
# la gestion de versions avec Git

A. Introduction aux VCS d3 à d6

B. Git

   a. Architecture d7 à d19

   b. Utilisation d20

   c. Installation d21

   d. Commandes de base d21 à d35

# Introduction aux VCS
## A propos des VCS

- Définition
  - Système permettant d'enregistrer dans le temps les modifications apportées à un ensemble de fichiers observés.
  - Précision: indifférent au langage utilisé et ne se limite pas au code.
- Plusieurs acronymes:
  - VCS: Version Control System
  - SCM: Source Code Management
  - RCS: Revision Control System

# Introduction aux VCS
## Pourquoi les VCS ?

- Problème:
  - Maîtriser le code produit durant un développement logiciel implique de savoir ce qui à été fait :
    - Par l'ensemble des développeurs (Qui ?)
    - Dans le temps (Quand ?)
    - Pour quel motif/fonctionnalité (Pourquoi ?)
    - Impliquant de nombreuses fonctions, dans de nombreux fichiers (où ? Comment ?)

→ l'avantage d'un VCS ne se limite pas au travail à plusieurs.

# Introduction aux VCS
## Pourquoi les VCS ?

- Avantages:
  - Permettre la traçabilité d'un développement (historique des changements).
  - Faciliter la collaboration, éviter les conflits ou aider à la leur résolution.
  - Garder une version du code source toujours fonctionnelle, tout en travaillant sur plusieurs fonctionnalités (notion de branche)
  - Permettre des schémas organisationnels structurant les développements (workflows)

# Introduction aux VCS

## Historique

- Principaux VCS
  - CVS (1990) :
    - Concurrent Version System
    - centralisé, travaille sur des fichiers, limité
  - SVN (2000) :
    - abrégé de SubVersioN licence Apache
    - centralisé, travaille sur des fichiers, workflows limités
  - Git (2005) :
    - Software Freedom Conservancy
    - décentralisé, travaille sur des arborescences de contenus
  - et d'autres, libres (Mercurial, …) ou propriétaires (Microsoft, HP, IBM)

# GIT : Architecture

## Origine

- Créé par Linus Torvalds pour gérer les sources du noyau Linux suite à la perte de gratuité de l'outil BitKeeper.
- Depuis sa naissance en 2005, Git a évolué et mûri pour être facile à utiliser.
- VCS le plus utilisé actuellement, libre (GPL), communauté très active.
- Devient un standard (l'est déjà dans le monde du libre)

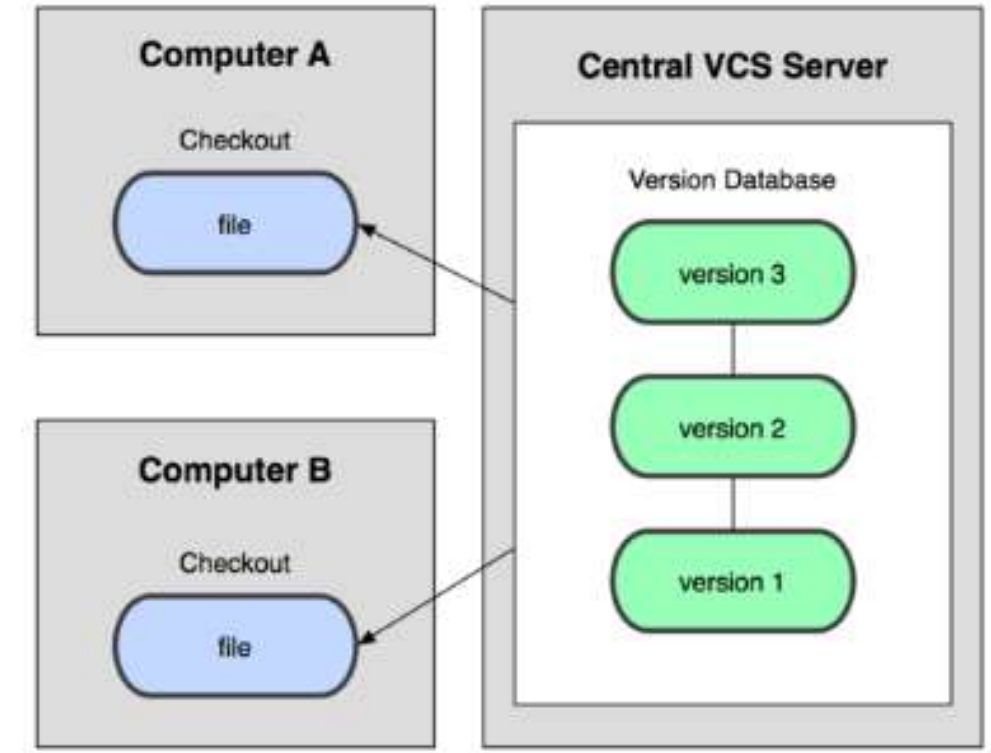
# GIT : Architecture
# un peu de vocabulaire

- Principaux termes
  - **Repository** ou **dépôt**: répertoire versionné (peut être local ou distant)
  - **Commit**: Enregistrement des dernières modifications dans le dépôt
  - **Version** ou revision: état du code source arrêté par un commit
  - **Branche**: version alternative du code source liée à une tentative de développement spécifique
  - **Head**: pointeur sur la version du code chargée (en général le dernier commit)
  - Trunk ou tronc ou **master**: branche principale du code source
  - **Merge**: tentative d'unification de deux branches
  - **Conflit**: problème de merge nécessitant une prise de décision

# GIT : Architecture

## GIT vs SVN

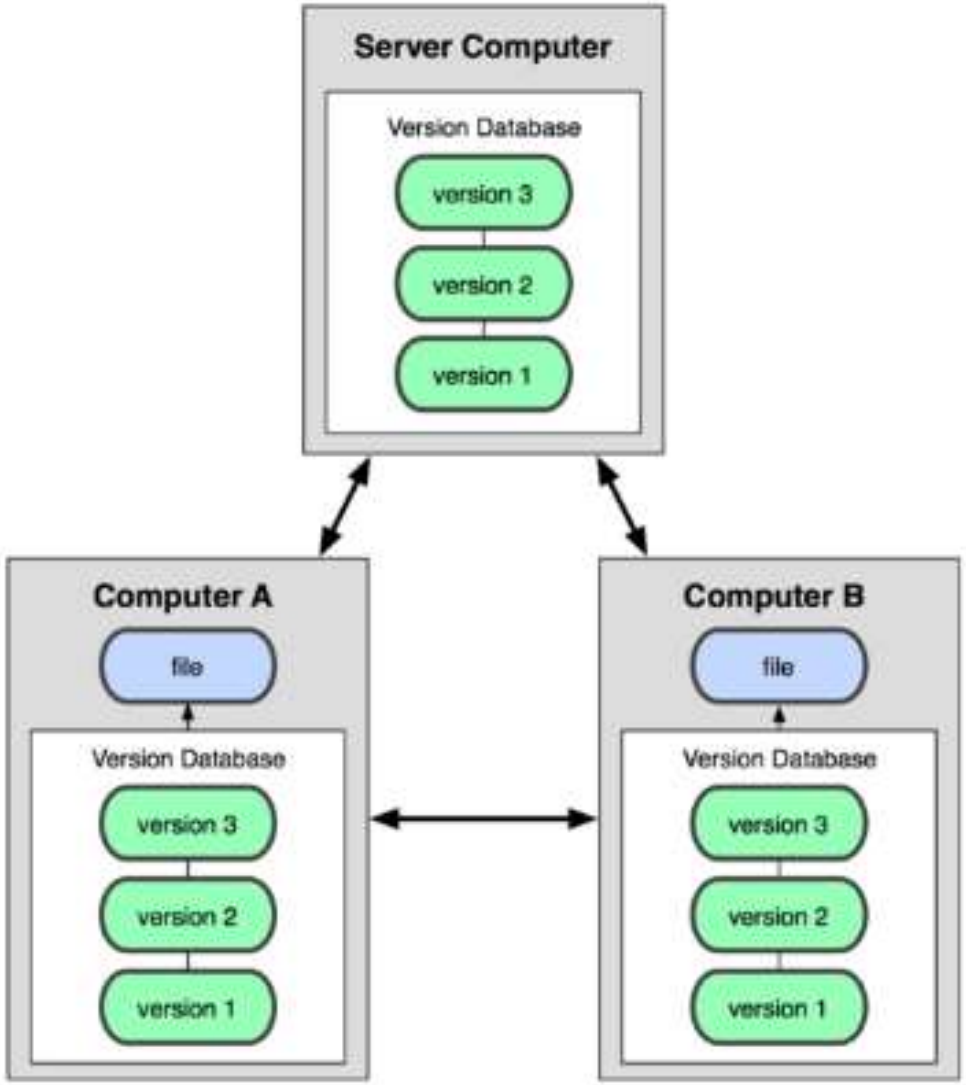
- Pas de serveur central (élément critique)
- Utilisable même si déconnecté
- Organisation du travail plus souple




Gestionnaire de version centralisé

# GIT : Architecture

## GIT vs SVN




Gestionnaire de version décentralisé

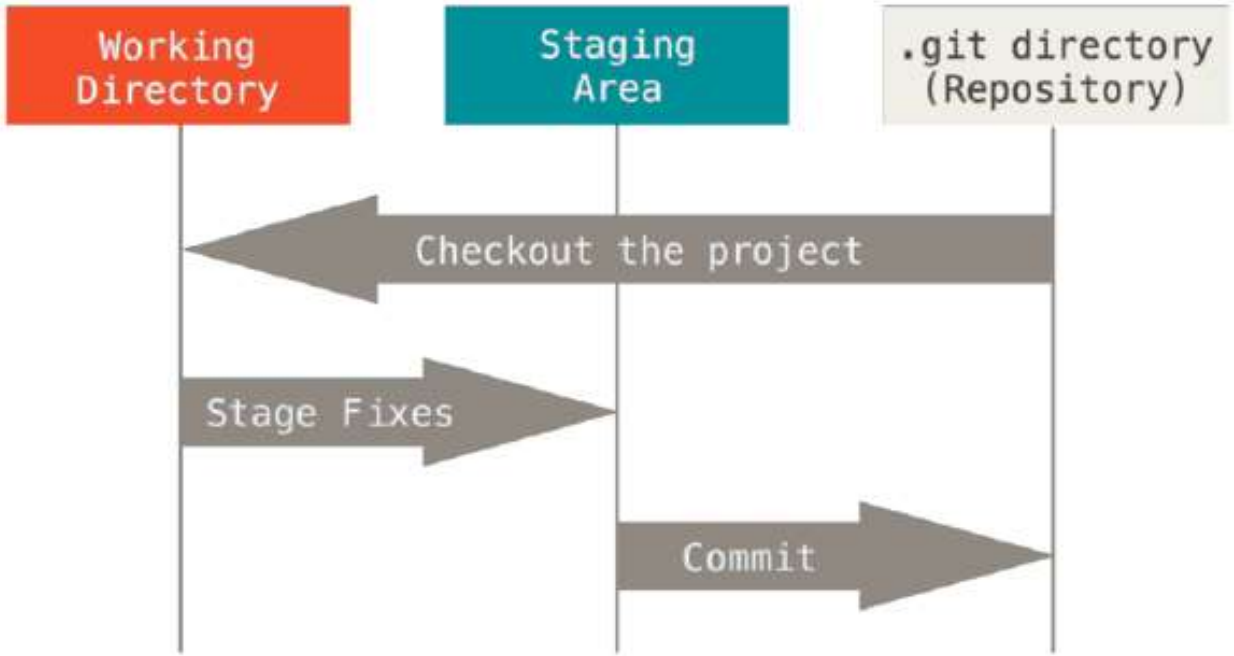
# GIT : Architecture
## VCS décentralisé

- Principes fondateurs
  - Chaque client git exécute son propre dépôt en local
  - Chaque utilisateur d'un dépôt partagé possède une copie de tout l'historique des changements enregistrés (full mirroring)
  - Abandon d'une vision chronologique des changements (pas d'ordre strict entre les commits) pour une vision structurelle (graphe de commits)

  - But : faciliter les développements parallèles, permettre au code de diverger/converger rapidement

# GIT : Architecture
## Zones de stockage dans git

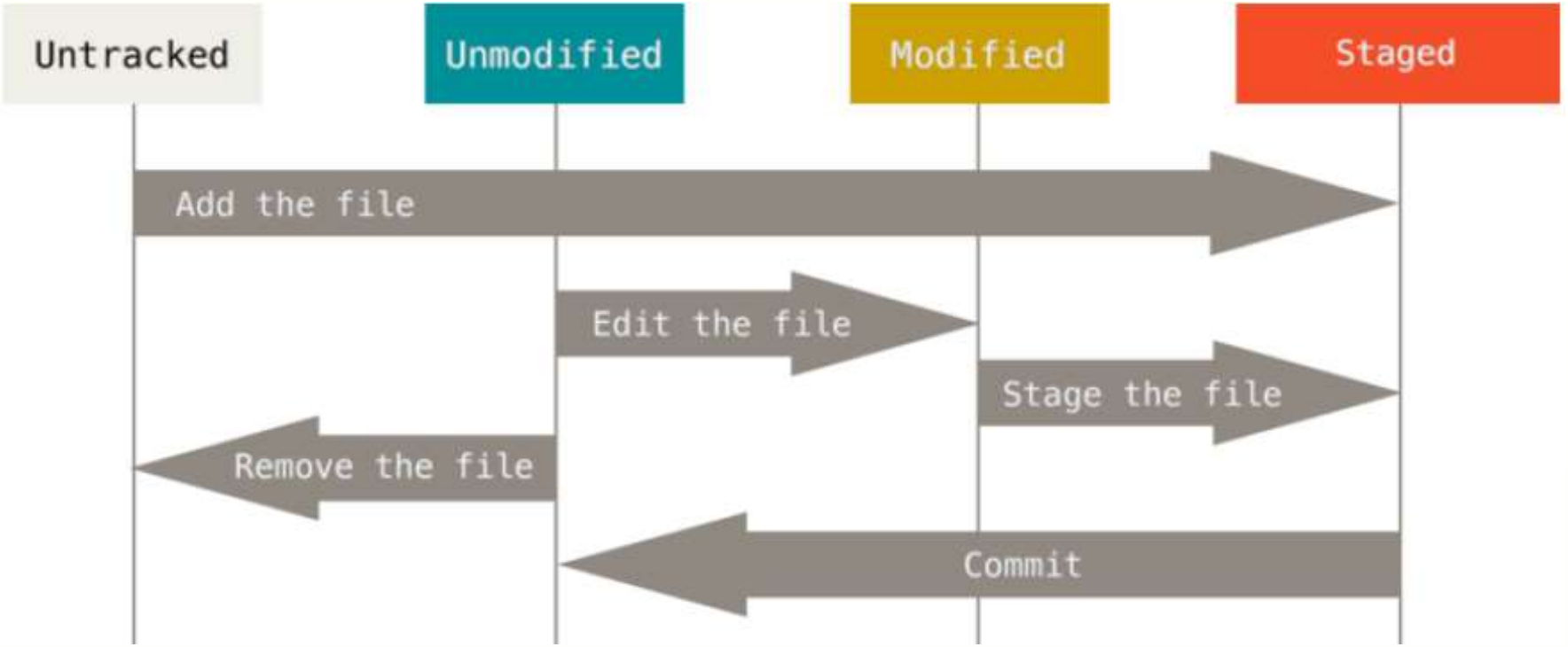
- Découpage interne en trois zones
  - le répertoire de travail (**working directory**) local où sont réalisés les changements
  - la « **staging area** » (aussi appelée **index**) où sont préenregistrés les changements (en attente de commit)
  - le dépôt git où sont enregistrés les changements



# GIT : Architecture

## staging area

- Index intermédiaire
  - Sert à préparer les commits progressivement
  - git commit enregistre les modifications indexées
  - La staging area peut être bypassée: git commit -a



# GIT : Architecture
# Organisation des informations

- Emplacement du dépôt local

  Les données propres à git sont stockées dans un unique répertoire « .git » a la racine du projet. C'est le dépôt local.

- Références

  L'historique du projet est un graphe de commit. Certaines références sur ce graphe sont utiles :

  - **master** : reference la branche principale
  - **HEAD** : par défaut, référence le commit le plus récent de la branche courante (sera le parent du prochain commit)

# GIT : Architecture
## Bonnes pratiques

- Ne pas versionner de fichiers créés automatiquement (logs, pdf, executables, etc.) ou personnels
  - Utiliser le fichier « .gitignore » pour définir les extensions de fichiers qui ne seront pas versionnées :

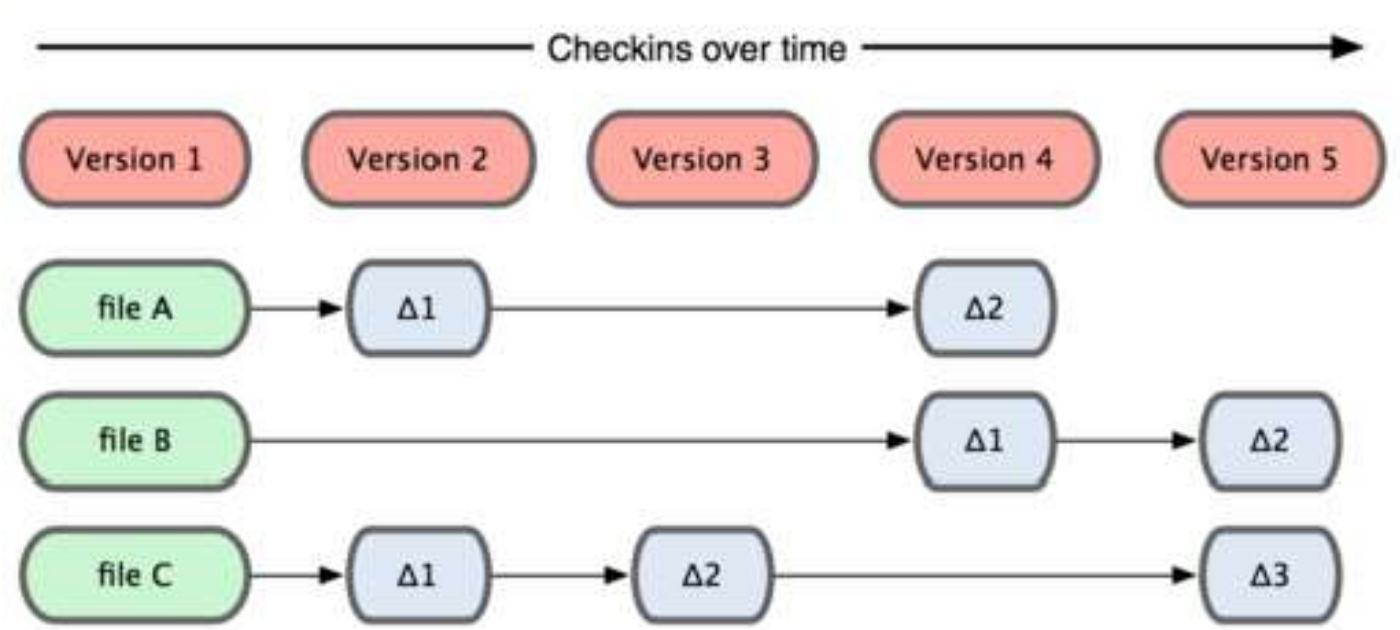
    https://github.com/github/gitignore
- Faire de petits commits réguliers et facile a intégrer, leur donner un nom explicite
- Utiliser les branches pour :
  - les développements à plusieurs
  - chaque développement conséquent d'une nouvelle fonctionnalité
- Ne pas développer sur la branche master à plusieurs pour éviter les conflits lors des pull
- Faire de petits commits locaux, et pusher des commits plus conséquents, toujours testés et fonctionnels !
- Faire des pull régulièrement

# GIT : Architecture
## Comment Git fonctionne

- Des instantanés, pas des différences
  - CVS, SVN ne stockent que les deltas des fichiers modifiés par un commit
  - Git stocke tout le contenu du répertoire versionné à chaque commit (mais utilise une compression intelligente basée sur la version antérieure la plus proche)
  - Permet une grande souplesse

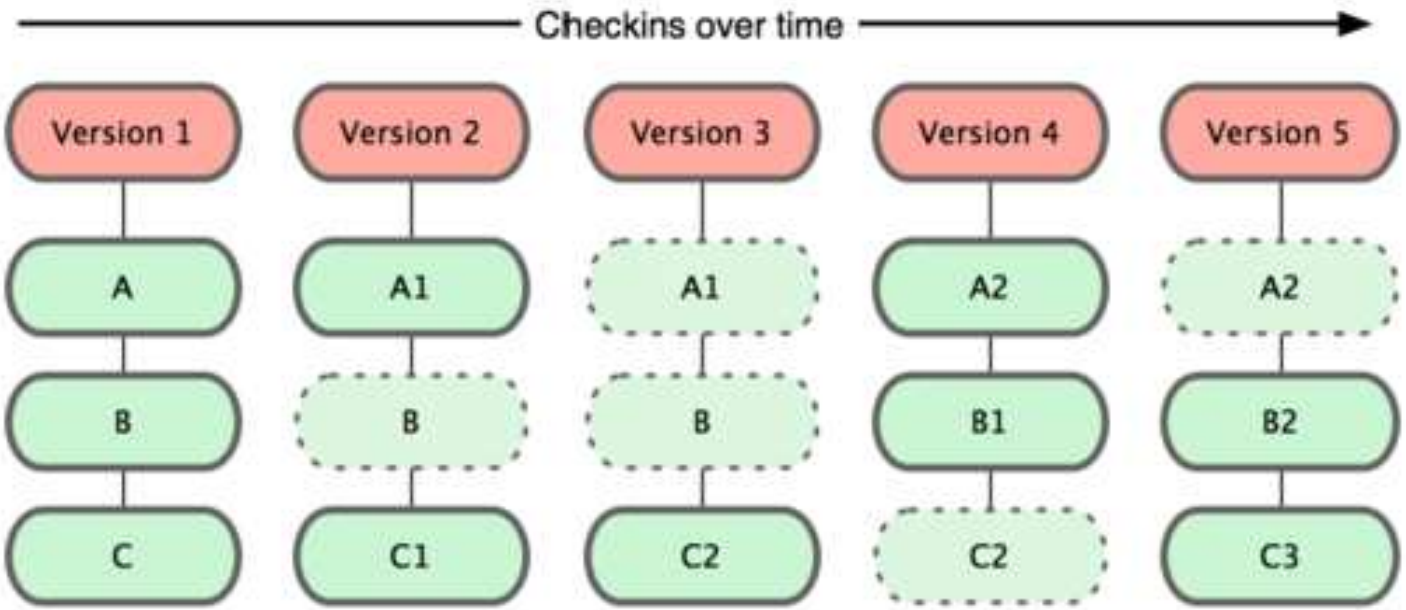
# GIT : Architecture
## Comment Git fonctionne

- CVS, SVN ne stockent que les deltas:



# GIT : Architecture
## Comment Git fonctionne

- Git stocke tout le contenu du répertoire



# GIT : Architecture
# Comment Git fonctionne

- Types de données Git:
  - **Blob** : contenu d'une version d'un fichier (Binary Large Object)
  - **Tree** (structure récursive) : arborescence de références vers d'autres Trees et Blobs
  - **Commit** (structure récursive) : pointe sur un Tree, sur le Commit parent et contient des métadonnées (date, auteur, etc.)
  - **Tag** : annotation manuelle d'un commit et créant une branche statique (équivalent à un pointeur)
  - **Identifiant** unique pour tout objet : hash SHA1 (Secure Hash Algorithm) de 40 caractères

# GIT : Utilisation la ligne de commande

- Plusieurs solutions pour utiliser Git
  - En ligne de commande:
    - Par l'invite de commande (commande « cmd » dans la barre de recherche Windows)
    - Le Powershell de Windows (commande « powershell » dans la barre de recherche Windows)
    - C'est de cette façon que l'on a accès à toutes les commandes Git.
  - En utilisant des utilitaires fournissant une interface graphiques avec des capacités variables.
  - Si vous connaissez les principales commandes en ligne, vous saurez à même de comprendre l'interface graphique

# GIT : Installation sur Windows

- Il existe aussi plusieurs manières d'installer Git sur Windows.
  - Rendez-vous sur:

    http://git-scm.com/download/win

  - Installer *Github for Windows*.

https://github.com/join?plan=free&source=pricing-card-free

L'installateur inclut une version en ligne de commande avec l'interface graphique.

# GIT : les commandes

## Configuration

- Après l'installation de Git, il faut le personnaliser afin qu'il vous connaisse (pour les commits, votre nom et mail seront utilisés).
- Ceci ce fait par la commande: git config …
- Git stocke ces informations dans un fichier « gitconfig » qui peut se retrouvé à plusieurs endroits:
  - Fichier /etc/gitconfig : Contient les valeurs pour tous les utilisateurs et tous les dépôts du système. Si vous passez l'option --system à git config, il lit et écrit ce fichier spécifiquement.
  - Fichier ~/.gitconfig : Spécifique a votre utilisateur. Vous pouvez forcer Git a lire et écrire ce fichier en passant l'option --global.
  - Fichier config dans le répertoire Git du dépôt en cours d'utilisation (c'est-a-dire .git/config): spécifique au seul dépôt en cours.

# GIT : les commandes

## Votre identité

- C’est une information importante car toutes les validations dans Git utilisent cette information et elle est indélébile.
- Configuration globale commune à tous les dépôts d'un utilisateur:
  - git config --global user.name “votre nom"
  - git config --global user.email votre_mail@xxx.fr

# GIT : les commandes
## Votre éditeur de texte et autre

- On peut configurer l'éditeur de texte qui sera utilisé quand Git demande de saisir un message:
  - git config --global core.editor votre_éditeur.exe
- La coloration du texte:
  - git config --global color.ui true/false
- Vérifier sa configuration:
  - git config --list
- Obtenir de l'aide:
  - git help <commande>
  - Git <commande> --help

# GIT : les commandes
## Premières commandes

- Initialisation du dépôt
  - Versionner un répertoire courant : **git init** (créer un dépôt en générant le répertoire « .git » à la racine)
  - Télécharger un dépôt existant : **git clone url**
- Indexer des modifications
  - Ajouter un fichier à suivre pour le prochain commit: **git add fichier_ou_dossier**
- Enregistrer les modifications dans le dépôt local: **git commit fichier**
  - ou **git commit -m "Description du commit"**
  - ou **git commit -a** (pour all, commit tous les fichiers modifiés sans passer par la staging area)

# GIT : les commandes
## Vérifier l'état des fichiers

- L'outil principal pour déterminer quels fichiers sont dans quel état est la commande:
  - git status

```
Glossaire>git status
On branch master
nothing to commit, working tree clean
```

- Si un fichier a été ajouté: ex js.txt

```
Glossaire>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        js.txt

nothing added to commit but untracked files present (use "git add" to track)
```

# GIT : les commandes
## Vérifier l'état des fichiers

- Si un fichier a été modifié: ex html.txt

```
Glossaire>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   html.txt

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

- On s'aperçoit que les changement ne seront pas commiter.

# GIT : les commandes
## Suivre des fichiers

- Pour commencer a suivre un nouveau fichier, vous utilisez la commande:
  - git add nom_fichier
- Pour suivre un fichier qui vient d'être modifié:
  - git add nom_fichier: ex html.txt

```
Glossaire>git add html.txt
```

```
Glossaire>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   html.txt
```

# GIT : les commandes
## Suivre des fichiers

- Si on remodifie le même fichier avant d'avoir effectué le commit:

```
Glossaire>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   html.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   html.txt
```

- On s'aperçoit que les premières modifications seront prises en compte mais pas les deuxièmes.
- Il faut refaire un « git add html.txt »

# GIT : les commandes
## Ignorer des fichiers

- Certains types de fichiers doivent être ignorés (fichier de config de votre IDE, fichiers générés automatiquement, fichiers de log, …)
- Il faut créer dans le dépôt un fichier nommé « .gitignore »
- Les règles de construction du fichier .gitignore
  - Les lignes correspondent à des « patrons »
  - les lignes vides ou commençant par # sont ignorées
  - les patrons standards de fichiers sont utilisables
  - si le patron se termine par une barre oblique (/), il indique un répertoire

# GIT : les commandes
## Exemple de .gitignore

```
# pas de fichier .a
*.a
# mais suivre lib.a malgré la règle précédente
!lib.a
# ignorer uniquement le fichier TODO à la racine du projet
/TODO
# ignorer tous les fichiers dans le répertoire build
build/
# ignorer doc/notes.txt, mais pas doc/server/arch.txt
doc/*.txt
# ignorer tous les fichiers .txt sous le répertoire doc/
doc/**/*.txt
```

# GIT : les commandes
## Voir ce qui a été modifié

- Pour visualiser ce qui a été modifié mais pas encore indexé:
  - git diff
- visualiser les modifications indexées:
  - git dif --staged

```
Glossaire>git diff
diff --git a/html.txt b/html.txt
index dd8bca0..59f3ae0 100644
--- a/html.txt
+++ b/html.txt
@@ -8,6 +8,4 @@ BODY
         partie de la page Html contenant les balises

 TAG, BALISE
-        Objet graphique visible pour l'utilisateur
-
-
+        Objet graphique visible pour l'utilisateur
\ No newline at end of file
```

# GIT : les commandes
## Valider les modifications

- Maintenant que votre zone d'index est dans l'état désiré, vous pouvez valider vos modifications:
  - git commit

  Ce qui ouvre l'éditeur pour vous demander de rentrer un message significatif de votre commit
  - git commit -m "votre message"

# GIT : les commandes
## Annuler les modifications

- Une des annulations la plus commune apparait lorsqu'on valide une modification trop tôt en oubliant d'ajouter certains fichiers, ou si on se trompe dans le message de validation.
- Pour rectifier cette erreur:
  - git commit --amend

1. La zone d'index n'a pas été modifiée:

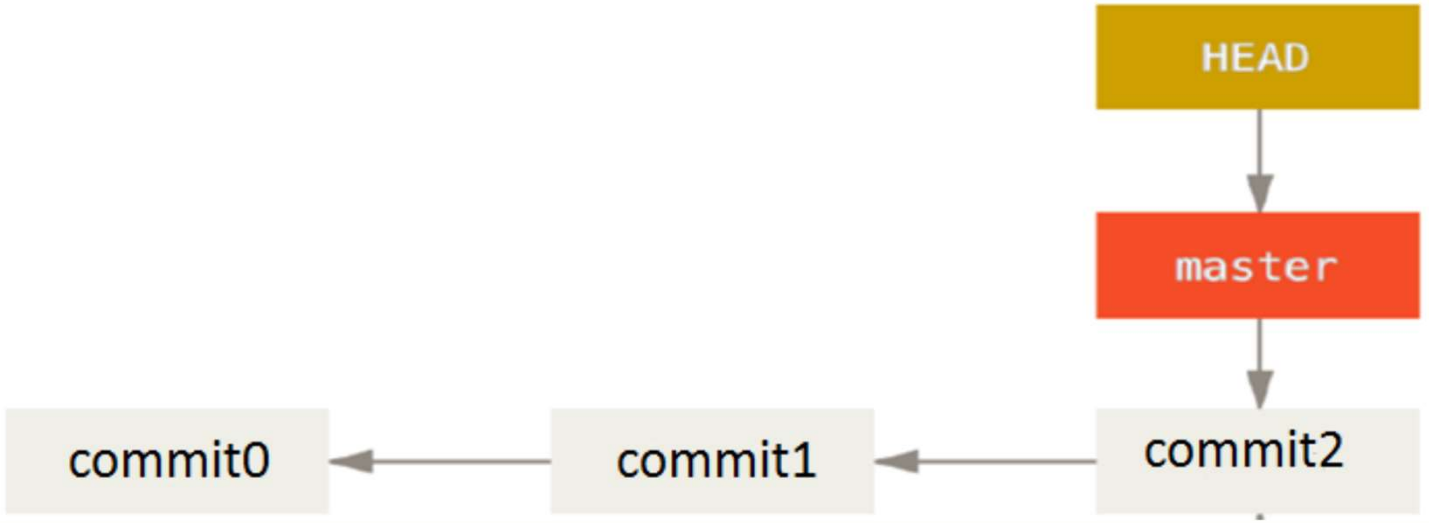
   l'éditeur de texte s'ouvre en vous donnant l'ancien message que vous pouvez modifier
2. La zone d'index a été modifiée par l'ajout ou la modification de fichiers:

   Vos modification seront prises en compte

- Des les 2 cas vous n'aurez qu'un seul commit dans le dépôt.

# GIT : les commandes
## Gestion des branches

- Une branche dans Git va vous permettre de travailler sur une nouvelle fonctionnalité, un correctif ou autre sans casser l'état de votre projet que vous avez validé (commité).
- Lors de l'initialisation de votre dépôt local (git init …) qui a créé par défaut une branche appelée « master ».
- Git maintient un pointeur nommé « HEAD » qui lui indique sur quel branche il travaille :

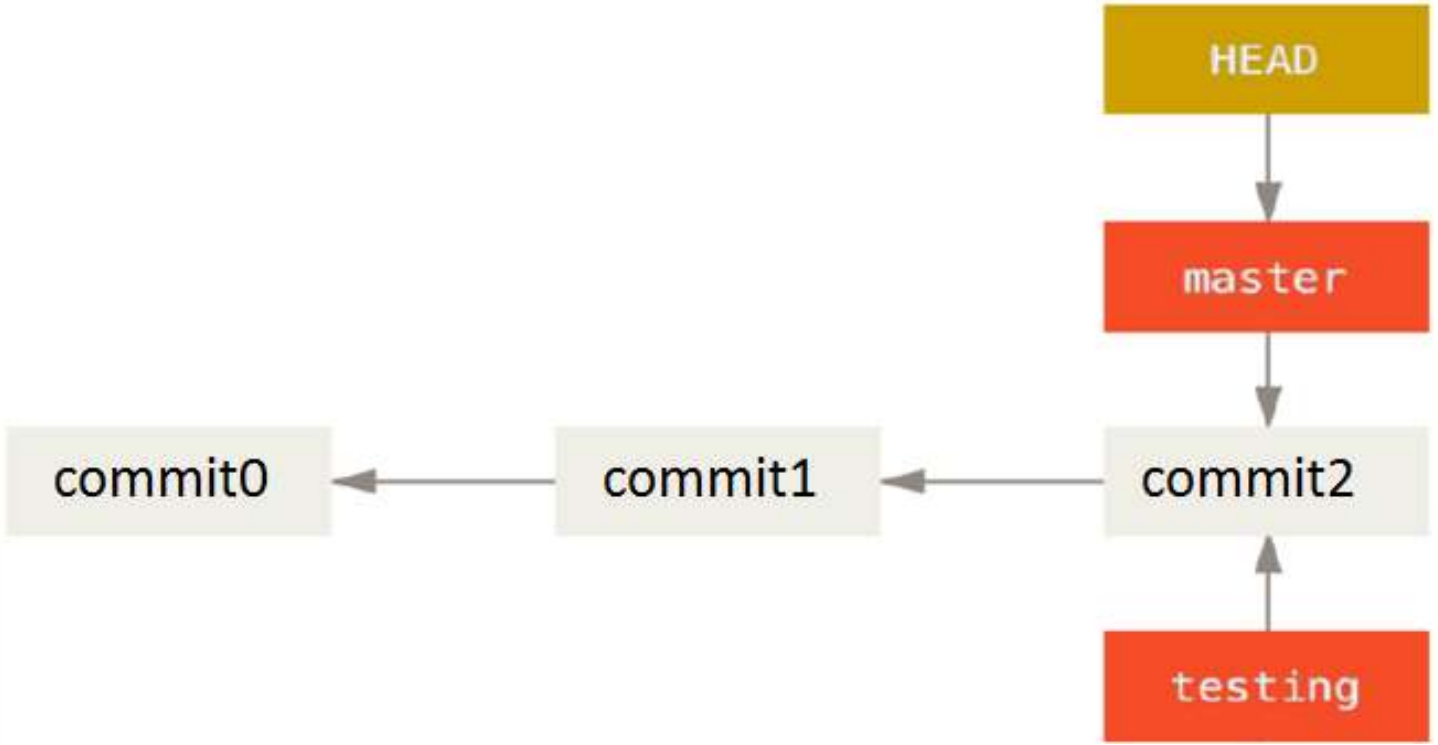


# GIT : les commandes
## Créer une branche

- Pour créer une branche, entrer la commande :

  « **git branch nom_de_nouvelle_branche** »

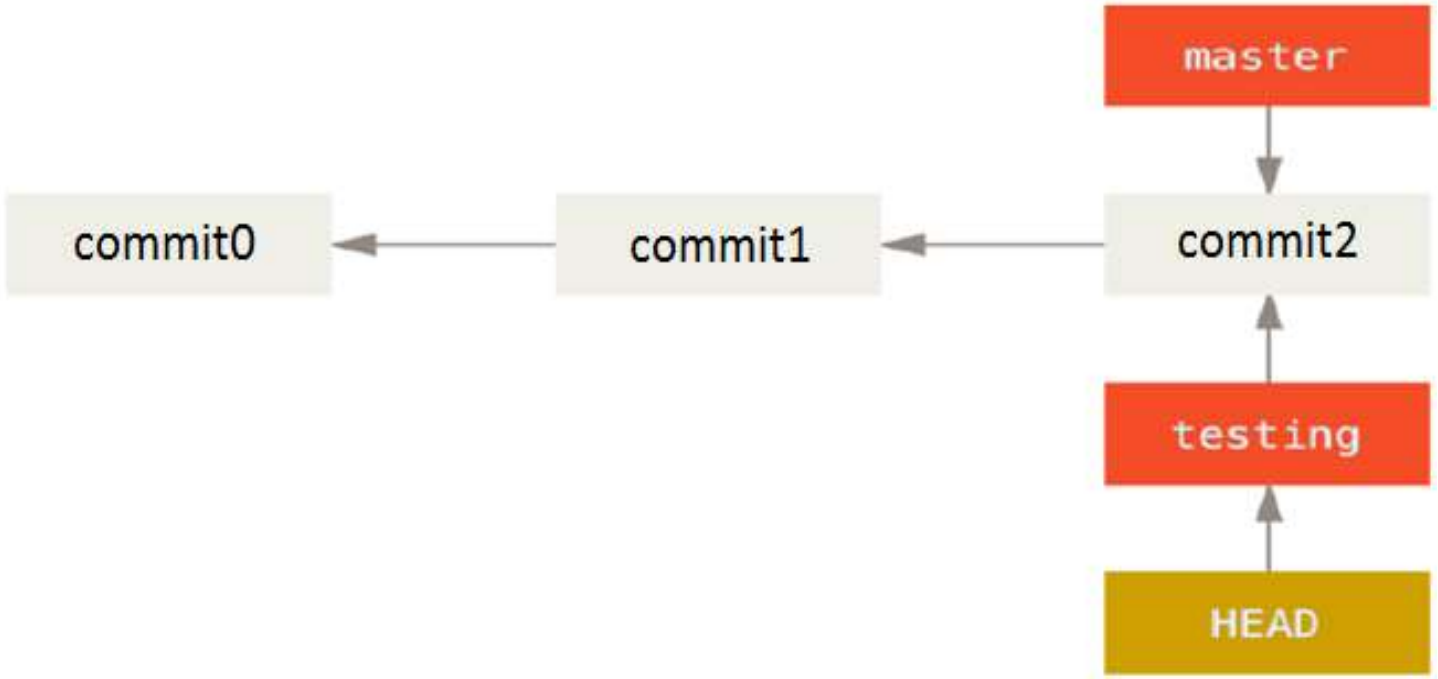
- Attention : ceci crée la branche mais HEAD pointe toujours sur la branche « master ».



# GIT : les commandes
## Changer de branche

- Pour changer de branche afin de travailler sur cette nouvelle fonctionnalité, entrez la commande :

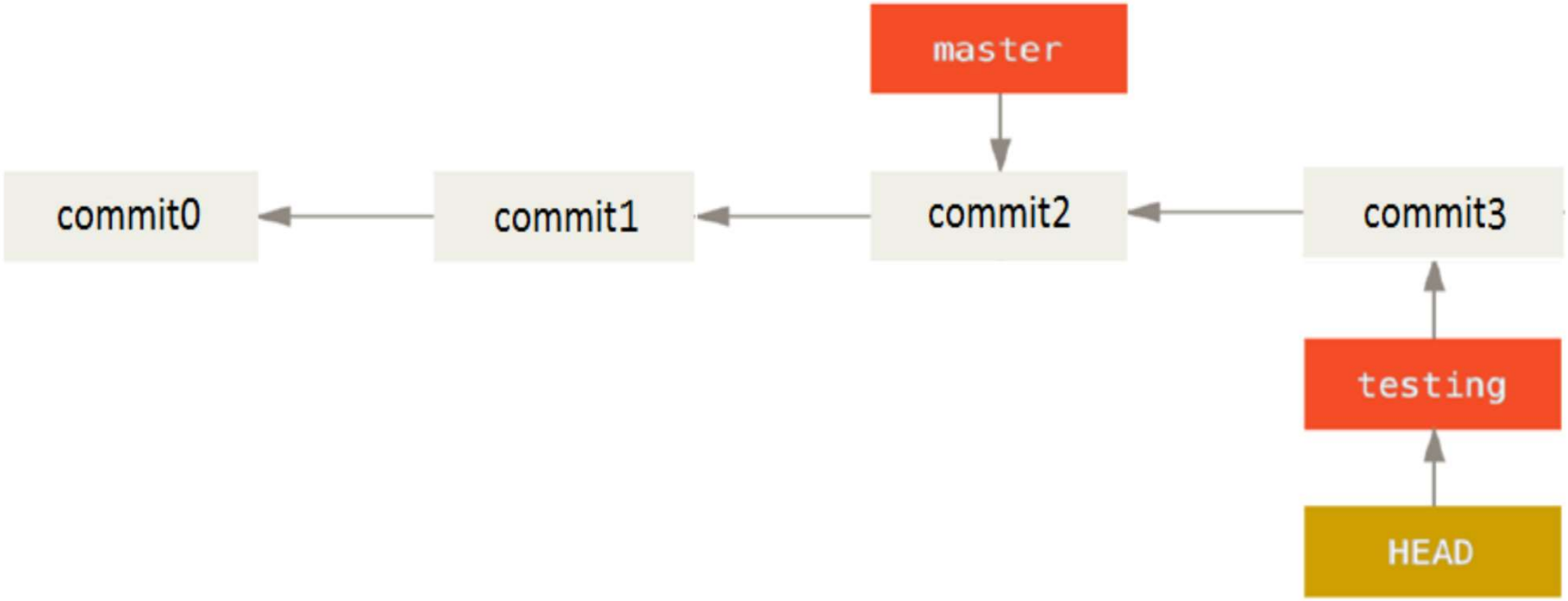
**« git checkout nom_de_la_branche »**




- La commande « **git checkout -b nom_branche** » réalise les 2 opérations : git branch et git checkout

# GIT : les commandes
## Fusion de branche

- Vous travaillez sur votre branche.
- Lorsque votre fonctionnalité ou correctif est terminé et opérationnel, vous l'avez validé par un commit final.

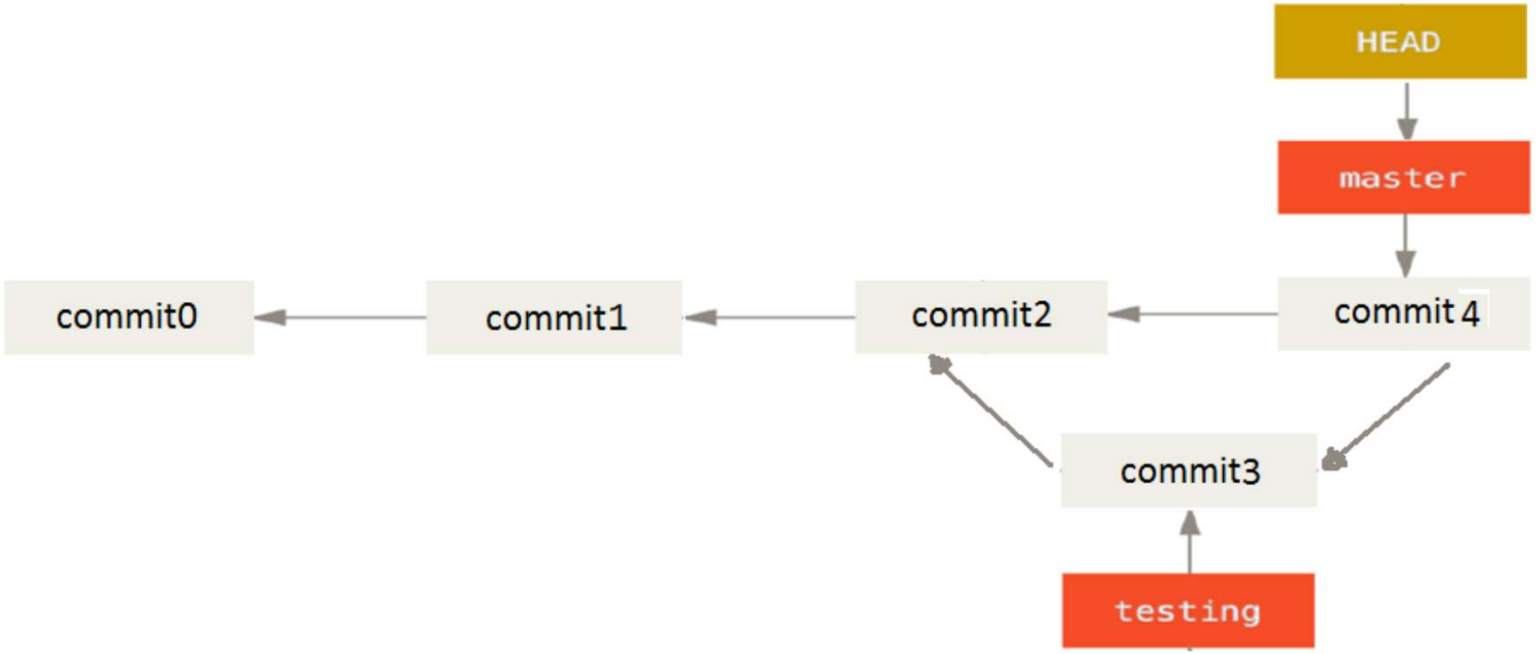


# GIT : les commandes
## Fusion de branche

- Il faut maintenant fusionner cette branche avec la branche principale « master ».
- Pour cela :
  - Retourner sur la branche master

    **git checkout master**
  - Merger master avec la branche

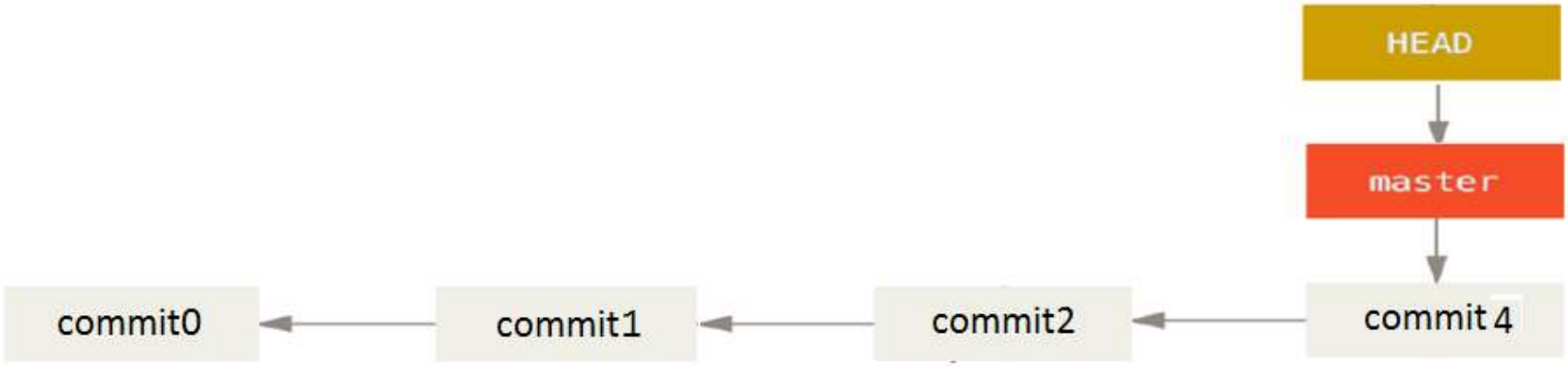
    **git merge nom_de_la_branche**



# GIT : les commandes
## Fusion de branche

- Si vous n'avez plus besoin de la branche, vous pouvez la supprimer :

**git branch -d nom_de_la_branche**



# GIT : les commandes

## Résumé

Source : Wikipedia