Initial commit

How to use git in the typycal workflow

What is `git`?

- Git (/gɪt/) is a distributed **version-control system** for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows[1].

Version Control

A component of software configuration management, *version control*, also known as *revision control* or *source control*, is the management of changes to documents, computer programs, large web sites, and other collections of information[2].

Who made git?

- Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano[1].

What is for

Tracking changes in source code

- The primary use case for *git* is to track changes in the source code during software development.

Non-linear workflows

- This means that we can develop the software on different parallel contexts without worrying about concluding this or that feature first.

Collaboration

- Non-linear workflows combined with the distributed nature of the repositories allow collaboration among teams on different computers.

What is NOT for

Tracking changes of non-textual files

- Aimed at tracking changes in source code files, *git* is not intended for binary and compressed files, especially if large. The indexing mechanism works on text differences among versions, which cannot be exploited when using files created by the machine.

File synchronization

- Born as a development tool, *git* should not be used to synchronize files between machines. The history should reflect the development cycle of the software.

Sensitive information

- As the repository could be accessible by the most different kinds of users, it should contain only information enabling the development of the software while any sensitive information should not be added.

Hands-on

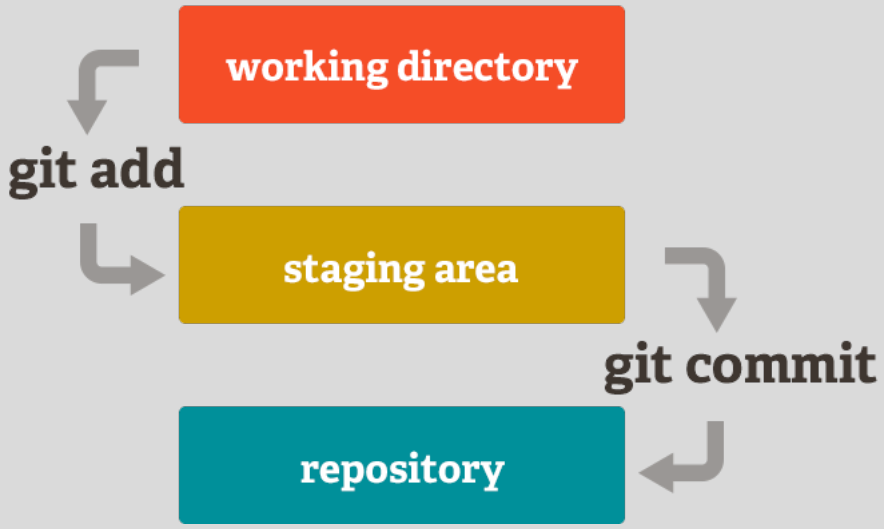
`init`

We begin by creating an empty repository using the `init` command.

```
marco@macbook ~/gittutorial> git init
Initialized empty Git repository in
/Users/marco/gittutorial/.git/
```

Tracking changes

Reviewing changes



- Before taking a snapshot and freezing a point in time we want to review the changes first. To achieve this, git implements
  - a **tracking index** which gives us evidence of the changes in the code base and
  - a **staging area**, in which we can temporarily put the changes we want to commit.
- Each change in the directory then can be in one of these states:
  - Untracked
  - Tracked and not staged for commit
  - Tracked and staged for commit

status

- After creating a file in the working directory, we are going to check the tracking INDEX.
- The tracking INDEX is empty as the file just created has to be added to it with the `add` command.

```
marco@macbook ~/gittutorial> touch hello.py
marco@macbook ~/gittutorial> git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.py
nothing added to commit but untracked files present (use "git add" to track)
```

add

- We are going to start tracking the modifications in the file and add it to the staging area by calling the `add` command.

```
marco@macbook ~/gittutorial> git add hello.py
marco@macbook ~/gittutorial> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.py
```

commit

- To create a persistent snapshot of this state we have to *commit* the changes in the staging area. To perform this operation we rely on the `commit` command.

```
marco@macbook ~/gittutorial> git commit -m "hello.py
added"
[master (root-commit) 488f920] hello.py added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello.py
```

log

- Now we can have a look at the commit history of the current repository by invoking the `log` command.

```
marco@macbook ~/gittutorial> git log
commit 488f92022e914b619fddc42937a3a4e71a4c2794
(HEAD -> master)
Author: marco <marco.nazzari@euclidea.com>
Date:   Mon Nov 18 12:57:39 2019 +0100

    hello.py added
```

log

- The previous history was a bit meager, as we had just one commit. A real-world scenario would look like this:

```
marco@macbook ~/brave-browser> git log --oneline
* f046083 (HEAD -> master) run npm install for muon
* 0fa0bcd cleanup options
* e266172 add readme
* b4b0fd7 use git urls
* 3f67923 muon build
* f0b159e default submodule sync to the root dir
* 2f915db cleanup
* ce6fa47 update node build config
* 35cadfb add build script
* fb92683 change name to sync
* 696b977 bootstrap script
```

What is HEAD?

The HEAD is the commit we are referring to at the present moment. It is the point in time we are pointing to and the files in the working directory (except not tracked ones) will be at that point in time.

checkout

- The `checkout` command allows us to move the HEAD to a different point in time, once specified the relative *hash* in the commit history.

```
marco@macbook ~/brave-browser> git checkout 0fa0bcd

Note: checking out '0fa0bcd'.


You are in 'detached HEAD' state. You can look around,
make experimental changes and commit them, and you can
discard any commits you make in this state without
impacting any branches by performing another checkout.

If you want to create a new branch to retain commits
you create, you may do so (now or later) by using -b
with the checkout command again. Example:


  git checkout -b <new-branch-name>


HEAD is now at 0fa0bcd cleanup options
```

checkout

```
marco@macbook ~/brave-browser> git log --oneline
* 0fa0bcd (HEAD) cleanup options
* e266172 add readme
* b4b0fd7 use git urls
* 3f67923 muon build
* f0b159e default submodule sync to the root dir
* 2f915db cleanup
* ce6fa47 update node build config
* 35cadfb add build script
* fb92683 change name to sync
* 696b977 bootstrap script
```

- The checkout command allow also to move to different *branches* of the repository, but we'll talk about that later.

Undoing changes

Undoing Changes – Working Directory

- To restore changes in the working directory at the last commit, we perform a `checkout` on the single files or on the entire project directory and eventually perform a `clean`.
- WARNING: after this operation all uncommitted and unstaged modifications will be lost!

```
marco@macbook ~/gittutorial> git checkout .
marco@macbook ~/gittutorial> git clean
marco@macbook ~/gittutorial> git status
On branch master
nothing to commit, working tree clean
```

Undoing Changes – Working Directory

- If we want to keep them to reapply them later we can store them in a temporary commit with the `stash` command.

```
marco@macbook ~/gittutorial> git stash
Saved working directory and index state WIP on master:
488f920 hello.py added
```

- These modifications will be added in a list of temporary commits which could be reapplied later. This list is a LIFO (Last In First Out) queue which means that when calling `stash pop` the last commit added will be applied

```
marco@macbook ~/gittutorial> git stash list
stash@{0}: WIP on master: 488f920 hello.py added
stash@{1}: WIP on master: 488f920 hello.py added
marco@macbook ~/gittutorial> git stash pop
Dropped refs/stash@{0}
(b5f40d23dca6363771421d2e0a1399ce261697e2)
```

Undoing Changes – Staging Area

- To remove all changes in the staging area, we perform a `reset` without specifying any parameter. We can reset also single files and folders by appending their paths after the command.

```
marco@macbook ~/gittutorial> git reset
Unstaged changes after reset:
M   hello.py
```

Undoing Changes – Commit History

- To get back to a previous commit, we are going to perform a `reset --hard` command.
- WARNING: This operation will restore the working directory and the staging area to the specified commit but we are going to loose FOREVER the commit history from that point onwards.

```
marco@macbook ~/brave-browser> git reset --hard
0fa0bcd
HEAD is now at 0fa0bcd cleanup options
```

Undoing Changes – Commit History

- A safer way to undo commits is represented by the `revert` command.
- `revert` does not erase any commits, instead it creates a new commit that reverts the previous ones, in order to restore the tracked files to the desired point.

```
marco@macbook ~/brave-browser> git log --oneline
* 8412a1b47 (HEAD -> master) Revert "add readme"
* f04608392 run npm install for muon
* 0fa0bcdd9 cleanup options
* e26617272 add readme
* b4b0fd736 use git urls
* 3f6792320 muon build
* f0b159eb5 default submodule sync to the root dir
* 2f915dbb5 cleanup
* ce6fa4789 update node build config
* 35cadfb71 add build script
* fb9268326 change name to sync
* 696b977eb bootstrap script
```

Review of the commands

Working directory

(modify files) *vs* `checkout`, `clean`

Tracking index

`add` *vs* `reset`

Commit history

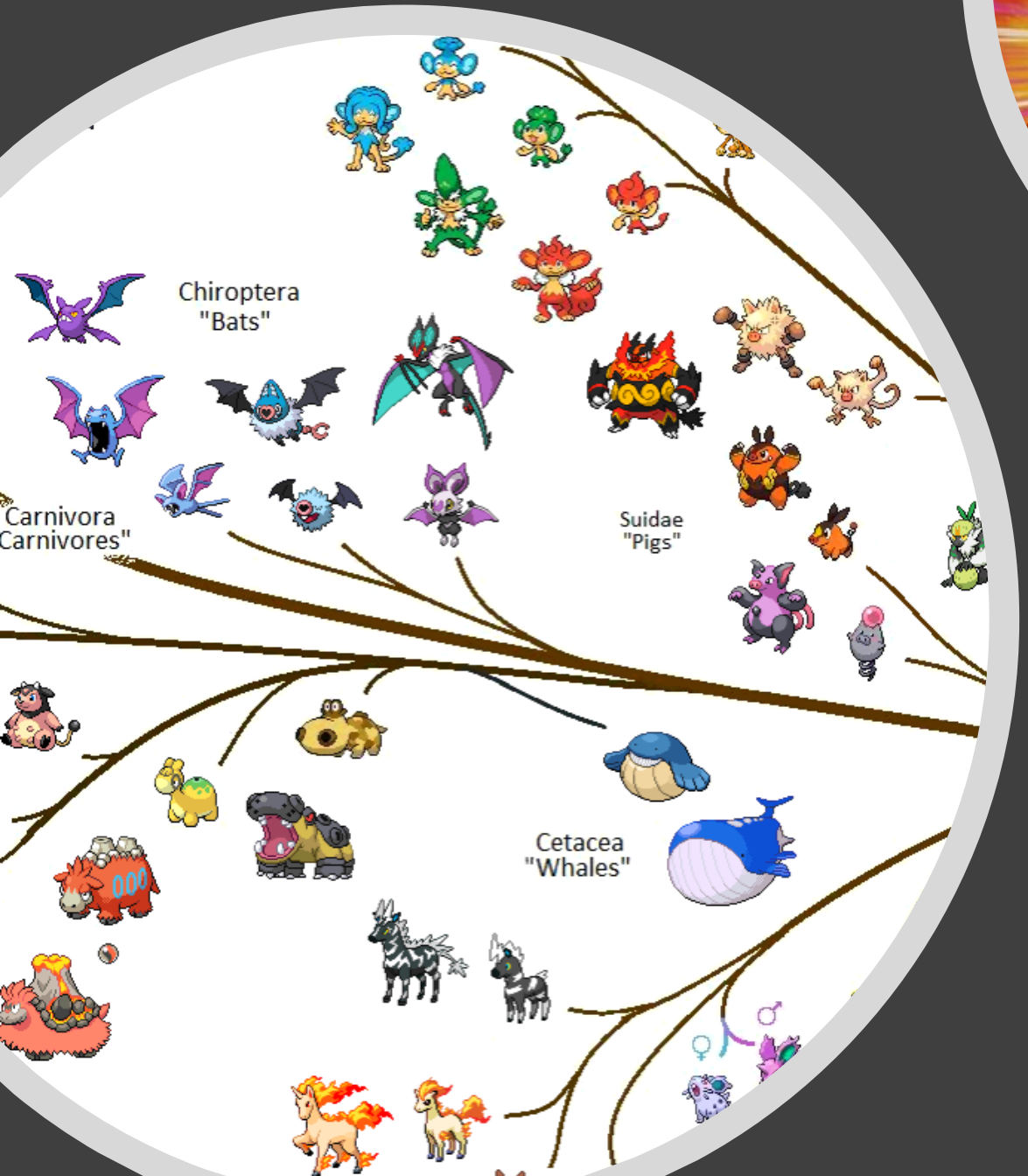
`commit` *vs* `revert`, `reset --hard`

Danger zone

- `git checkout`
- `git clean`
- `git reset --hard`

Cheats

Description	Command
Add to staging area in patch mode	`git add -p [ <file> ]`
Commit with message	`git commit -m "<messagge>"`
Amend commit (*for local repos*)	`git commit --amend`
Log on one line	`git log --oneline --graph`



Parallel development

Branching and merging

- With git it is possible to develop simultaneously on parallel environments, called **branches**, starting from a common starting point.
- Each branch will then have its own commit history and these environments could be reconciled one in the other, by performing an operation called **merge**.
- A special satus goes to the **master** branch which identifies the main branch which every other one should be merged into. This branch contains the agreed version of the software and it is the most accessed by developers therefore it should be the most stable and the most protected of the branches.

branch

- Branches are created based on another branch. The `branch` command takes an existing branch and creates a separate branch to work in.
- If no starting branch is specified the active one is considered.

```
marco@macbook ~/gittutorial> git branch test_branch
```

- To list local branches we simply call branch without any argument.

```
marco@macbook ~/gittutorial> git branch
* master
  new
```

- The active branch is indicated by the asterisk.

checkout

- To switch to the branch we've just created we use the checkout command.

```
marco@macbook ~/gittutorial> git checkout test_branch
```

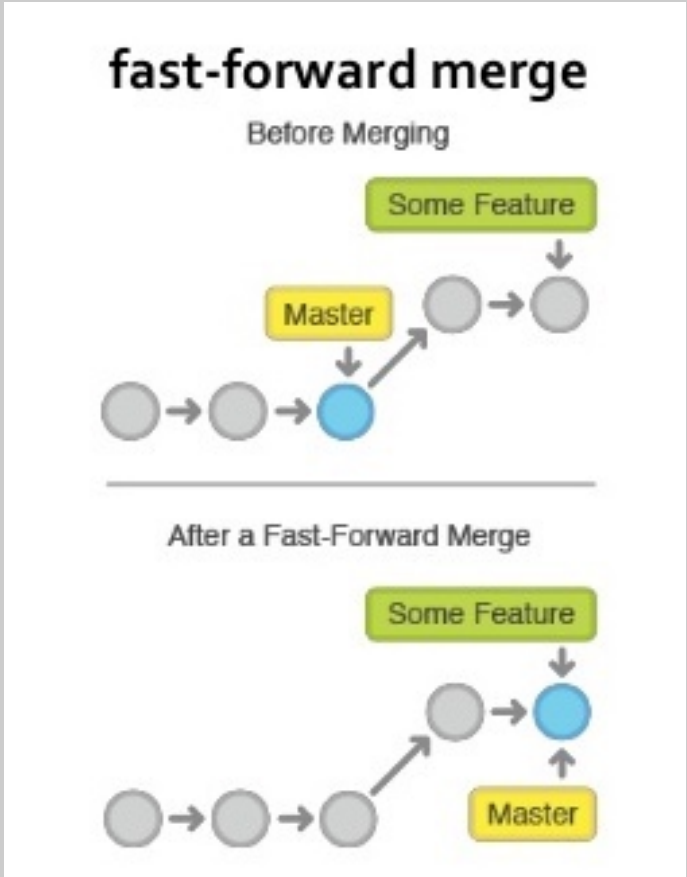
- Hence checkout can be used to navigate through the *branches* and through the *commits* inside the branches.
- Note that if we have changes in the staging area we won't be able to checkout to a different branch and we will have to undo the modifications or put them in a stash with the `stash` command.

merge

- Each branch has its own commit history, starting from the root point which is shared with the branch it was originated from.
- At a certain point we would like to include the modifications made on a branch into another branch. This operation is called **merge** and if underestimated can be the source of many migraines.

Fast-forward merge

- In the simplest case, the source branch which we started developing from hasn’t changed.
- In this case merging is like copying the history of a branch into the other, as if we had developed directly on the target branch.



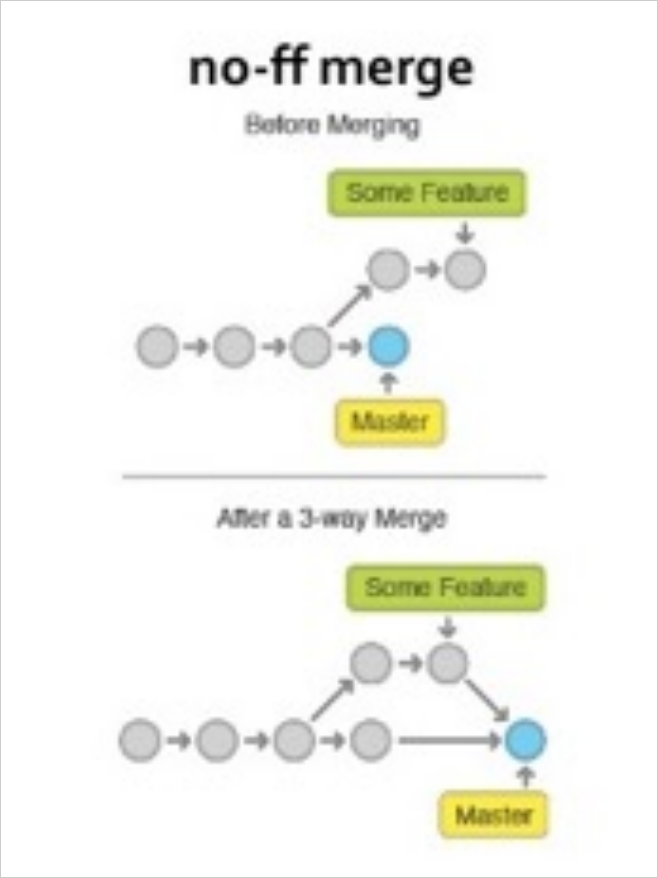
merge

- To perform a merge we put ourselves into the destination branch and we specify the branch we want to merge inside it with the `merge` command.
  - Note that if we have changes in the staging area we won't be able to checkout to a different branch and we will have to undo the modifications or put them in a stash with the stash command.

```
marco@macbook ~/gittutorial> git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
marco@macbook ~/gittutorial> git merge test_branch
Updating 515abc8..14d0c3b
Fast-forward
 test.py | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.py
```

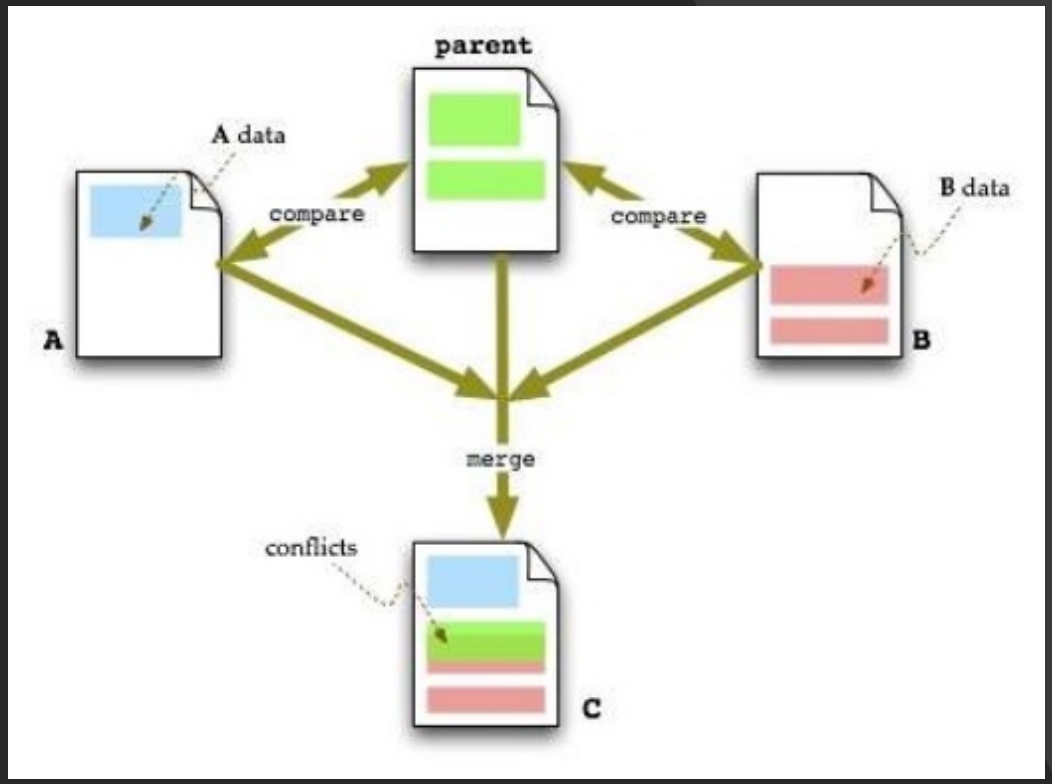
Non fast-forward merge

- Typically, however, when we start developing on a new branch, we'll continue to work parallely on the other branches.
- In this case when merging we will have to **compare** the modifications made on both branches and determine if the modifications conflict in any way.



3-way merge

- Say you and your friend both checked out a file, and made some changes to it and you need to merge his changes into your copy.
- If you were doing a **two-way merge**, the tool could compare the two files, and see that the files are different. But how would it know what to do with the differences?
- With a **three-way merge**, it can compare the two files, but it can also compare each of them against the **original** copy (before either of you changed it) and it can use that information to produce the merged version[3][4].

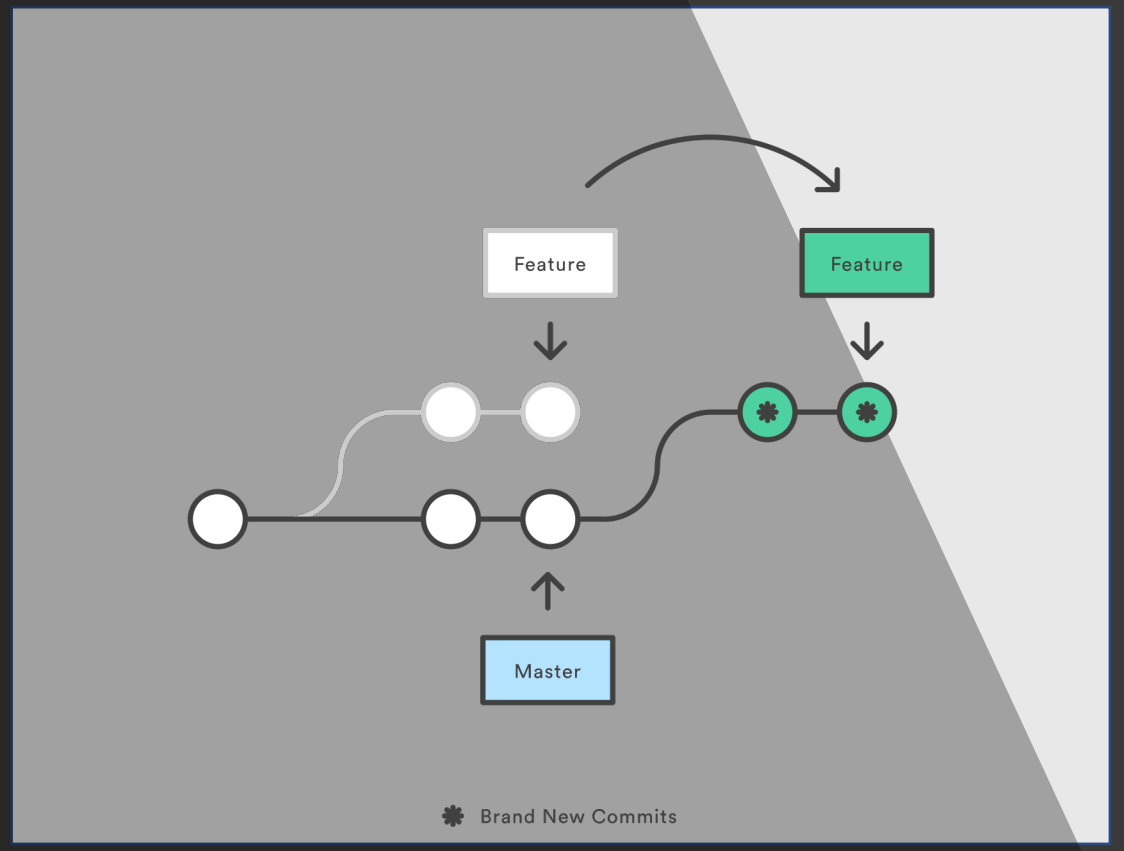


Conflicts

- When working simultaneously on code the main issue that can arise is that changes could overlap, creating **conflicts**.
- Conflicts cannot be resolved automatically and each one has to be carefully considered and resolved **manually**.

Rebasing

- It is usually recommended to maintain a **linear** project history which means recurring to fast-forward merge whenever possible.
- When developing a new feature this is rarely the case as the branch onto which we want to merge – tipically the master branch – has progressed in the meantime.
- To achieve this we use the `rebase` command, which moves the root of the branch to the HEAD of the branch.
- Eventual conflicts that may occur have to be solved when moving the starting point of the branch and once solved a fast forward merge will be performed.



rebase

```
marco@macbook ~/gittutorial> git log --oneline --graph --all
* f5eabe0 (test_branch) world.py added
| * 9cae9d0 (HEAD -> master) hello.py modified
|/
* be81027 hello.py added
marco@macbook ~/gittutorial> git checkout test_branch
Switched to branch 'test_branch'
marco@macbook ~/gittutorial> git rebase -i 9cae9d0
Successfully rebased and updated refs/heads/test_branch.
marco@macbook ~/gittutorial> git log --oneline --graph --all
* b3ef0a8 (HEAD -> test_branch) world.py added
* 9cae9d0 (master) hello.py modified
* be81027 hello.py added
```

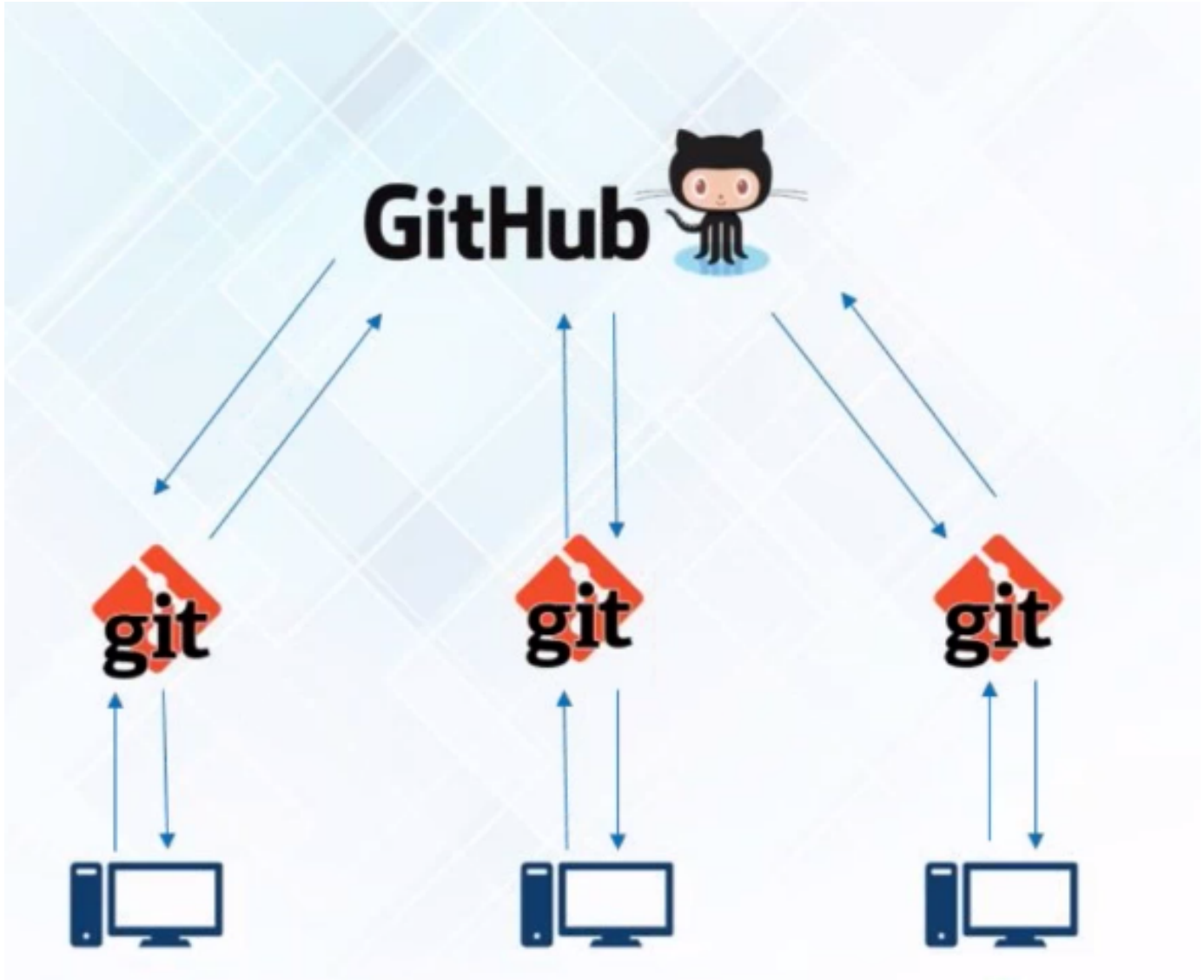
rebase

```
marco@macbook ~/gittutorial> git checkout master
Switched to branch 'master'
marco@macbook ~/gittutorial> git merge test_branch
Updating 9cae9d0..b3ef0a8
Fast-forward
 world.py | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 world.py
marco@macbook ~/gittutorial> git log --oneline --graph --all
* b3ef0a8 (HEAD -> master, test_branch) world.py added
* 9cae9d0 hello.py modified
* be81027 hello.py added
```

Cheats

Description	Command
Create new branch and checkout	`git checkout –b <branch-name>`
List local and remote branches	`git branch -a`
Include commit message at HEAD of each branch	`git branch -v`

Distributed



Local vs Remote repositories

- Git is a **distributed** version control system. This means that we have a central repository that can be copied on any number of local machines and modified locally. The modifications can then be re-sended to the central repository.
- The **local vs remote** repository distinction is fundamental. The remote repository will be considered the official version of the repository.
- Receiving and sending modifications to the remote repository are delicate operations, especially if done on the `master` branch.

Commit history

- The commit history on the central repository should never be altered locally otherwise the synchronization operations will fail.
- This is desidered, as we don’t want to corrupt the history of the project which is shared with the other users.

remote

- After creating a remote repository on a server (we'll skip this part), we can link our local repository to that one through the remote command.
- We can add, rename, remove multiple repositories, but the typical workflow generally involves just one repository.

```
marco@macbook ~/gittutorial> git remote add origin
git@gitlab.com:euclidea/gittutorial.git
```

push

- At this point we want to push our local data to the remote repository.
- Best practice is to checkout to the branch that we want to push and then call push.

```
marco@macbook ~/gittutorial> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 305 bytes | 305.00
KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gitlab.com:euclidea/gittutorial.git
   b3ef0a8..3f68758  master -> master
```

- If we have set multiple remote repositories we can also specify the repo to push on.

Pushing commits

- Note that before pushing our local commits, we should have already pulled all the remote commits on our local branch.
- If this is not the case and we try to push to the remote repository the operation will fail.
  - Solving the issue is a bit tricky as we have to create a new branch, restore the old branch to the last remote commit we have on disk, pull and merge the new branch.
  - That’s why it is recommended to work on separate branches when developing new features.

fetch

- The fetch command retrieves data from the remote repository and indexes it, enabling the possibility to merge it locally.
- Note that if we have a disalignment with the local commit history we will have to be careful when performing the merge.

```
marco@macbook ~/brave-browser> git fetch
remote: Enumerating objects: 142, done.
remote: Counting objects: 100% (142/142), done.
remote: Compressing objects: 100% (27/27), done.
remote: Total 208 (delta 130), reused 123 (delta 115), pack-reused 66
Receiving objects: 100% (208/208), 126.23 KiB | 839.00 KiB/s, done.
Resolving deltas: 100% (141/141), completed with 20 local objects.
From github.com:brave/brave-browser
   080985095..9ffe603ac  master -> origin/master
   6b1cd664e..05b38984a  1.1.x  -> origin/1.1.x
```

pull

- Tipically we want to fetch the data and perform a merge, especially if no modification has been made locally and we want to make the current branch progress.
- To perform this we can use the `pull` command which is equivalent to executing a `fetch` followed by a `merge origin/<branch>`.

```
marco@macbook ~/gittutorial> git pull
From gitlab.com:euclidea/gittutorial
 * branch            master     -> FETCH_HEAD
Updating 60d460d..567ab3f
Fast-forward
 Pipfile                    |   1 +
 Pipfile.lock               | 135 +++++++-------
2 files changed, 32 insertions(+), 28 deletions(-)
```

clone

- If a git repository is published on a server we can copy the project with its entire history with the `clone` command.

```
marco@macbook ~> git clone git@github.com:brave/brave-browser.git
Cloning into 'brave-browser'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 87512 (delta 5), reused 4 (delta 2), pack-reused 87503
Receiving objects: 100% (87512/87512), 135.76 MiB | 1.68 MiB/s, done.
Resolving deltas: 100% (61768/61768), done.
```

Recommendations

- Do not modify commit histories that have already been pushed.
  - `revert` is the best tool for undoing shared public changes.
  - `reset` is best used for undoing local private changes.
- Operate locally and push only at the end of the day.
- Make the commit history readable.
  - Parameters values shouldn’t be versioned.
  - Maintain a linear project history (prefer fast-forward merge, rebase whenever possible).
- Do not use git as a synchronization tool.

Useful resources

- https://www.atlassian.com/git/tutorials
- https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- https://git-scm.com/docs
- https://learning.oreilly.com/scenarios/

Credits

[1] https://en.wikipedia.org/wiki/Git

[2] https://en.wikipedia.org/wiki/Version_control

[3] https://stackoverflow.com/a/4129145