Sakli Leila

https://git-scm.com/doc

Plan

- Introduction
- Système de contrôle de version
- Git
- Installation et configuration de Git
- Création d’un dépôt

Introduction

Pendant la réalisation d'un projet informatique

Il y aura plusieurs versions de notre logiciel

On partage le code avec d'autres développeurs

On le modifie souvent

après une (petite) modification, plus rien ne fonctionne

Système de contrôle de version

- Un système de contrôle de version est un outil de développement
- Un logiciel qui aide une équipe de développement à gérer les changements apportés au code source au fil du temps.
- Ce système garde une trace de chaque changement apporté au code.
- Localiser la source d’une éventuelle erreur et, par conséquence minimise les perturbations pour tous les membres de l'équipe.

Système de contrôle de version

Trois modèles de versioning (VCS : Version Control System)

Modèle local : une base de données simple pour conserver les modifications d'un fichier.

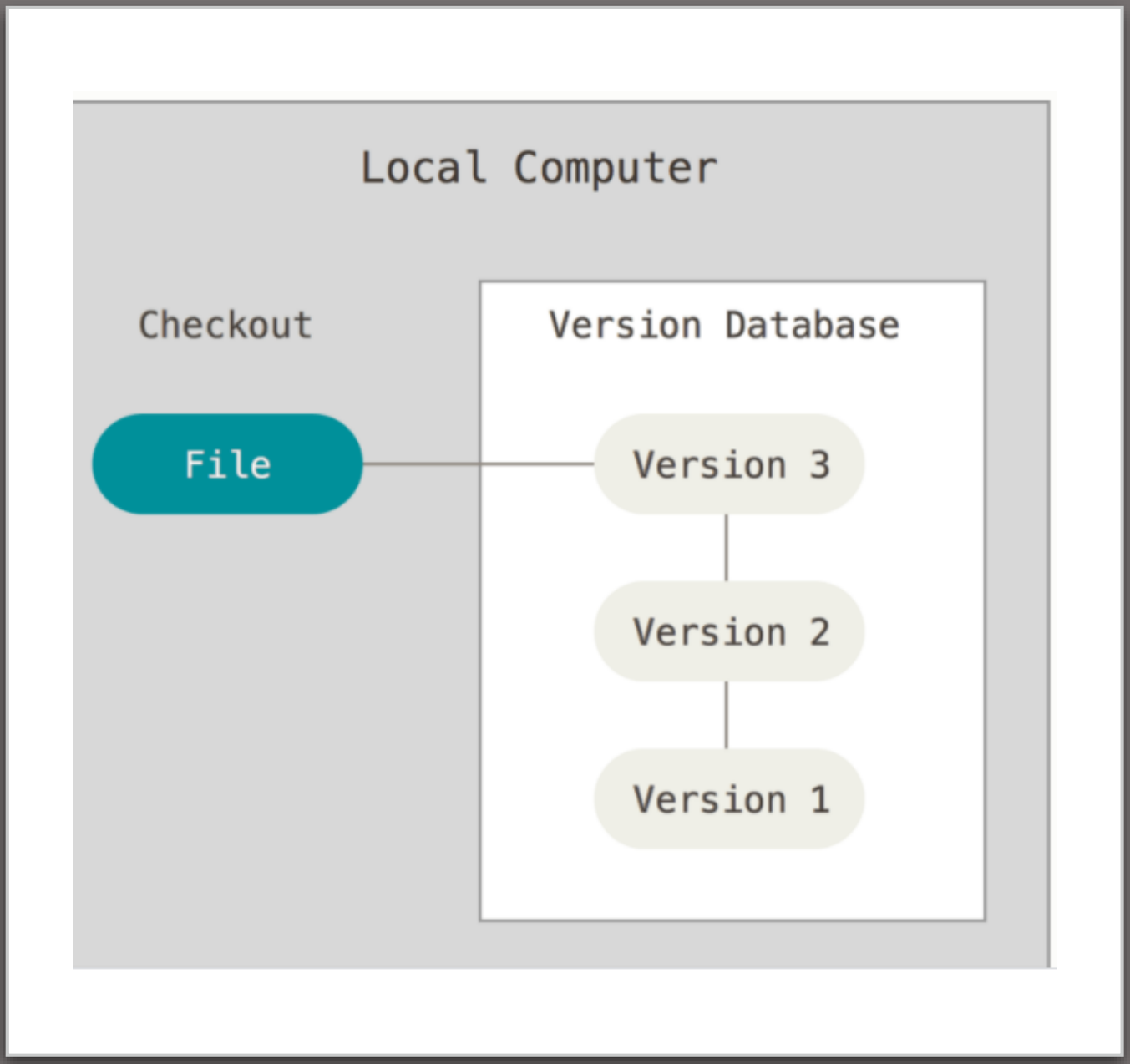
Exemple: RCS

Modèle centralisé : le code du logiciel est géré par un serveur central

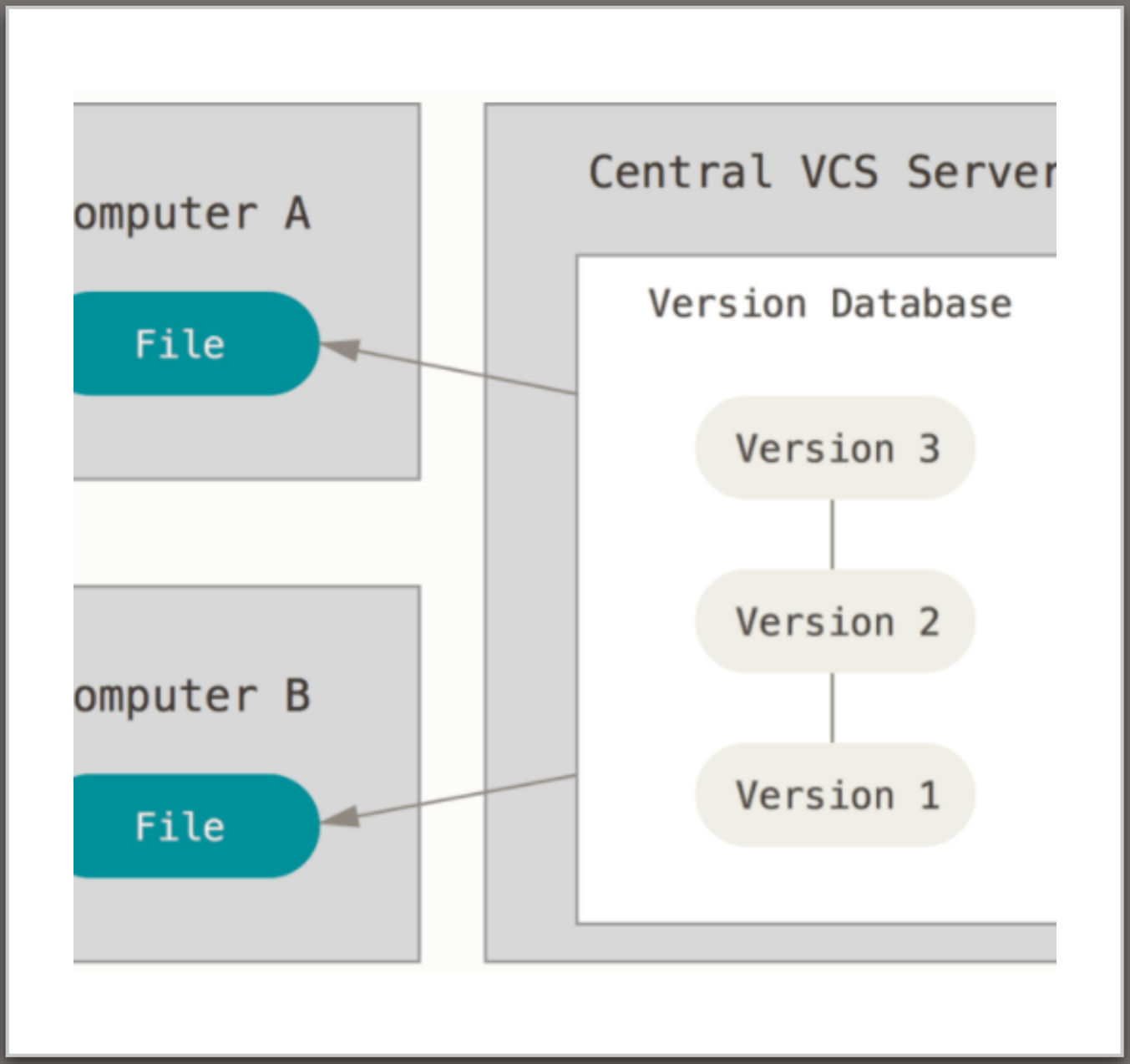
Exemple : SVN, CVS

Modèle distribué : tous les développeurs ont accès au code sans passer par un serveur

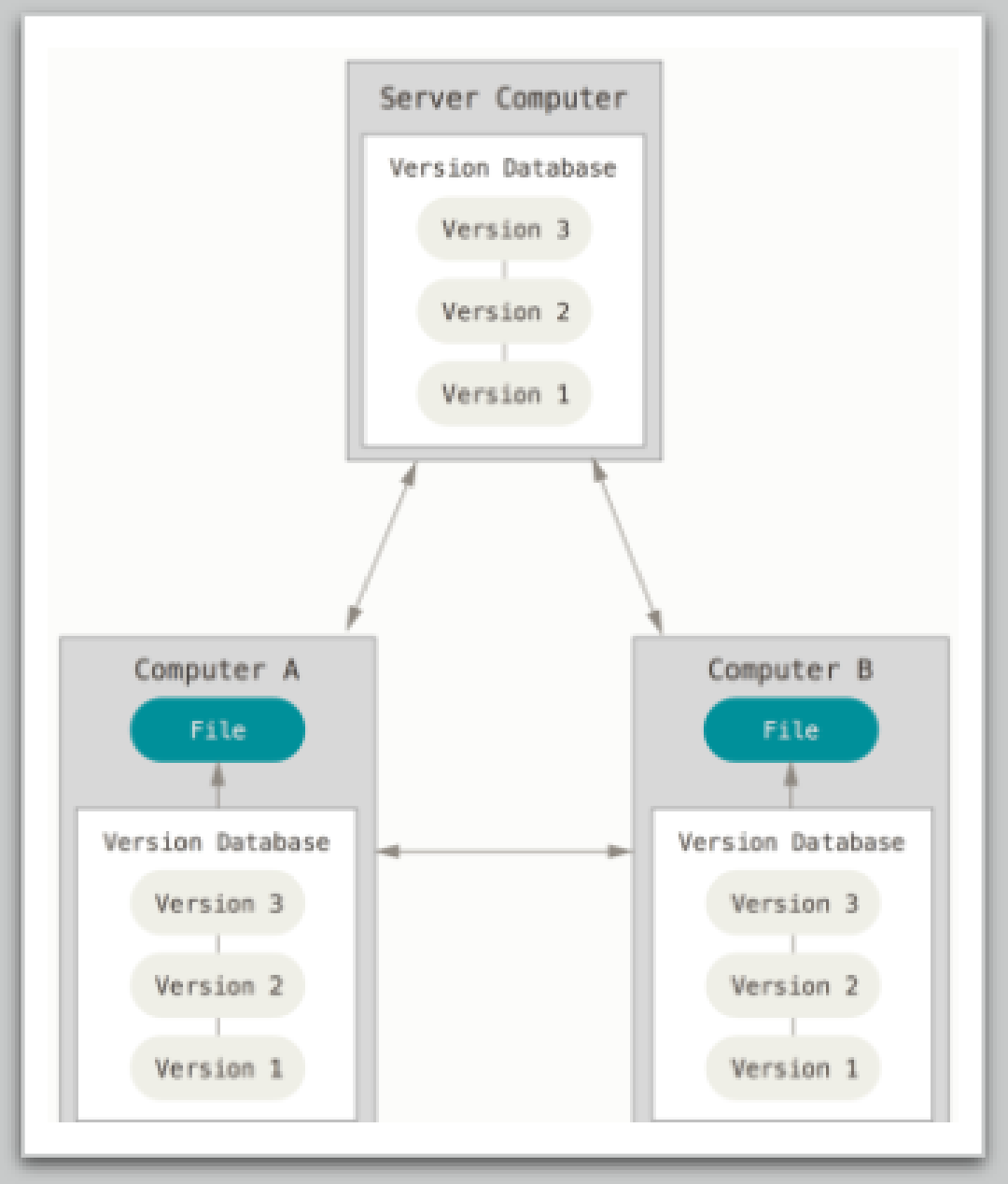
Exemple : Git, Mercurial, Bazaar, Darcs



Modèle Local



Modèle centralisé



Modèle distribué

Pourquoi un VCS:

Permet aux développeurs de collaborer

- Combinez le travail de plusieurs collaborateurs
- Ne permet pas d'écraser les modifications de l'autre.
- Maintient un historique de chaque version.
- Comprendre les changements
- Soutenir le développement incrémentiel
- Comparer et revenir aux versions antérieures
- Sauvegarde
- Développement de documents (pour les autres développeurs et vous-même, pas pour les utilisateurs)

Git

- Logiciel de gestion de version créé en Avril 2005 par Linus Torvald : fondateur du Kernel Linux
- Mai 2013 : 36% des professionnels utilisent Git comme logiciel de versioning
- Avril 2013 : Github déclare avoir 3.5 millions d'utilisateurs
- Mai 2019 : Github déclare avoir 37 millions d'utilisateurs et plus de 100 millions de dépôts (repositories)
- Mai 2018, Github est acheté par Microsoft

Git

- En quelques mots
- Syntaxe proche de Shell Linux
- Basée sur des commit (une version valide du code)
- Permettant de retrouver un fichier supprimé, une ancienne version modifiée...
- Acceptant toute extension de fichier (JS, Java, PHP, ASP...) Utiliseé par des sites web de partage
- Github : https://github.com/ BitBucket : https://bitbucket.org/
- Gestion de projet de taille importante

Installation et configuration de Git

L' installation dépend du système d'exploitation

- Sous Unix
  - Sur Fedora (ou toute distribution parente basée sur RPM), vous pouvez utiliser dnf :

    ```
    $ sudo dnf install git-all
    ```
  - Sur une distribution basée sur Debian, telle que Ubuntu

    ```
    $ sudo apt install git-all
    ```

    Pour plus d'options, des instructions d'installation sur différentes versions Unix son disponibles sur le site web de Git, à http://git-scm.com/download/linux.

Installation et configuration de Git

- Sous MAC : via homebrew (brew install git)
- Sous Windows : le nouveau lien depuis l'achat par Windows https://gitforwindows.org/
- Pour les informations sur la configuration suivez ce lien

https://git-scm.com/book/fr/v2/Personnalisation-de-Git-Configuration-de-Git

Niveaux de configuration

- git config : un outil **Git** pour voir et modifier les variables de configuration. Ces variables peuvent être stockées dans trois endroits différents :
- [chemin]/etc/gitconfig : Contient les valeurs pour tous les utilisateurs et tous les dépôts du système.
- Fichier ~/.gitconfig : Spécifique à votre utilisateur.
- Fichier config dans le répertoire Git (c'est-à-dire .git/config) du dépôt en cours d'utilisation : spécifique au seul dépôt en cours.

Configuration de la console

Pour désactiver la coloration dans la console (par défaut activée)

```
$ git config --global color.ui false
```

Pour désactiver la coloration dans la console (par défaut activé)

```
$ git config --global color.diff auto
$ git config --global color.status auto
$ git config --global color.branch auto
$ git config --global color.interactive auto
```

Configuration identité

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Pour vérifier la valeur d'une propriété de configuration

```
$ git config user.name
```

De nombreux outils graphiques vous aideront à le faire la première fois que vous les lancerez.

Configuration d’éditeur du texte

Par défaut, Git utilise l’éditeur configuré au niveau système, qui est généralement Vi ou Vim qui sera utilisé quand Git vous demande de saisir un message.

Pour configurer l’utilisation d’un autre éditeur de texte, comme Emacs, vous pouvez entrer ce qui suit :

```
$ git config --global core.editor emacs
```

Sur un système Windows system, si vous souhaitez utiliser un autre éditeur de texte, vous devez spécifier le chemin complet de son fichier exécutable.

```
$ git config --global core.editor " ’C:/Windows/system32/notepad.exe' -multiInst -notabbar -nosession -noPlugin"
```

Configuration d’éditeur du texte

Ajouter Sublime Text 3 aux variables d’environnement de Windows

- Dans la zone de recherche de Windows 10, chercher Système
- Aller dans Paramètres système avancés et choisir Variables d’environnement
- Dans Variables système, sélectionner PATH puis cliquer sur Modifier
- Cliquer sur Nouveau, ajouter le chemin vers le dossier d’installation de Sublime Text 3 (C:\Program Files\SublimeText 3)

Création d'un dépôt

- Les commandes Unix sont toujours valables :
- pwd : imprimer le chemin d'accès
- ls : lister le contenu du répertoire courant
- cd : changer de répertoire
- mkdir : créer un répertoire
- rm : supprimer un répertoire ou un fichier
- touch : créer un fichier
- echo : écrire dans la console ou dans un fichier
- head : afficher le contenu d'un fichier dans la console
- ...

Création d'un dépôt

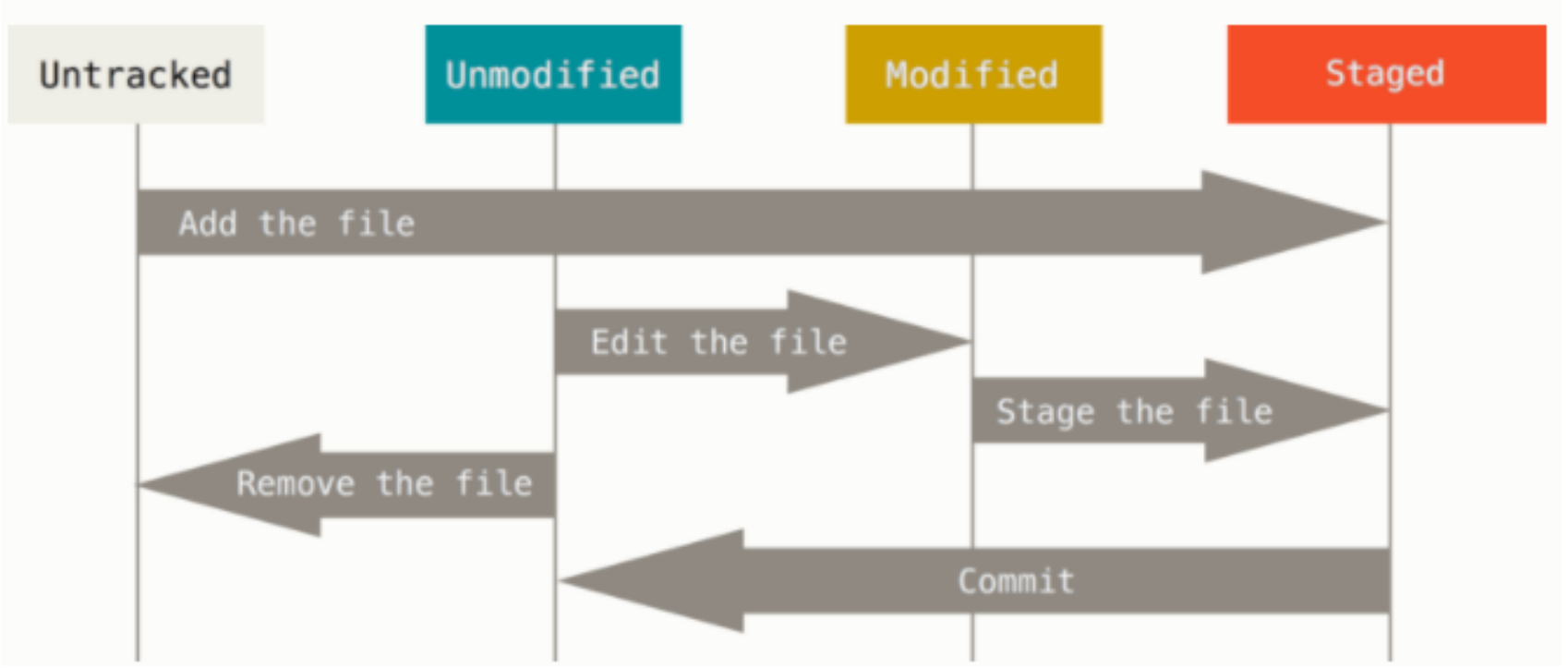
- Un dépôt (repository) est un répertoire de travail géré par Git contenant les éléments à partager ou à « commiter »
- Pour créer un dépôt:

```
$ mkdir firstGit //créer un répertoire

$ cd firstGit //se positionner dedans

$ git init//déclarer ce répertoire comme un dépôt
```

Trois états d’un fichier



Premier Commit

La commande qui vérifie l'état d'un fichier est :

```
$ git status
```

Pour indexer (ajouter le fichier au Staging Area) un fichier : `$ git add NameFile`

La commande qui permet de valider les fichiers modifiés et indexés : `$ git commit`

Exercice: First commit

1. Vérifier le contenu de dépôt (`$ git status`)
2. Créer un fichier `file.txt` (`$ touch file.txt`)
3. Revérifier le contenu de dépôt
4. Modifier le contenue du fichier file.txt
5. Revérifier le contenu de dépôt

Exercice: First commit

6. Indexer le fichier file.txt (`$ git add file.txt`)

(ou bien `$ git add .` **Ou encore** `$ git add –all`**)**

7. Revérifier le contenu de dépôt

Remarque on peut utiliser

`$ git rm --cached <fichier>...` pour désindexer

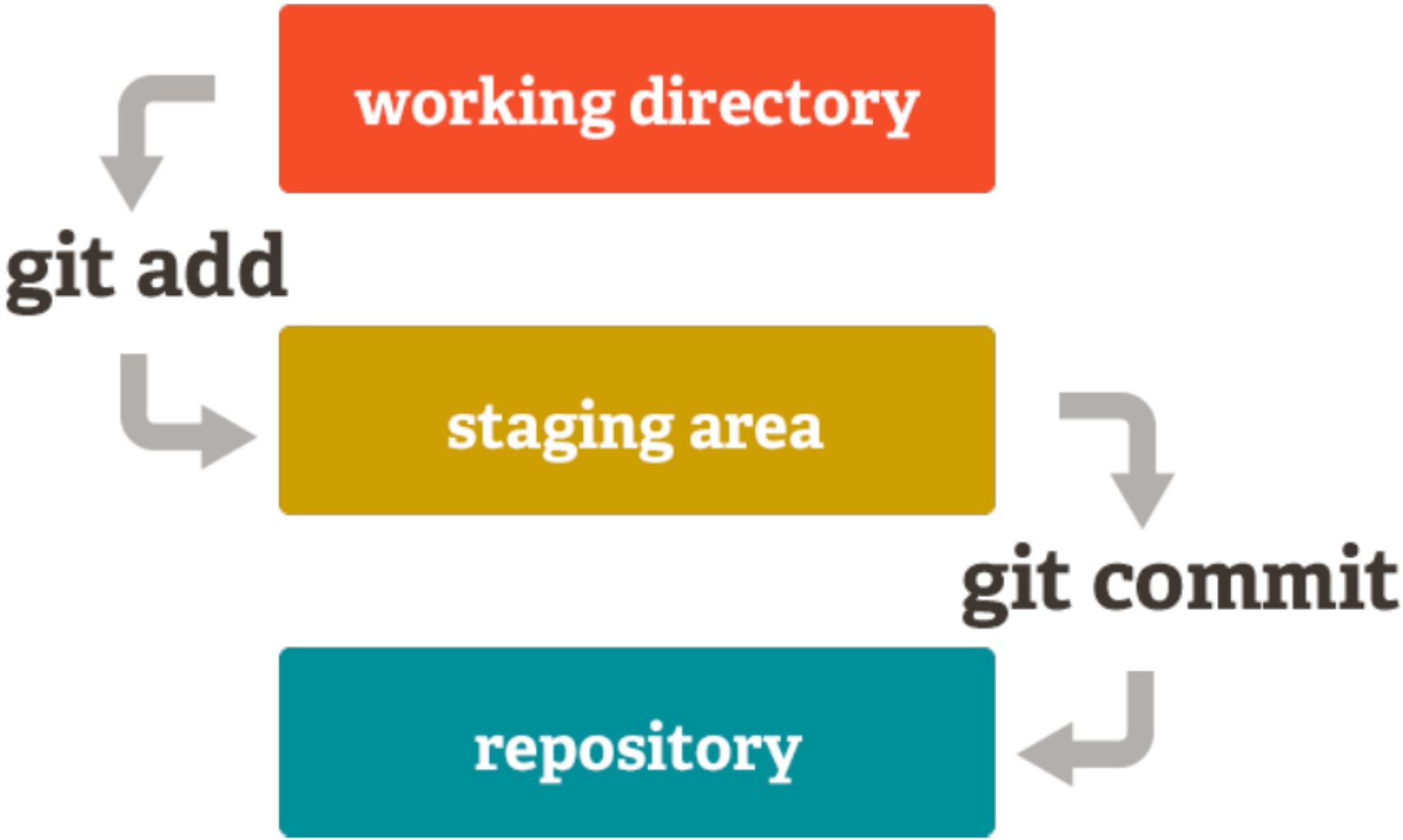
Exercice: First commit

8. Valider le fichier

```
$ git commit -m “first commit”
```

7. Revérifier le contenu de dépôt

```
$ git status
```



https://git-scm.com

Exercice: nième commit

Deux façons de faire:

Soit refaire les deux étapes de l'exercice précédent

Soit fusionner les deux étapes

1. Modifier le contenue du fichier file.txt
2. Vérifier le statut du fichier
3. Faire un commit (`$ git commit -am "second commit"`)
4. Vérifier le statut du fichier

Afficher la liste de Commit

La commande qui permet d'afficher l'historique des modifications

```
$ git log
```

Cet affichage peut être en mono-ligne

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

Pour un affichage mono-ligne mais avec un identifiant complet

```
$ git log --pretty=oneline
```

Pour un afficher les Commit sous forme d'un graphe

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

Connaitre la différence entre 2 versions

Les points de différence entre deux Commit

```
$ git diff idCommit1 idCommit2
```

La différence d'un Commit avec le staging area

```
$git diff idCommit1
```

Connaitre la différence entre 2 versions

Exercice:

1. Modifier le contenu de file.txt
2. Vérifier la différence d’un Commit avec le staging area

Naviguer entre les Commit

- Aller sur un autre Commit

```
$ git checkout idCommit
```

- Pour pointer sur le dernier commit sans préciser son identifiant

```
$ git checkout master
```

Modifier le message du dernier Commit

Modifier le message du dernier Commit

- En utilisant l'argument m

```
$ git commit --amend -m "second commit"
```

- Sans utiliser l'argument m

```
$ git commit –amend
```

Dans vi

Saisir i pour modifier le message

Pour terminer, cliquer sur echap puis saisir :wq

Valider en cliquant sur Entree

Exercice : Annuler un Commit

1. Créer un deuxième fichier file2.txt et faire un troisième Commit avec le message creating file2.txt
2. Ajouter une troisième ligne third dans file.txt et faire un cinquième Commit avec le message third
3. Ajouter une quatrième ligne fourth dans file.txt et faire un cinquième Commit avec le message fourth

Exercice : Annuler un Commit

1. Vérifier les nouvelles modifications

```
$ git log –oneline
```

5. Annuler le commit ayant comme message creating file2

```
$ git revert idCommit
```

6. Vérifier l'annulation

Supprimer des modifications

- Pour supprimer une modification on a trois possibilités
  - Annuler le commit et garder les modifications dans le working directory(mode **mixed** : par défaut)
  - Annuler le commit et garder les modifications dans le staging area(mode **soft**)
  - Annuler le commit et ne pas garder les modifications (mode **hard**)

Supprimer des modifications

- Syntaxe

```
$ git reset --mode idCommit
```

- Exemple

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

- Explication

Tous les Commit réalisés après le commit ayant comme identifiant idCommit seront et impossible de les récupérer.

En faisant git status, il n'y a rien à indexer ni à valider.

Les tags

- Les tags c'est quoi?

C'est une étiquette qui permet de marquer un Commit/une version de notre application et fait donc référence vers un Commit

- Les tags c'est pourquoi?
    - Pour accéder à un commit qui présente une version importante de notre projet
    - Eviter de chercher le commit en question en lisant les messages de tous les Commit, pour faire git checkout

Les tags

- Syntaxe de création d'un tag sur le Commit actuel

```
$ git tag -a nom-tag -m "message"
```

- Exemple

```
$ git tag -a v0 -m "premiere version du projet"
```

- Syntaxe de création d'un tag sur un commit en utilisant son identifiant

```
$ git tag -a nom-tag idCommit -m "message"
```

- Exemple

```
$ git tag -a v0 -m "premiere version du projet"
```

Les tags

- On peut aussi se positionner sur un tag

```
$ git checkout nom-tag
```

- Pour lister les tags

```
$ git tag --list
```

- Exemple

```
$ git tag nom-tag --delete
```

Les branches

- La branche master
  - branche principale
  - contenant seulement des Commit représentant les différentes versions de notre application
  - Comment faire alors ? ⇒ Créer des branches et les utiliser
- Une branche, c'est quoi ?
  - déviation par rapport à la branche principale pointeur sur le dernier Commit
  - permettant de développer une nouvelle fonctionnalité, préparer une correction

Les branches

- Pour créer une branche

```
$ git branch nom-branche
```

- Changer de branche

```
$ git checkout nom-branche
```

- Créer et changer de branche

```
$ git checkout -b nom-branche
```

Les branches

- Remarque 1

En créant une branche, cette dernière pointe sur le commit à partir duquel elle a été créée

- Remarque 2

En faisant un Commit à partir de la branche créée, cette dernière dévie de la branche principale

Les branches

- Pour lister les branches locales

```
$ git branch --list
```

- Ou tout simplement

```
$ git branch
```

- Pour lister les branches (avec l'identifiant du dernier commit de chaque branche)

```
$ git branch -v
```

Les branches

- Pour lister les branches distantes

```
$ git branch -r
```

- Ou aussi

```
$ git branch --all
```

- Pour supprimer une branche vide (ou fusionnée)

```
$ git branch -d nom-branche
```

- Pour forcer la suppression d'une branche

```
$ git branch -D nom-branche
```

La fusion

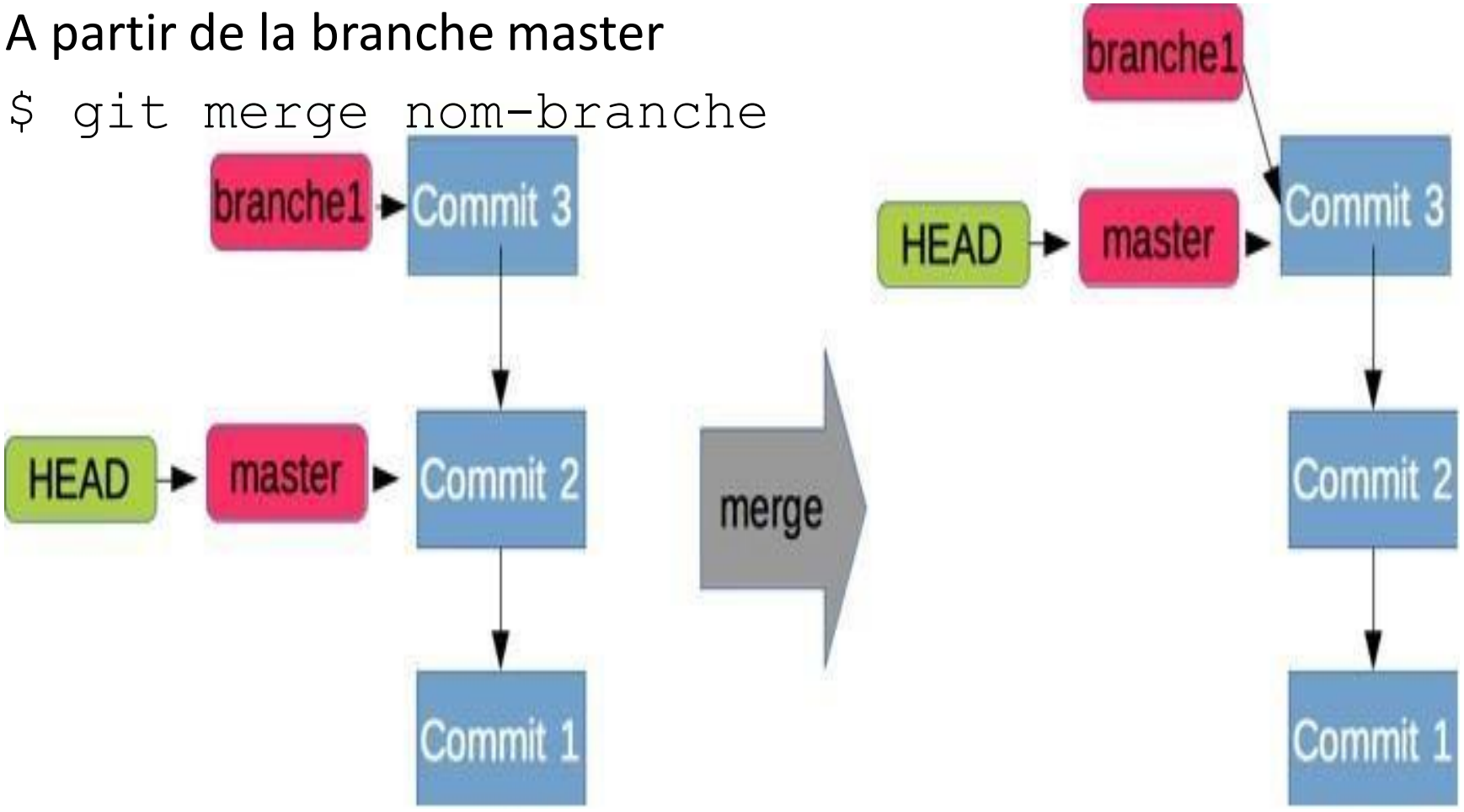
Fusion : deux cas possibles

1. sans conflit
   - fast forward : sans commit de fusion
   - non fast forward (avec l'option --no-ff) : avec un commit de merge
2. avec conflit : avec un commit de merge

La fusion

A partir de la branche master

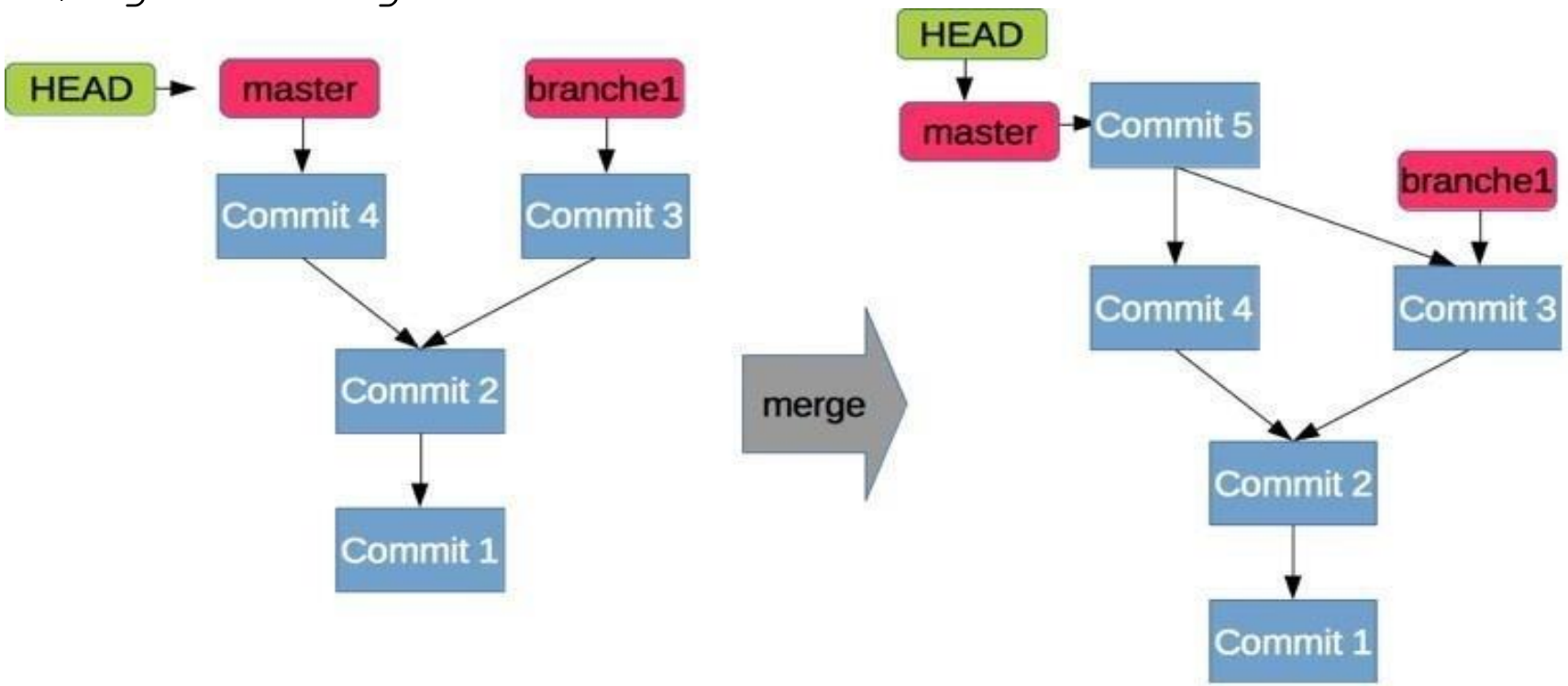
```
$ git merge nom-branche
```



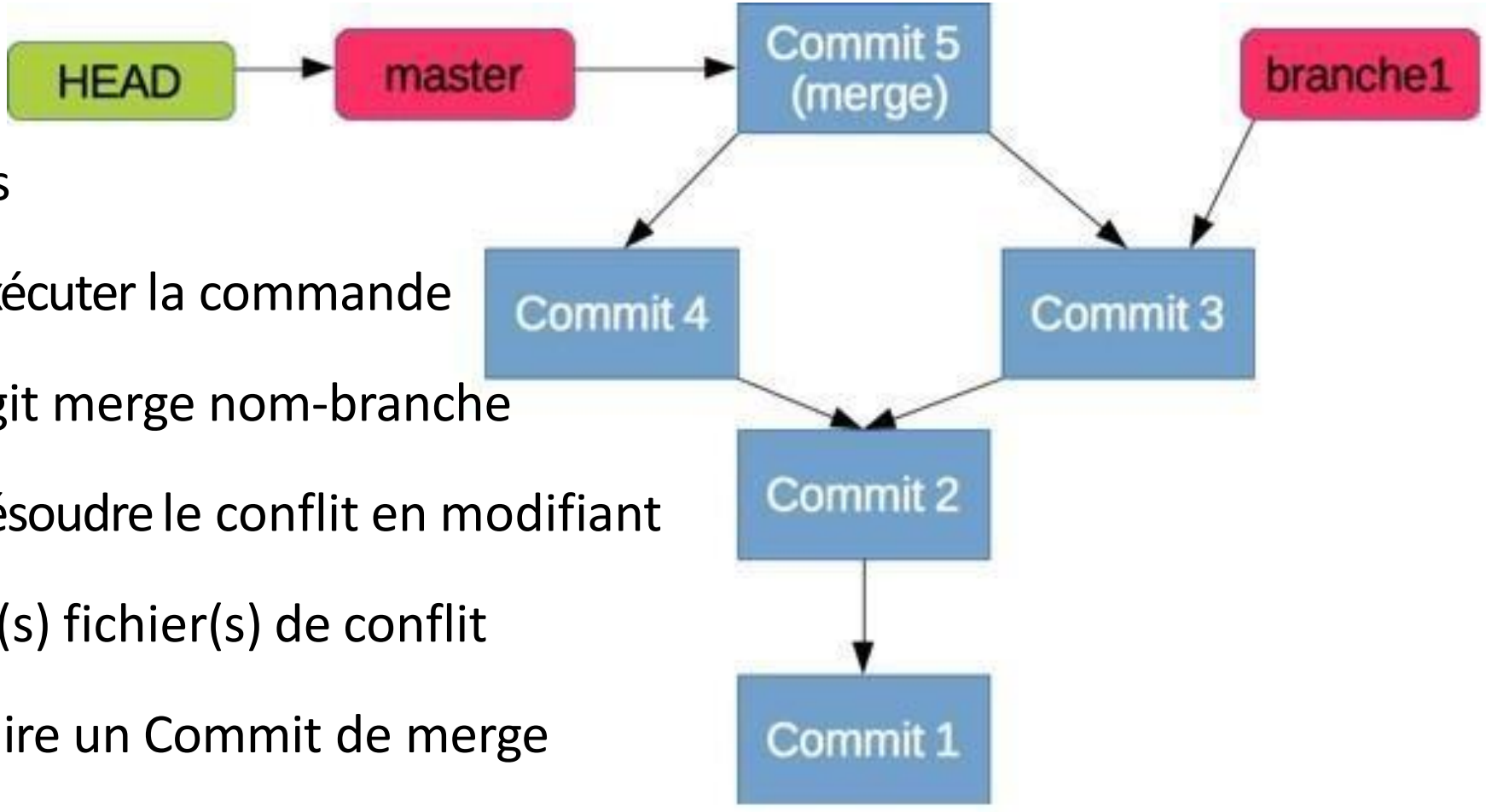
La fusion

Il faut ajouter l'argument `--no-ff`

```
$ git merge --no-ff nom-branche
```



La fusion



Etapes

1. Exécuter la commande
2. $git merge nom-branche
3. Résoudre le conflit en modifiant le(s) fichier(s) de conflit
4. Faire un Commit de merge

Exercice 1 : La fusion

1. Créer un nouveau repository Git
2. Ajouter un fichier et le commiter (C1)
3. Créer une branche (B1) à partir de C1
4. Faire un checkout sur B1
5. Modifier le fichier et faire un Commit (C2)
6. Merge B1 dans master de manière à avoir un Commit de merge dans master

Exercice 2 : La fusion

1. Créer un nouveau repository Git
2. Ajouter un fichier et le commiter (C1)
3. Modifier le fichier et le commiter (C2)
4. Créer une branche (B1) à partir de C1
5. Faire un checkout sur B1
6. Modifier le fichier et faire un Commit (C3) Merge B1 dans master en résolvant le conflit

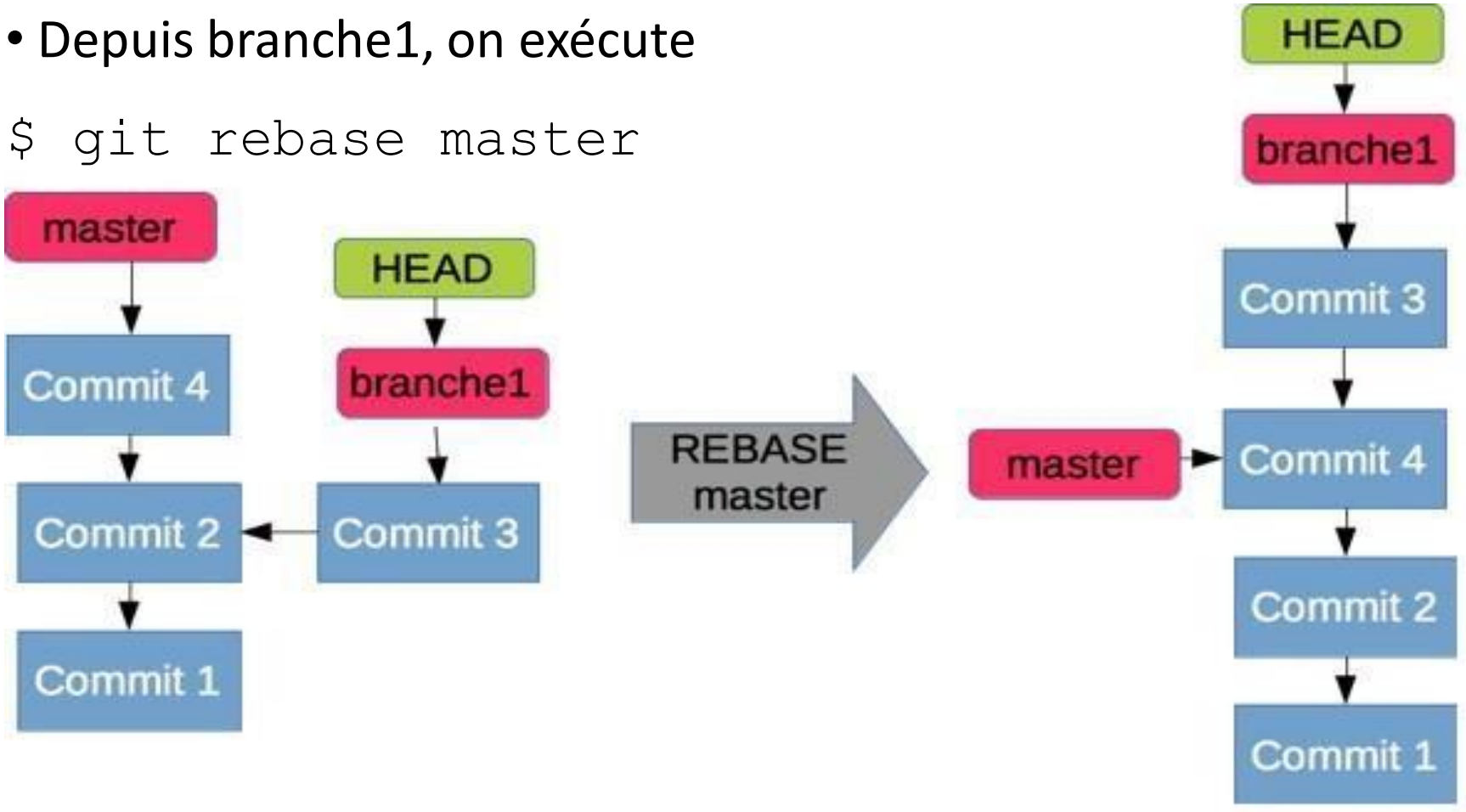
Rebase

- Problème de la fusion
  - Des cycles, des fois, inutiles
  - Des Commit de merge non nécessaires
- Le rebase, permet de
  - manipuler l'historique en réécrivant le passé
  - linéariser le graphe de commit en évitant les Commit de merge et en fusionnant les Commit d'une même branche dans un seul Commit

Rebase

- Depuis branche1, on exécute

```
$ git rebase master
```



Rebase

- Remarque
- master, branche principale, décalée par rapport à branche1

- **Solution : fast forward**

```
$ git merge branche1
```

Exercice: Rebase

1. Créer un nouveau repository Git
2. Ajouter un fichier et le commiter (C1)
3. Modifier le fichier et le commiter (C2)
4. Créer une branche (B1) à partir de C1 Faire un checkout sur B1
5. Créer un nouveau fichier et faire un Commit (C3)
6. Merge B1 dans master de manière à avoir un historique linéaire

Le rebase interactif

- inverser l'ordre de deux ou plusieurs Commit modifier le message d'un Commit
- supprimer un Commit
- fusionner plusieurs Commit en un seul

```
$ git rebase -i idCommit
```

Le rebase : fusionner plusieurs Commit

créer un fichier c.txt et faire un Commit

créer un fichier d.txt et faire un Commit

créer un fichier e.txt et faire un Commit

- En faisant $ git log --oneline

e8058fc (HEAD -> master) commit e

ff2c409 commit d

5960613 commit c

Le rebase : fusionner plusieurs Commit

- Pour fusionner les trois derniers Commit

```
$ git rebase -i HEAD~3
```

- Les trois premières lignes affichées

```
pick 5960613 commit c
pick ff2c409 commit d
pick e8058fc commit e
```

- Dans **vim** remplacer `pick` de deux derniers Commit par `squash`

Rebase : supprimer des Commit

créer un fichier f.txt et faire un Commit

créer un fichier g.txt et faire un Commit

créer un fichier h.txt et faire un Commit

- En faisant `$ git log --oneline`

```
e8058fc (HEAD -> master)
commit h
ff2c409 commit g
5960613 commit f
```

Rebase : supprimer des Commit

- Pour supprimer des Commit

```
$ git rebase -i HEAD~3
```

- Les trois premières lignes affichées

```
pick 5960613 commit f
pick ff2c409 commit g
pick e8058fc commit h
```

- Dans vim remplacer pick par drop

Rebase : renommer un Commit

Créer un fichier i.txt et faire un Commit

En faisant `$ git log –oneline`

```
e8058fc (HEAD -> master) commit i
```

- Pour renommer un Commit

```
$ git rebase -i HEAD~1
```

- La première ligne affichée : pick e8058fc commit i
- Dans vim remplacer `pick` par `reword` et modifier le message

Rebase : inverser des Commit

créer un fichier j.txt et faire un Commit

créer un fichier k.txt et faire un Commit

créer un fichier l.txt et faire un Commit

En faisant `$ git log --oneline`

```
e8058fc (HEAD -> master) commit l
ff2c409 commit k
5960613 commit j
```

Rebase : inverser des Commit

- Pour inverser des Commit

```
$ git rebase -i HEAD3
```

Les trois premières lignes affichées

```
pick 5960613 commit j
pick ff2c409 commit k
pick e8058fc commit l
```

- Dans vim placer le curseur au début du Commit à déplacer puis cliquer sur dd pour le couper cliquer sur p pour coller le Commit copié

La planque (stash)

- Problématique
- Des travaux non-finis (qu'on ne peut valider)
- Nécessité de publier des travaux validés (d'une autre branche)
- Impossible de changer de branche sans supprimer les modifications courantes ⇒ Solution : utiliser la planque pour mettre les travaux encours de coté

La planque (stash)

- **Exemple : sur la branche** `master`

```
$ echo bonjour > fichier1.txt
# crée et ajoute bonjour dans
fichier.txt
$ echo bonsoir > fichier2.txt
$ git add . # indexer les deux fichiers
$ git commit –m "first commit"
$ git checkout -b ma-branche
```

La planque (stash)

- **Exemple : sur la branche** `ma-branche`

```
$ echo hello >> fichier1.txt
$ git commit -am "second commit
for fichier1"
$ echo ciao >> fichier1.txt #
```

La planque (stash)

- Impossible de changer de branche car on n’a pas validé les dernières modifications

```
$ git checkout master
```

```
error: Your local changes to the following files would
be overwritten  by checkout:
        file1.txt
Please commit your changes or stash them before you
switch branches.  Aborting
```

La planque (stash)

- Pour ajouter un (ou plusieurs) fichier(s) a` la planque

```
$ git stash save
```

- Si le fichier modifié se trouve dans le `staging area`

```
$ git stash push
```

- Pour afficher le contenu de la planque

```
$ git stash list
```

La planque (stash)

- Pour récupérer un fichier de la planque

```
$ git stash apply
```

- Mais une copie de ce fichier est toujours dans la planque, vérifier

```
$ git stash list
```

- Pour vider la planque

```
$ git stash drop
```

- Pour récupérer un fichier de la planque sans qu'une copie y reste

```
$ git stash pop
```

Recherche

- Pour chercher un mot dans le dépôtˆ t

```
$ git grep "mot"
```

- Pour afficher le numéro de la ligne dans le fichier où le mot se trouve

```
$ git grep -n "mot"
```

Le fichier `.gitignore`

- Si on a un (ou plusieurs) fichier(s) (de configuration par exemple) qu'on ne voit aucun intérêt de les valider
- On peut les citer dans un fichier de configuration appelé .gitignore
- Un nom par ligne
- Ce fichier peut être indexé et validé

Le fichier `.gitignore`

- Exemple

```
$ echo info.txt>>.gitignore
$ echo *.html >> .gitignore
$ echo view/* >> .gitignore
$ echo java >> info.txt
```

- Explication
- En faisant git status, aucun fichier à indexer à l'exception de .gitignore
- Tous les fichiers avec l'extension html sont ignorés
- Aussi, tous les fichiers du répertoire view

L'historique du pointeur HEAD

- Pour connaître le journal du pointeur HEAD

```
$ git reflog
```

- Pour avoir un peu plus de détails

```
$ git log -g
```

DEPÔT nu

- dépôt sans espace de travail (en lecture seule) pas de Commit possible
- une sorte de serveur pour tous les utilisateurs
- Se placer dans le parent du dépôt courant

```
$ cd ..
```

- Créer un répertoire

```
$ mkdir firstGitBare
```

- Se placer dans ce répertoire

```
$ cd firstGitBare
```

- Créer un dépôt nu

```
$ git init --bare
```

Dépôt distant

- C’est un dépôt nu sur site hébergeur (GitHub, Bitbucket...)...

- A partir de notre premier dépôt (firstGit), créer un premier dépôt distant

```
$ git remote add origin c:/cheminVers/firstGitBare
```

- Afficher la liste des dépôts distants

```
$ git remote
```

- Afficher les branches distantes

```
$ git branch –r # aucune
```

Dépôt distant

- Envoyer (publier) la branche master sur le dépôt distant

```
$ git push origin master
```

- Afficher les branches distantes

```
$ git branch -r
```

- Ou aussi

```
$ git branch -a
```

Dépôt distant

- Supprimer une branche distante

```
$ git push origin --delete nomBranche
```

- Vérifier la réception de la branche a` partir du firstGitBare

```
$ git branch # *master
```

- Afficher tous les Commit

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

Dépôt distant

- Pour supprimer un remote

```
$ git remote remove nomRemote
```

- Pour renommer un remote

```
$ git remote rename oldName newname
```

Cloner un dépôt

- Se placer dans le parent du dépôt courant

```
$ cd ..
```

- Cloner le depot firstGit dans firstGitClone

```
$ git clone
    c:/cheminVers/firstGitBare
firstGitClone
```

- Se placer dans le répertoire cloné

```
$ cd firstGitClone
```

- Vérifier le dépôt distant

```
$ git remote -v
```

- Vérifier les Commit

```
git log --oneline
```

Cloner un dépôt

- Préparer un nouveau Commit

```
$ echo bonjour >>
new.txt  git add .
```

```
$ git commit -m "adding
new.txt"
```

- Publier le commit dans firstGitBare

```
$ git push origin master
```

- Vérifier cela depuis firstGitBare

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

Cloner un dépôt

- Pour récupérer le Commit depuis firstGit

```
$ git pull origin master
```

- Vérifier cela depuis firstGitBare

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

- Préparer un nouveau Commit à partir de firstGitClone

```
$ echo bonjour >> a.txt  git add .
$ git commit -m "adding a.txt"
```

Cloner un dépôt

- Publier le commit dans firstGitBare

```
$ git push origin master
```

- Vérifier cela depuis firstGitBare

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

- Refaire la même chose depuis firstGit

```
$ echo bonjour >> b.txt
$ git add .
$ git commit -m "adding b.txt"
```

- Publier le commit dans firstGitBare

```
$ git push origin master
```

Cloner un dépôt

- Solution Personnaliser le commit de merge

```
$ git pull origin master
```

- Tout publier dans firstGitBare

```
$ git push origin master
```

Et si on ne veut pas fusionner, on préfère plutôt le rebase

```
$ git pull --rebase origin master
```

Cloner un dépôt

- Remarques
  - On peut également faire un push pour un Commit, un tag...
  - On peut forcer le push -f même en cas de contenu divergeant : notre historique remplacera celui du dépôt distant

Exercice : Cloner un dépôt

1. Créer un nouveau repository GitHub (R1)
2. Ajouter un fichier et le commiter (C1)
3. Cloner le repository (R2)
4. Lister toutes les branches locales et distantes
5. Sur R1, modifier le fichier et commiter (C2)
6. Sur R2, récupérer C2
7. Vérifier avec git log Sur R2,
8. créer une branche B1
9. Aller sur B1, modifier le fichier et commiter (C3)
10. Publier B1 sur R1