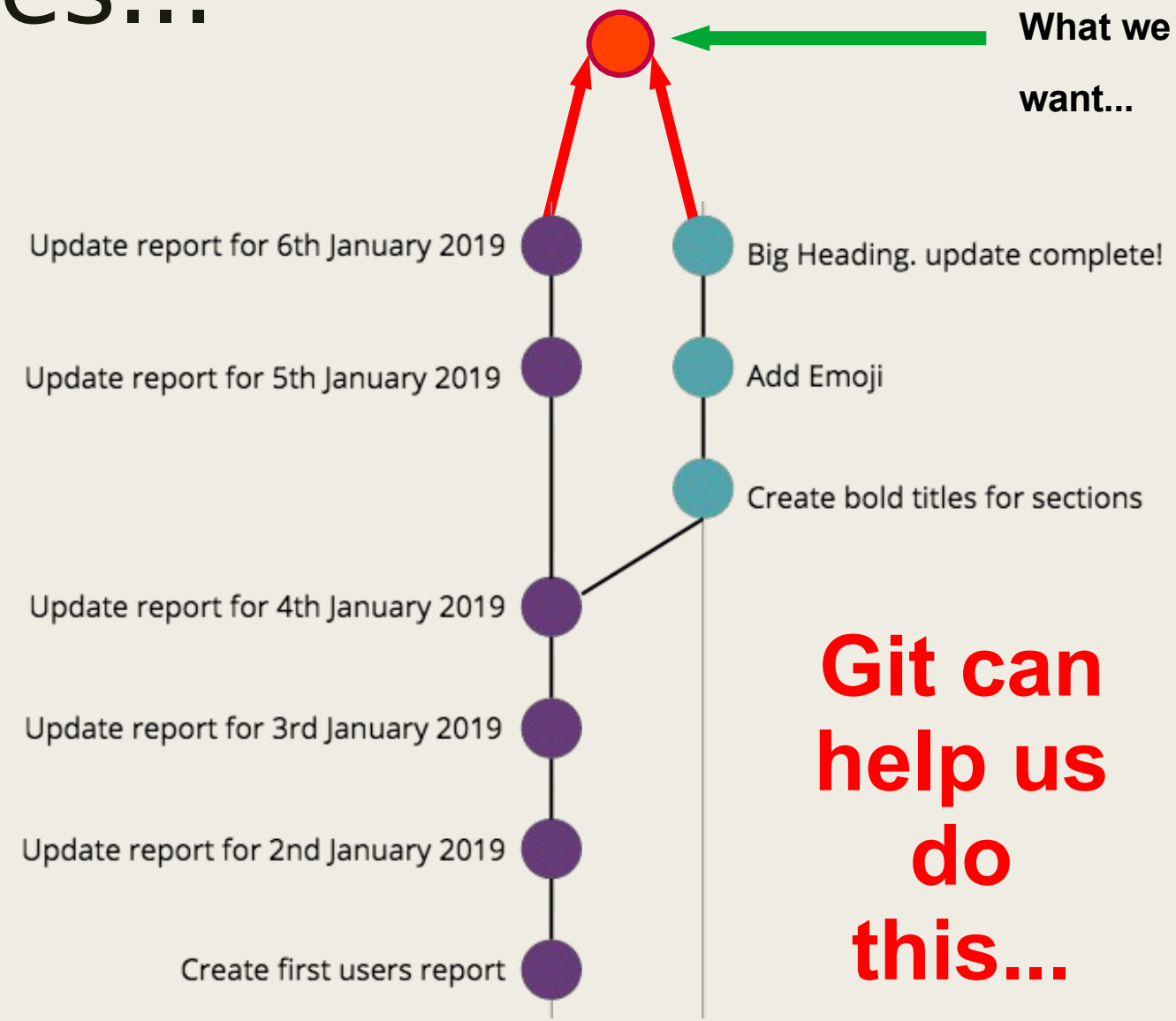


- ☑ 1. Clone a repo with a single commit on master called "First commit"
- ☑ 2. Make a commit to master with the message "Add homepage"
- ☑ 3. Create and checkout a branch called '**`css-design`**' and make a commit called 'Added site banner'.
- ☑ 4. Checkout master again.
- ☑ 5. Make a commit called "Added 'About us' page"
- ☑ 6. Checkout **`css-design`** branch and make a commit called "Redesigned site banner"
- ☑ 7. Checkout master then create/checkout a branch called '**`javascript`**' and make a commit called 'Added sign-up form'.
- ☑ 8. Checkout **`master`** again and make a commit called "Revised 'About Us' information"

MERGING BRANCHES

Merging branches...

- Eventually, Worker 2 finished her revamp of the report design
- However, Worker 1's version contains the latest data and not the new formatting, and vice versa for Worker 2
- They'd like to be able to combine the two files to create a single version...



Git can help us do this...

The desired effect:

Up-to-date data but boring style

Users report

Date: 6th January

Users: 35

Hits: 99

Sales: 16

Master

Users Report

Date

6-Jan-2018

Users

35

Hits

99

Sales

16

Up-to-date data AND fancy styling!

Users Report

Date

4-Jan-2018

Users

23

Hits

98

Sales

Fancy-report

Fancy styling but out-of-date data

Master:

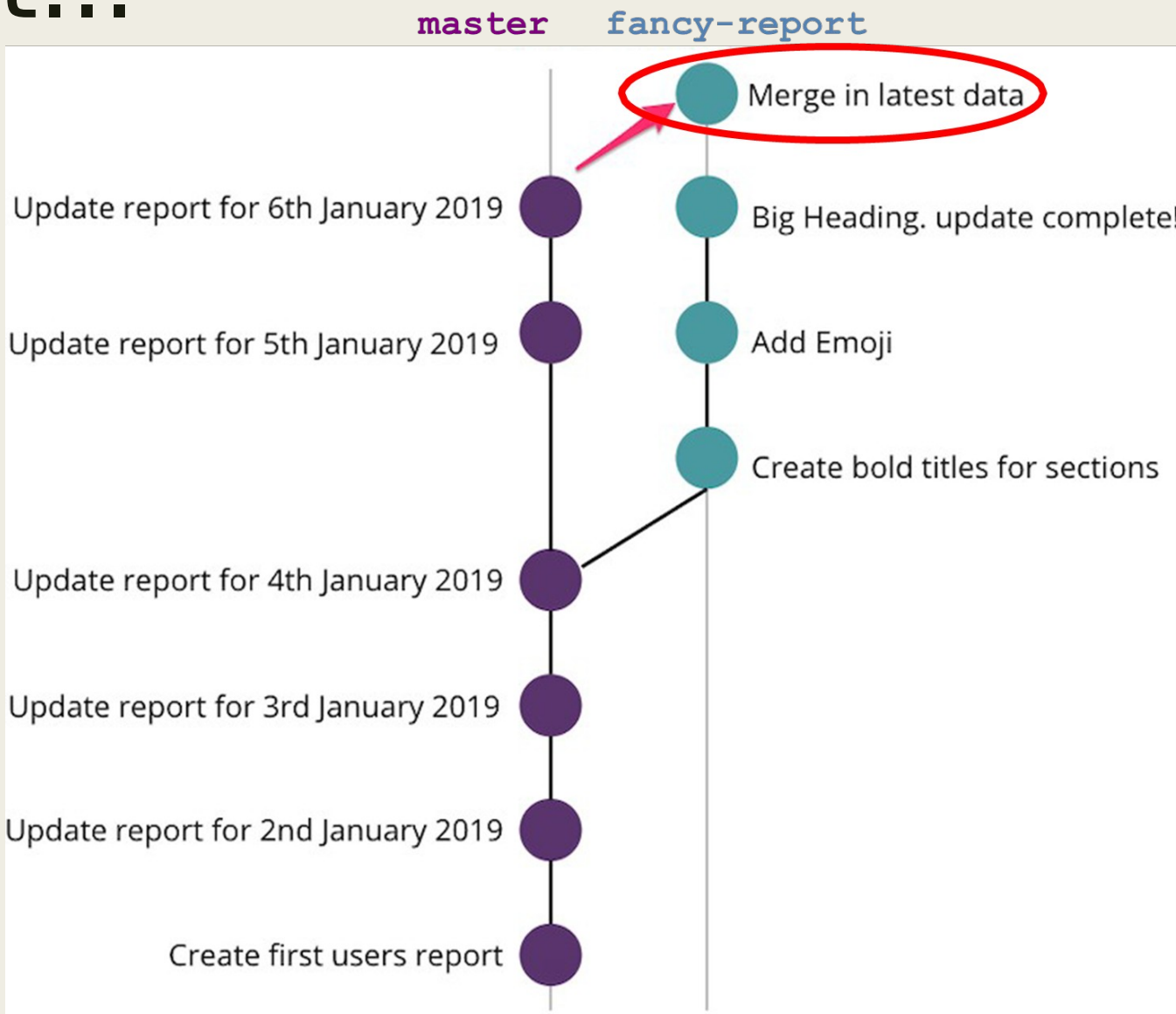
```
<p>
Date: 6th January
</p>
<p>
Users: 35
</p>
```

Fancy-report:

```
<h2>
  Date
  <img
  src="calendar.png"/>
</h2>

<p>4-Jan-2018</p>

<h2>
  Users
  <img src="users.png"/>
</h2>

<p>23</p>
```

Merging with git…

- You merge the contents of a chosen branch with your current branches content
- For example, we could merge '**`master`**' into '**`fancy-report`**' to obtain the most up-to-date data
- This creates a new commit on the **`fancy-report`** branch, containing both the up-to-date data, and the styling



Merging with git…

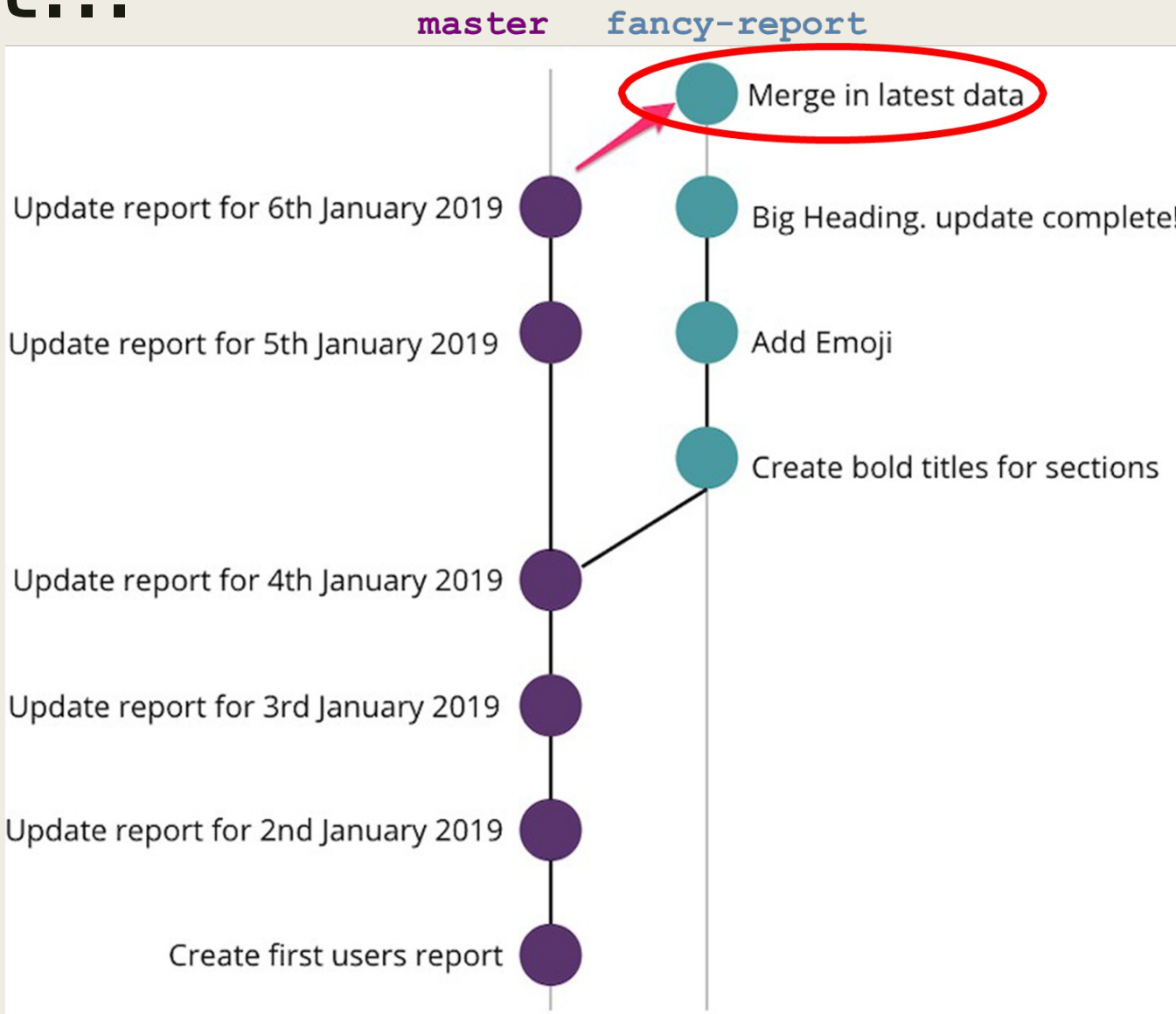
Assuming that Worker 1 has **pushed his changes to the central repo,** to achieve this you would write:

```
git pull

git checkout fancy-report

git merge master
```

You should always:

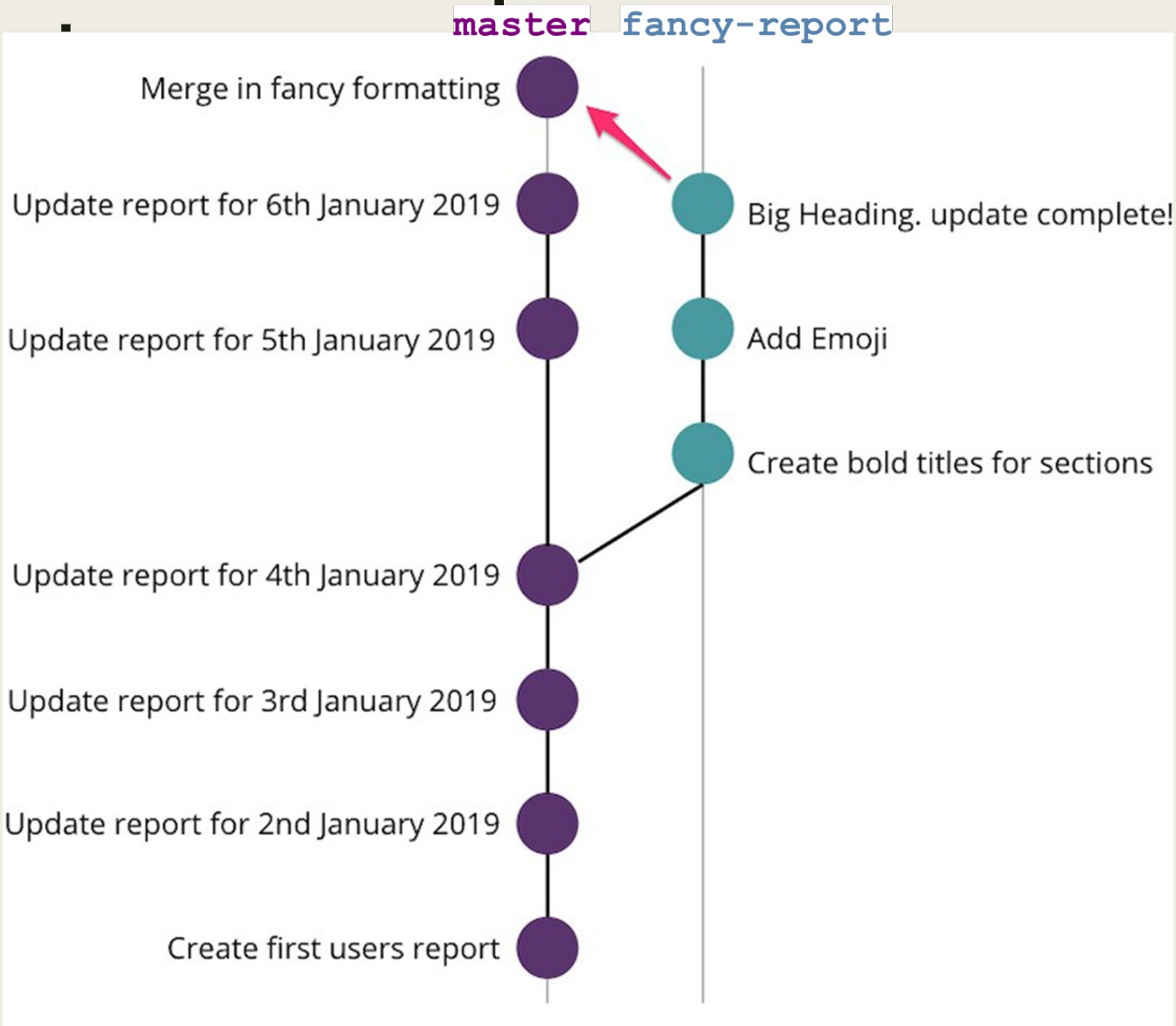
1. Checkout the branch you wish to **merge into**

2. Do ‘`git merge <branch>`’, where `<branch>` is the name of the branch containing the **desired content**



Merging a branch-developed feature into ma

- Assume that Worker 2 has now finished stylising the report in her own branch and pushed changes
- Now Worker 1 wants to continue with his daily updates, but with the style included

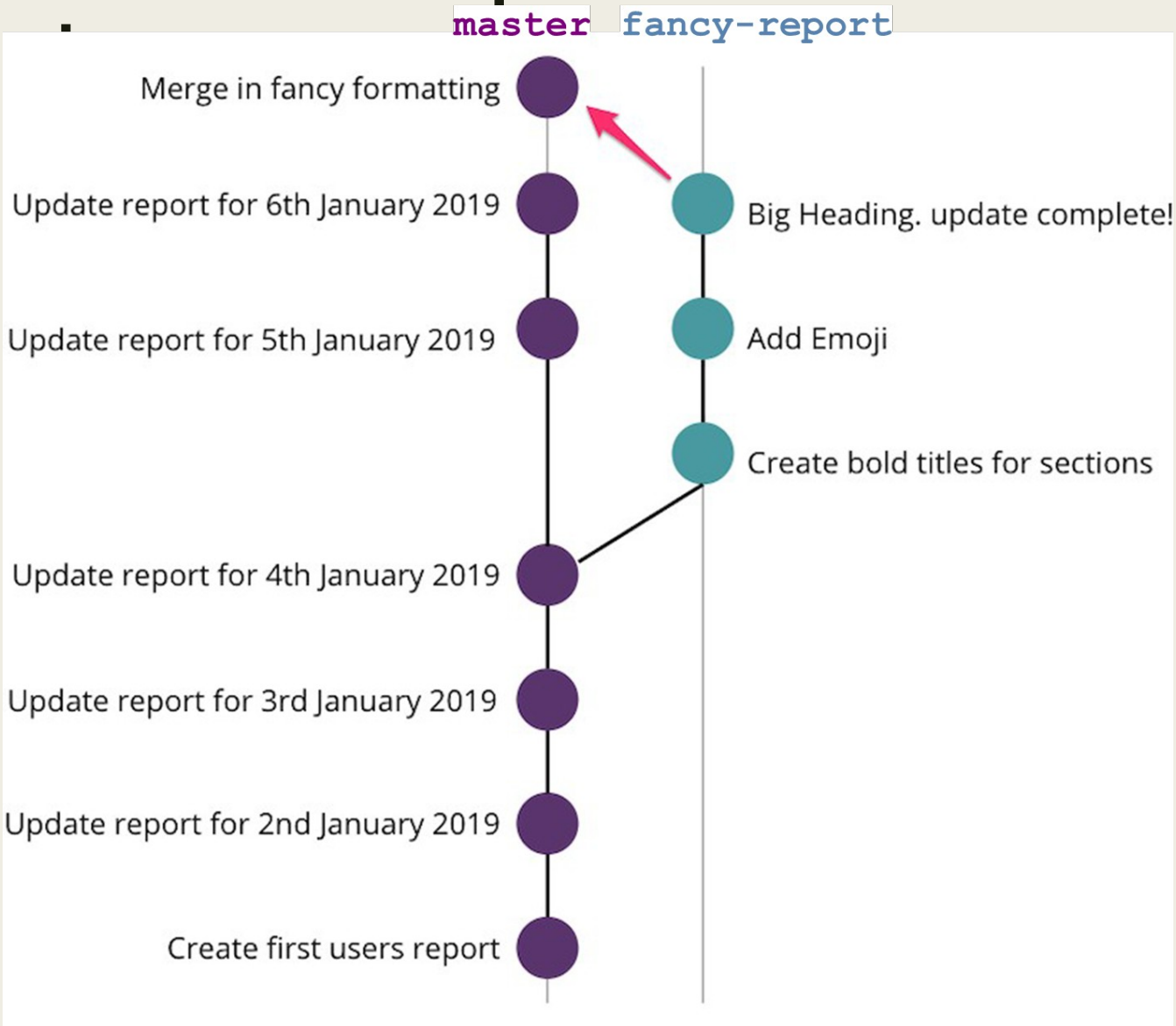


Merging a branch-developed feature into ma

- Assume that Worker 2 has now finished stylising the report in her own branch and pushed changes
- Now Worker 1 wants to continue with his daily updates, but with the style included

Question: Which commands would Worker 1 need to do in order to achieve this?

https://app.tophat.com/login
Join: **147651**



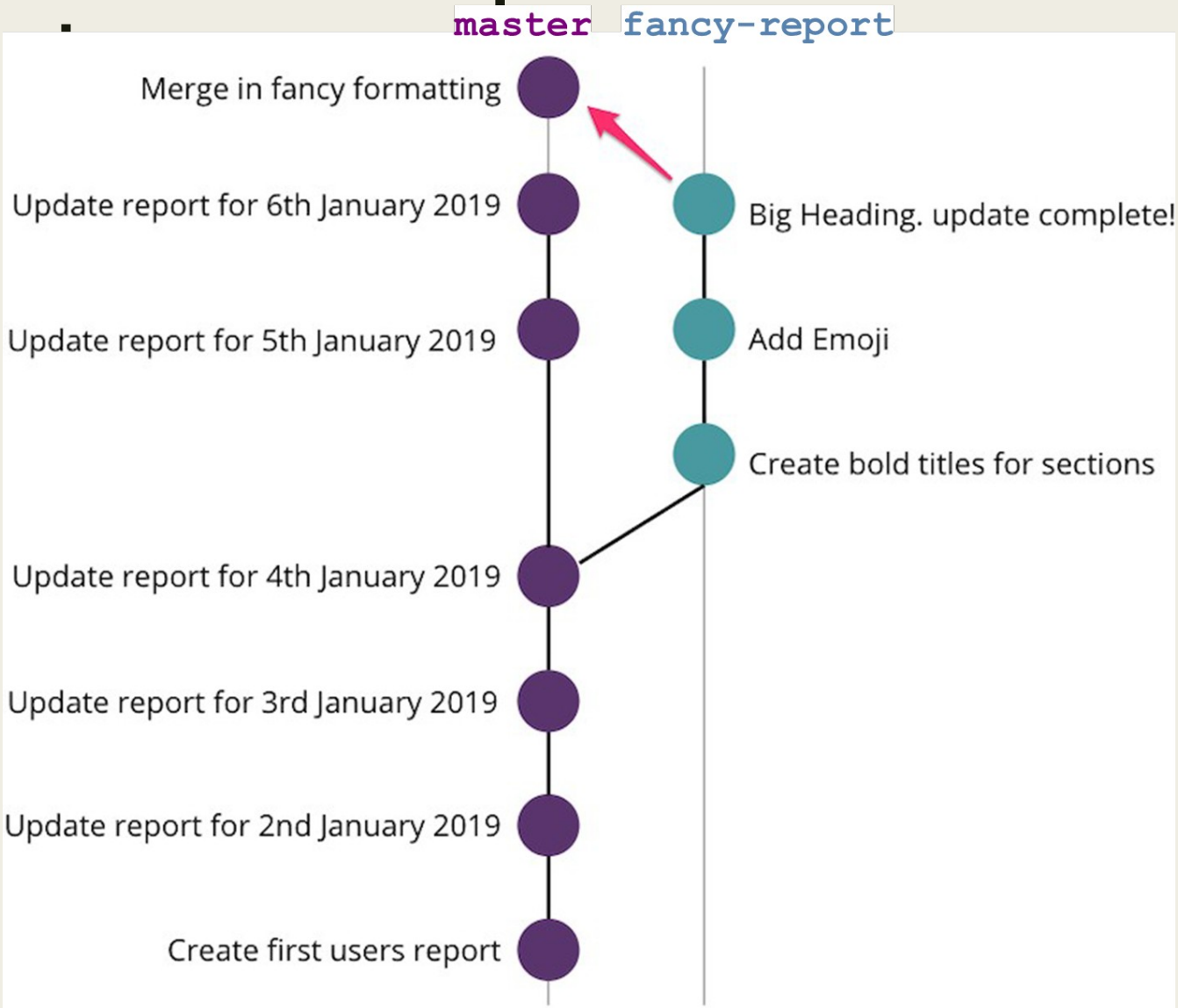
Merging a branch-developed feature into ma

- Assume that Worker 2 has now finished stylising the report in her own branch and pushed changes
- Now Worker 1 wants to continue with his daily updates, but with the style included

Question: Which commands would Worker 1 need to do in order to achieve this?

Answer:

```
git pull
git checkout master
git merge fancy-report
```

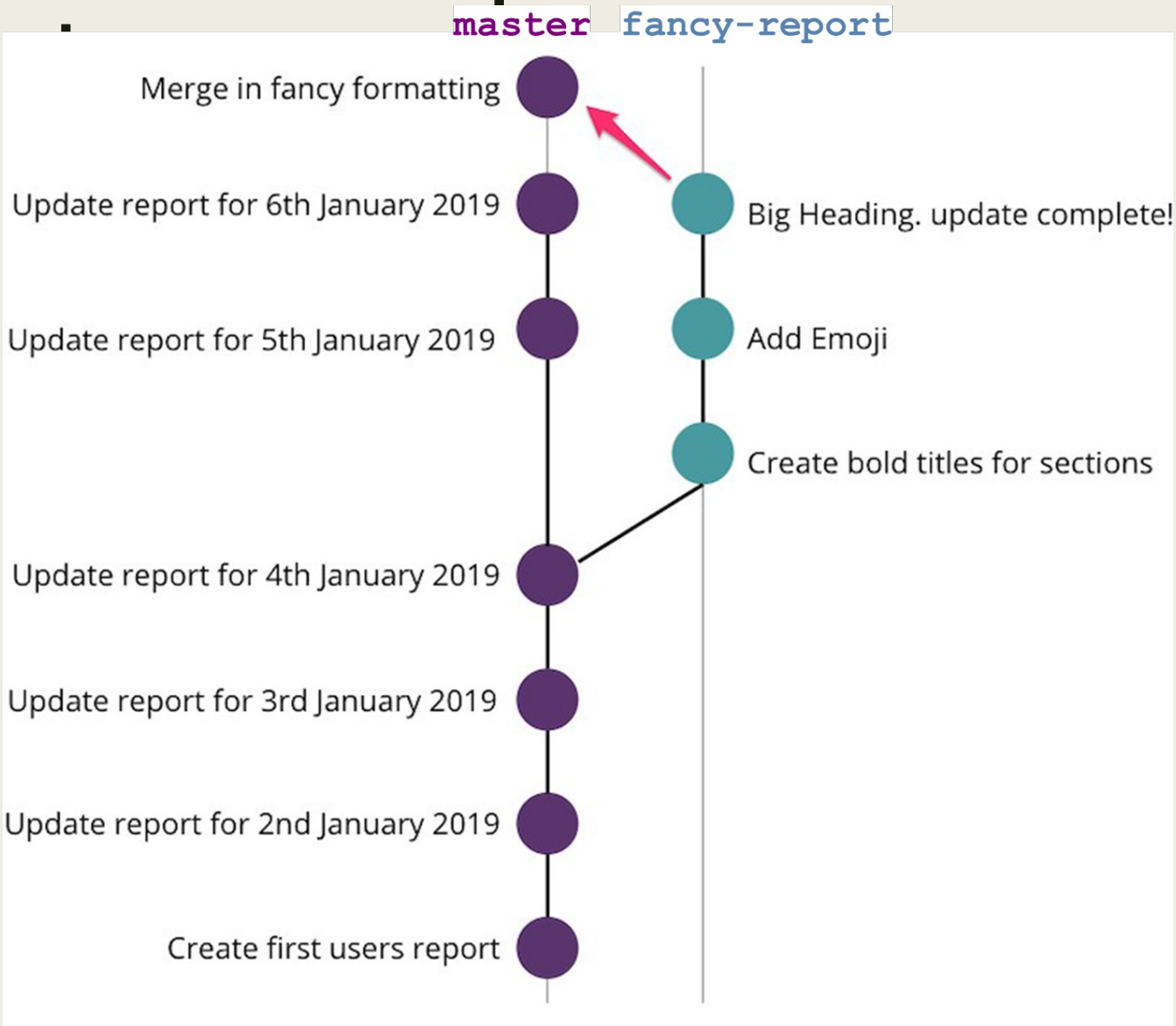


Merging a branch-developed feature into ma

- Assume that Worker 2 has now finished stylising the report in her own branch and pushed changes
- Now Worker 1 wants to continue with his daily updates, but with the style included

Question: What would happen if Worker 2 has not already merged in the latest data from Worker 1's report?

https://app.tophat.com/login
Join: **147561**



Deleting a branch

- Once we've finished developing a feature and have merged it back into **`master`**, it's good practice to delete that branch using the command:

  '`git branch -d <branch_name>`'

- Note the usage of the '**`-d`**' (i.e., 'delete') flag. We get the following output:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/git_sandbox/B$ git branch -d new_branch
Deleted branch new_branch (was 5a52c24).
```

- If there are still changes on **`<branch_name>`** that are yet to be merged, git also warns us of this and we get a chance to merge them before deleting:

```
jakob@jakob-IdeaPad-5-15ITL05:~/Desktop/git_sandbox/B$ git branch -d new_branch
error: The branch 'new_branch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D new_branch'.
```

Pushing branch changes to a remote repo

Point 1: Whenever you do a ‘git push’ command, it only pushes changes/commits on the currently-checked out branch, and not others.

E.g.: if you currently have **`master`** checked out and make a commit, but there is also an unpushed commit on **`branch1`**, then **`git push`** will only publish your **`master`** commits, and **not those on `branch1`**.

Pushing branch changes to a remote repo

Point 2: The first time you push changes on a locally-created branch **`<branch_name>`** to the remote, you must **create that branch on the remote.**

Do this with the command:

```
git push -u origin <branch_name>
```

All subsequent pushes can be done using the standard '**`git push`**' command

Relationship between push/pull and merge

Note that whenever we do a git push or a git pull, a git merge is usually involved:

When we do a `git pull`, changes in the remote are merged into our local copy

When we do a `git push`, changes in our local copy are merged into the remote copy

?Recap questions?

- **Question 1**:
  In which situation are branches typically used and why are they useful?

https://app.tophat.com/login
Join: **147561**

- **Question 2**:
  Why is it important not to let two branches ‘diverge’ too far from each other?

MERGE CONFLICTS

Merge conflicts:
When merging goes wrong

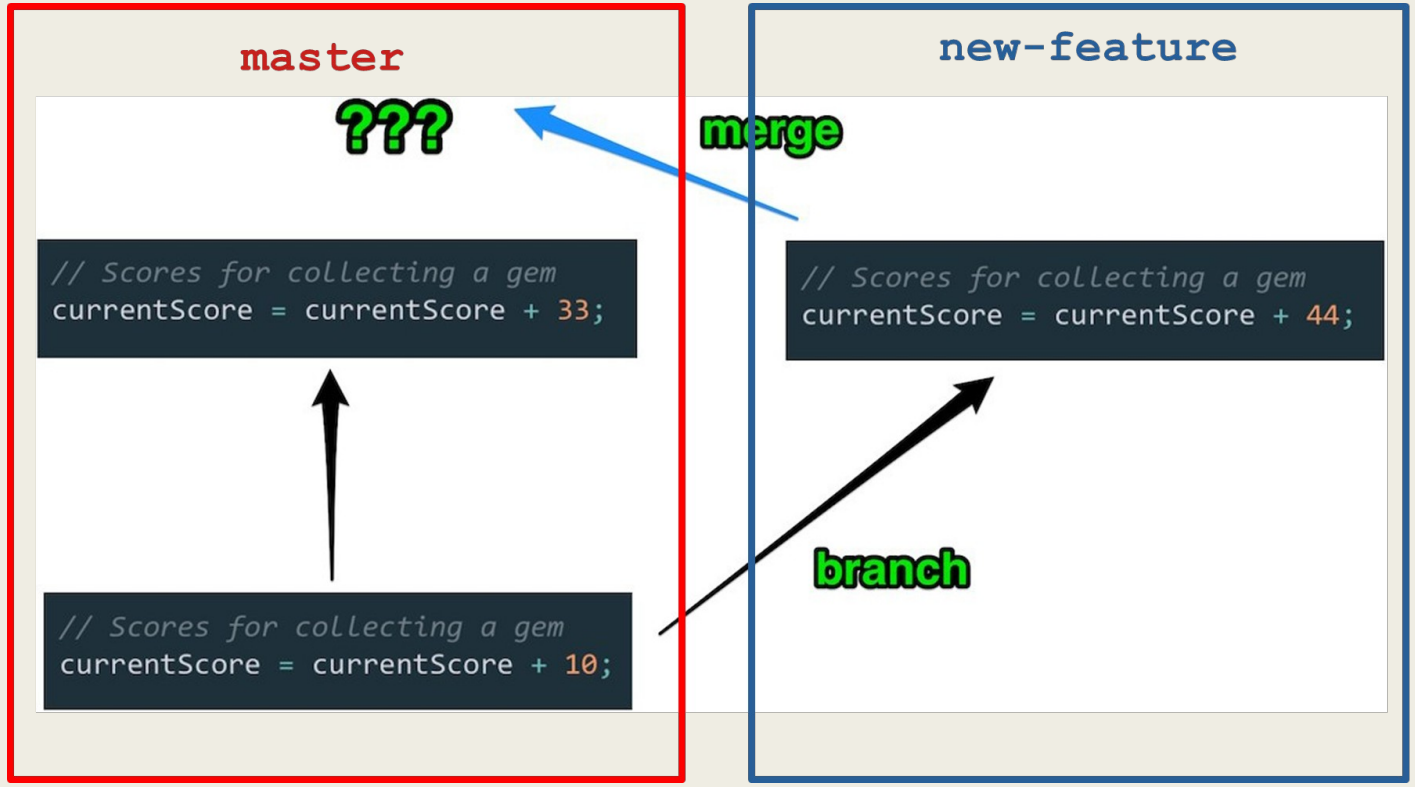
Merge conflicts:
When merging goes wrong

Merge conflicts happen when we attempt to merge two branches and **2 conditions are met**:

- Both branches contain some version of a particular file
- The two versions of the file differ **in the same line (or possibly multiple lines)**

So how do we know if a merge conflict has occurred, and how can we put it right?

A simple example: Some code for a game



Note that the assignment to variable '`currentScore`' differs between branches...

Question: What happens when we try to merge the changes made in the `new-feature` branch back into `master`?

https://app.tophat.com/login
Join: **147561**

Resolving a merge conflict

Git makes it easy for us by **editing the conflicting file** in the ‘merged-into’ branch’s working directory and highlighting the conflict:

```
// Scores for collecting a gem
<<<<<<< HEAD
currentScore = currentScore + 33;
=======
currentScore = currentScore + 44;
>>>>>>> new-feature
```

We then delete the annotations added by git, **remove the unwanted content**, save the file, then do a new commit...

About using branching / merging for CW2

- Branches are an extremely useful tool in practice and it is a good idea to use them for large scale projects
- The idea of this lecture was to introduce you to them and how to use them, **but…**
- It is up to your group to decide on your workflow and manage your git collaboration throughout CW2
- You are **not required** to use branches, but there are some marks available

Have a look in the assesment brief

What’s next?

- In this week’s tutorial you will:
  - *Complete a short exercise that will guide you through creating a branch, merging changes on that branch back into ‘main’, and resolving a merge conflict*
  - *Create Markdown document that will list the functional requirements of your system*

- Tutorials this week in different rooms!
  (for PC access)

- Lecture next week:
  UML Class Diagrams

Questions?

- Dr. Matthias Heintz or Prof. Shigang Yue
  - *mmh21@leicester.ac.uk or sy237@Leicester.ac.uk*
  - *Microsoft Teams*
  - *Office 613 or 608 in Ken Edwards Building*

UNIVERSITY OF LEICESTER