

# Git

# Sommaire

1. Présentation de Git
2. Les fondamentaux
3. Les branches
4. Synchronisation à distance
5. Les outils de git
6. Git workflow

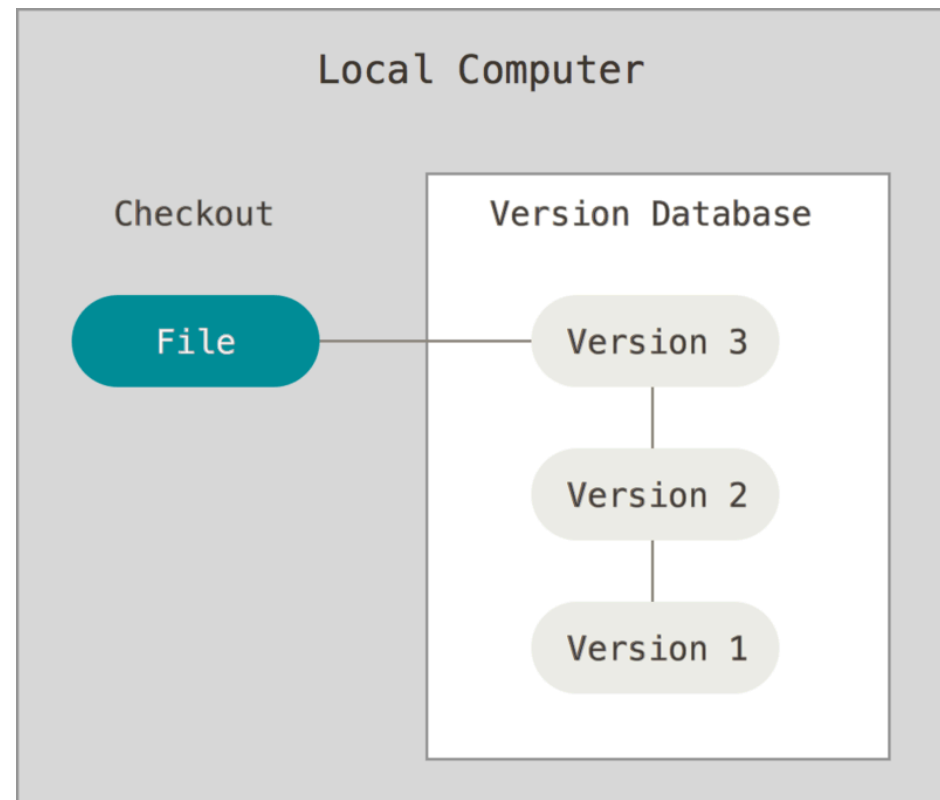
# 1. Présentation de Git

# Définition de gestion de version

“ Un gestionnaire de version est un système qui enregistre **l'évolution d'un fichier** ou d'un **ensemble de fichiers** au cours du **temps** de manière à ce qu'on puisse rappeler une **version antérieure** d'un fichier à tout moment ”

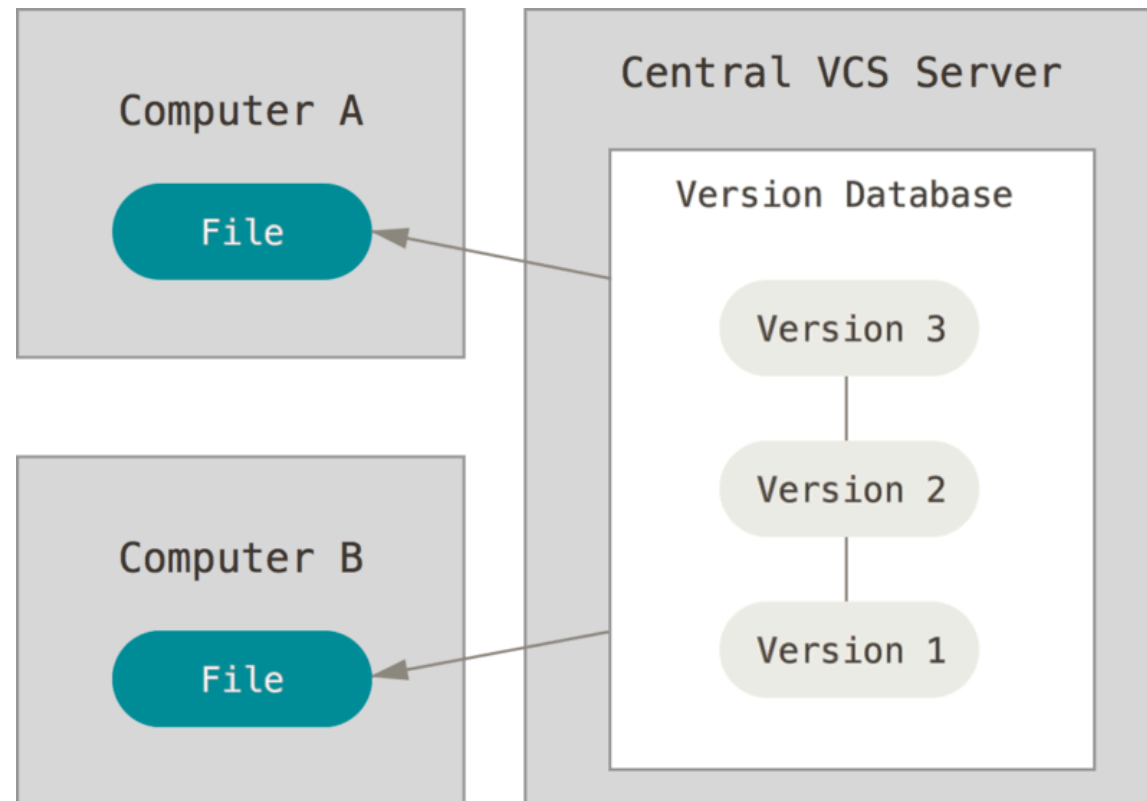
# Les systèmes de gestion de version locaux

- Une méthode courant consiste à recopier les fichiers dans un autre répertoire avec un nom différent
- Les développeurs ont ensuite inventé des VCS locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier



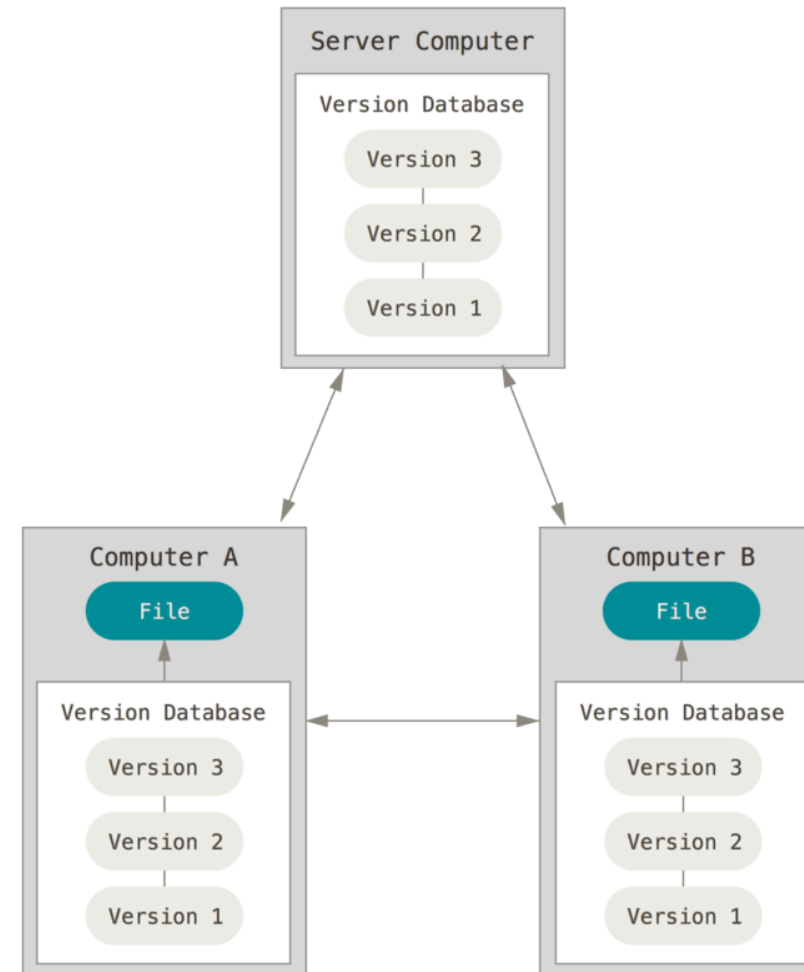
# Les systèmes de gestion de version centralisés

- Pour gérer la collaboration sur des projets, des CVCS ont été inventés
- **avantages:** gestion des droits, visibilité sur l'avancement du projet
- **défauts:** risque de panne du serveur central



# Les systèmes de gestion de version distribués

- Les Distributed Version Control Systems (DVCS) dupliquent entièrement le dépôt
- En cas de problème avec le serveur, une restauration peut être effectuée
- Ces outils permettent de mettre en place différentes chaînes de traitements



# Histoire de Git

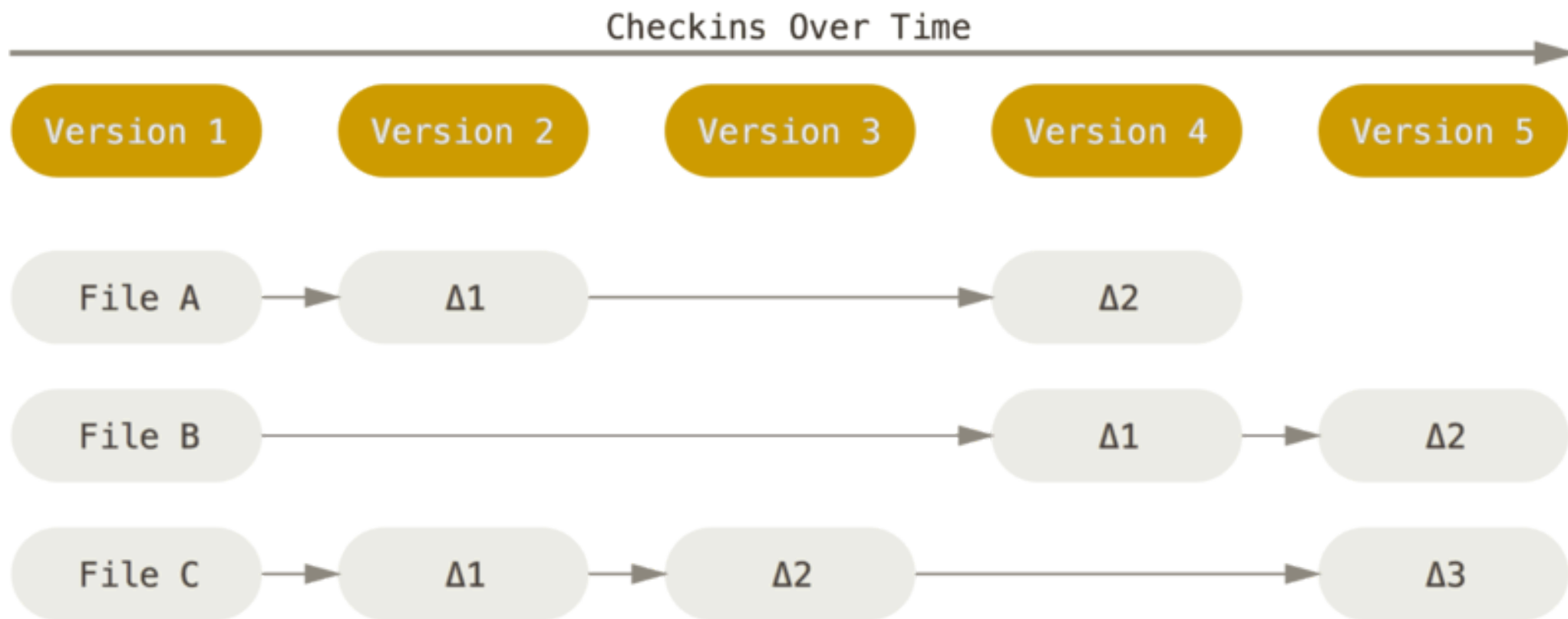
- Git a été créé par **Linus Torvald** (inventeur de Linux) en 2005
- Il est né d'un désaccord entre la communauté de Linux avec l'outil BitKeeper (DVCS propriétaire) devenu payant
- Git est basé sur les objectifs suivants:
  - **Vitesse** et **conception simple**
  - Support pour le **développement non linéaire**
  - Système **distribué**
  - Gestion de **projets d'envergures**



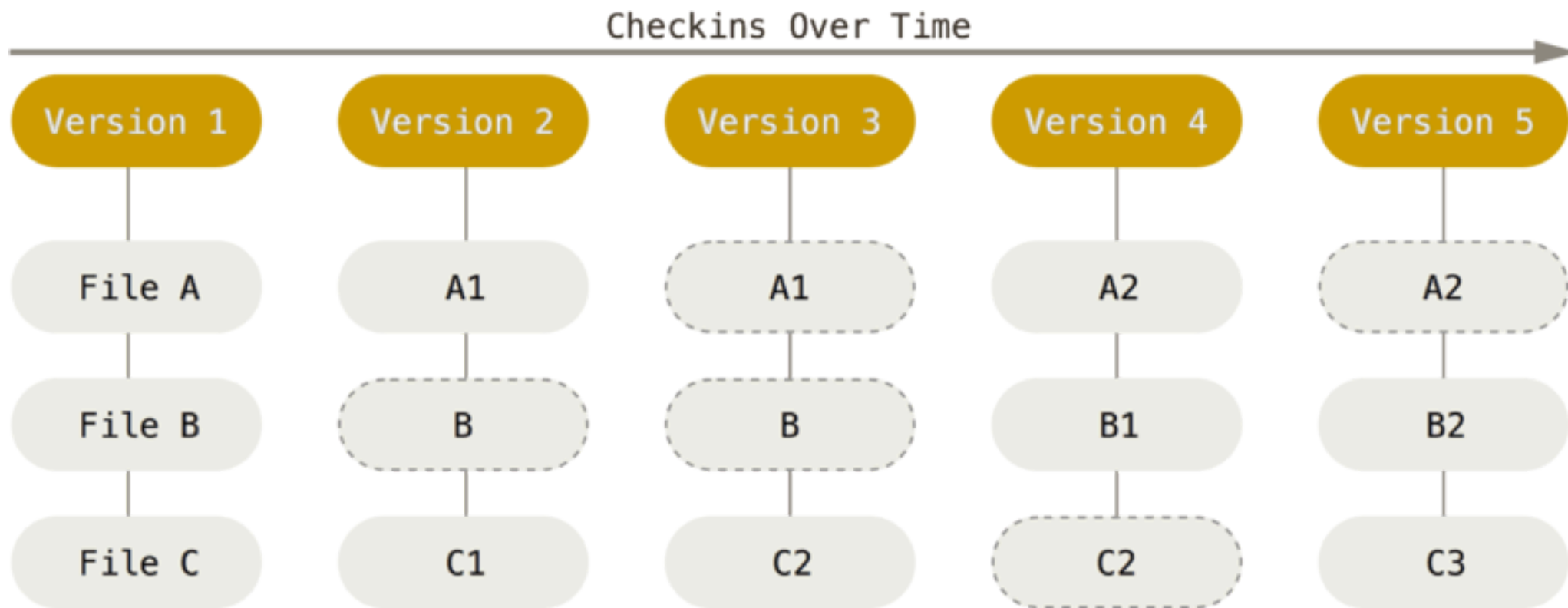
# Fonctionnement de Git

- Le système de versionnage de git se base sur le principe de **snapshots** et non pas de différences entre les fichiers
- Chaque fois que l'état d'un projet est validé ou enregistré, git enregistre une référence à ce snapshot
- Si les fichiers n'ont pas été modifiés, git n'enregistrera pas à nouveau le fichier mais fera référence à sa dernière version modifiée

# Systeme basé sur la différence des fichiers



# Systeme basé sur les snapshots (git)



# Installation

- Pour installer git il suffit de se rendre à l'URL suivant:  
<https://git-scm.com/downloads>



# Paramétrage de git

- **git config** est un outil qui permet de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git
- Configuration de l'identité (utilisée dans les validations) :

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

- Configuration de [l'éditeur](#) (vscode) :

```
git config --global core.editor "code --wait"
```

# Vérification des paramètres

- Définir le nom de la branche par défaut (de base 'master')
  - `git config --global init.defaultBranch main`
- Pour afficher la configuration et son origine
  - `git config --list --show-origin`
- Pour afficher la valeur d'un paramètre
  - `git config [paramètre]`

## Obtenir de l'aide

- Pour obtenir la documentation liée à une commande, il suffit d'utiliser les différentes commandes suivantes

```
git help <commande>  
git <commande> --help  
man git-<commande>
```

- Par exemple pour obtenir l'aide de la commande config:  
`git help config`
- Pour une documentation concise: `git <command> -h`

## 2. Les fondamentaux



# Glossaire

- **branch** : version du dépôt pour travailler sur une fonctionnalité particulière sans impacter le code courant
- **commit** : enveloppe qui contient une petite portion de codes modifiés d'un ou plusieurs fichiers
- **dépôt** : environnement virtuel d'un projet qui permet d'enregistrer les versions du code et d'y accéder au besoin
- **merge** : action qui applique les changements d'une branche sur une autre

# Glossaire

- **HEAD** : pointeur vers le dernier commit de la branche en cours push
- **fetch** : récupère les commits du dépôt distant en local sans les appliquer
- **tag** : pointeur vers un commit particulier (release)
- **pull** : télécharge les commits manquant du dépôt distant sur le dépôt local et les applique
- **Index/staging area** : Zone mémoire qui contient les fichiers mis de côté pour préparer le commit. On dit que les fichiers sont indexés

# Initialiser un dépôt

- Pour créer un nouveau dépôt git saisir: `git init`
- Cette commande crée un répertoire **.git** dans le répertoire courant
- Le répertoire **.git** contient tous les fichiers nécessaires au fonctionnement d'un dépôt git

```
|—hooks  
|—info  
|—objects  
|—refs  
  |—heads  
  |—tags
```

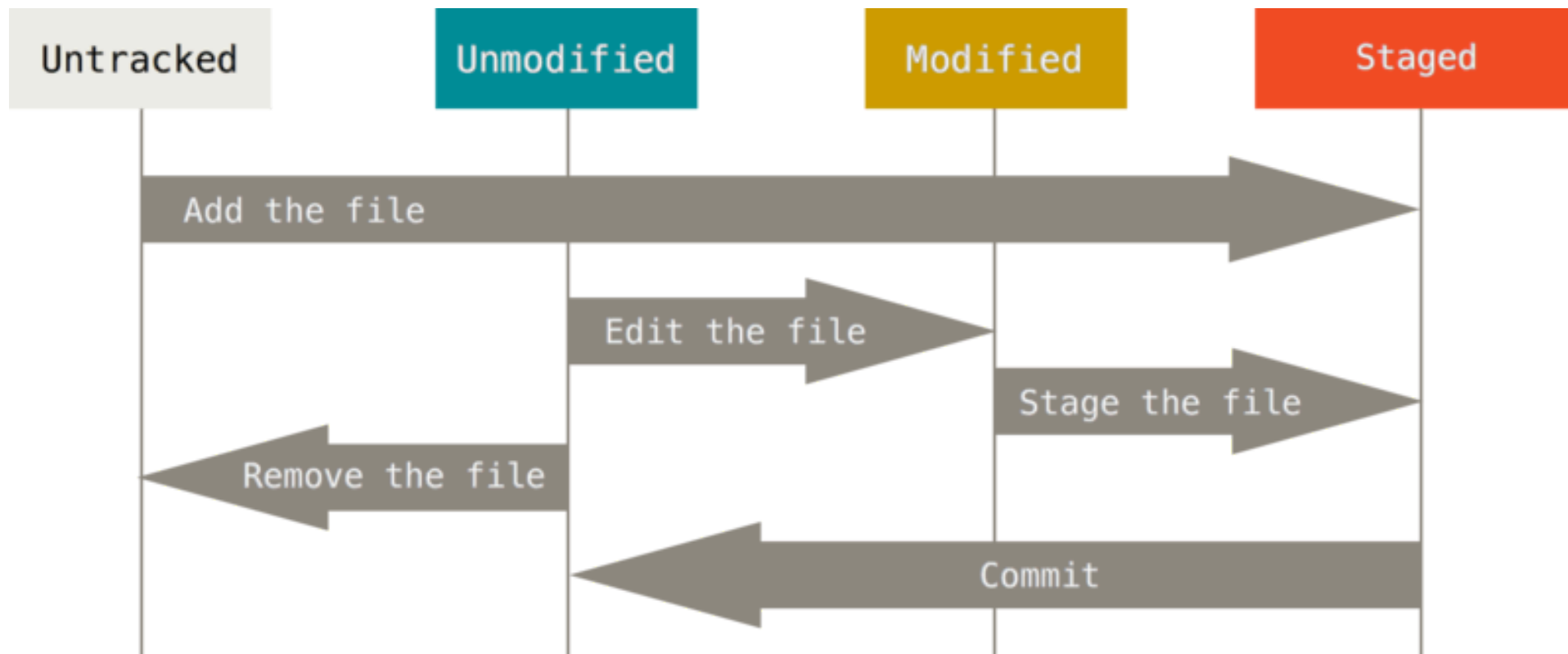
# Cloner un dépôt existant

- Pour obtenir les sources d'un dépôt existant `git clone [url]`
- Git reçoit une copie des données sur le serveur distant avec toutes les versions et l'historique du projet
- `git clone https://github.com/utopios/pokemon-app`
- Cette commande crée un répertoire `pokemon-app`, initialise un répertoire `.git` avec toutes les données du dépôt

## Enregistrer des modifications dans le dépôt

- Chaque fichier dans le répertoire de travail peut avoir deux états
  1. **non suivi** : non connu par git
  2. **suivi** : appartient à un snapshot, il peut être **inchangé**, **modifié** ou **indexé**

# Le cycle de vie des fichiers



# Ajouter un fichier au suivi de version

- `git add [fichier]` : permet d'ajouter un fichier au suivi de version
- `git add .` : ajoute le répertoire actuel au suivi de version
- `git add --all` : fait la même chose que `git add .`
- `git add -A` : raccourci de la commande précédente

## Vérifier l'état des fichiers

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

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



## Ignorer des fichiers

- Certains fichiers ne nécessitent pas d'être versionnés, par exemple : les fichiers de compilations, les logs, les fichiers temporaires, les clés d'api ...
- Pour éviter de tracer ces fichiers dans git, il faut créer un fichier `.gitignore` à la racine du projet
- Les lignes commençant avec le caractère `#` sont des commentaires
- Si la ligne termine par `/` cela indique un dossier

## Exemple .gitignore java

```
# fichiers de class compilés
*.class

# Fichiers logs
*.log

# Package
*.jar
*.war

# répertoire secret
secret/
```


# Comparer des changements

- `git diff` est une commande Git qui lance une fonction de différenciation sur les sources de données Git
- Ces sources de données peuvent être des commits, des branches, des fichiers, etc.
- `git diff` montre les lignes exactes qui ont été ajoutées, modifiées ou effacées – le patch en somme
- `git diff --staged`: permet de comparer les fichiers indexés et le dernier snapshot

# Valider des modifications

- Une fois le staging area (zone d'index) dans l'état désiré, il faut valider les modifications
- Pour valider les modifications: `git commit`
- Pour documenter les modifications d'un commit, on ajoute **toujours** un message descriptif au commit avec le paramètre `git commit -m "<message>"`
- Cette opération crée un nouveau **snapshot**

## Ajouter et valider des modifications

- Il existe un raccourci qui permet d'ajouter les fichiers en suivi de version directement via l'opérateur de commit
- `git commit -a` : cette commande ordonne à tous les fichiers de se placer dans la zone d'index et de réaliser la validation ce qui permet de s'absoudre de taper `git add`
- cas d'usage : `git -am "maj fichiers"`
-  Les nouveaux fichiers créés non suivis ne seront pas pris en compte

# Visualiser l'historiques des validations

- Pour visualiser l'historique des commits : `git log`
- Cette commande invoque les commits du plus récents aux plus anciens
- Le paramètre `git log -p -<nombre>` permet d'afficher les différences entre les validations, on peut préciser le nombre de commit que l'on souhaite afficher
- `git log --stat` permet d'afficher les statiques des commits

## git clean

Utilisé pour supprimer des fichiers non suivis dans un répertoire de travail

- Commandes courantes :
  - `git clean -n` : Affiche les fichiers qui seraient supprimés
  - `git clean -f` : Supprime les fichiers non suivis
  - `git clean -fd` : Supprime les fichiers et les répertoires non suivis

## git revert

Annule un commit en créant un nouveau commit qui annule les modifications du commit spécifié

- Commandes courantes :
  - `git revert <commit>` : Crée un nouveau commit inversant les changements du commit spécifié
- Utile pour annuler des modifications dans l'historique sans réécrire l'historique



## git reset

Permet de réinitialiser l'index (staging area) et l'arbre de travail au dernier commit ou à un commit spécifié.

- Types de reset :
  - `git reset --soft <commit>` : Réinitialise l'index au commit spécifié, sans toucher à l'arbre de travail
  - `git reset --mixed <commit>` : Réinitialise l'index et l'arbre de travail au commit spécifié (par défaut)
  - `git reset --hard <commit>` : Réinitialise l'index et l'arbre de travail au commit spécifié, supprimant toutes les modifications

## git rm

Utilisé pour supprimer des fichiers du répertoire de travail et de l'index

- Commandes courantes :
  - `git rm <fichier>` : Supprime le fichier du répertoire de travail et de l'index
  - `git rm --cached <fichier>` : Supprime le fichier de l'index mais le garde dans le répertoire de travail
- Utile pour préparer les suppressions de fichiers pour le prochain commit

## Annuler une action

- Certaines opérations d'annulations sont définitives, il faut donc être prudent avant de les exécuter
- Lorsque l'on souhaite éditer le nom d'un commit ou que l'on a oublié d'y ajouter un fichier, il suffit d'utiliser la commande `git commit --amend`
- Le commit à éditer s'affichera dans l'éditeur configuré sur git

# Désindexer et réinitialiser fichier

- Si un fichier a été indexé par mégarde, il est possible de désindexer avec l'une ou l'autre commande:
  - `git reset HEAD <fichier>`
  - `git restore --staged <fichier>`
- Si l'on souhaite réinitialiser un fichier à l'état du commit précédent il suffit d'utiliser:
  - `git checkout -- <fichier>`
  - `git restore <fichier>`

## Les tags

- Git donne la possibilité d'étiqueter un certain état dans l'historique comme important
- Généralement, les gens utilisent cette fonctionnalité pour marquer les états de publication
- Pour lister les tags de manière croissante: `git tag`
- `git tag -l 'v1.8.5*'` permet de filtrer avec un motif particulier

## Les types de tags

On différencie deux types de tags:

- **Les tags légers:** ressemble à une branche qui ne change pas, c'est un pointeur vers un commit spécifique
- **Les tags annotés:** stockés en tant qu'objet dans git, ils contiennent le nom et l'email de l'auteur, un message et une signature facultative

Il est généralement recommandé d'utiliser les tags annotés

## Créer un tag

- `git tag -a v1.4 -m 'ma version 1.4'` : permet de créer un tag annoté avec l'option `-a`. Le message s'indique avec l'option `-m`
- `git tag v1.4-lg` : créé un tag léger, il suffit d'omettre les paramètres
- `git show 1.4` permet d'afficher le tag précédemment créé
- `git tag -a v1.2 9fceb02` : permet de tagger un ancien commit

## Partager les tags

- Par défaut, la commande git push ne transfère pas les étiquettes vers les serveurs distants
- Il faut explicitement pousser les étiquettes après les avoir créées localement
- `git push origin [nom-du-tag]`
- `git push origin --tags` permet de pousser tous les tags



## Supprimer les tags

- Pour supprimer une étiquette du dépôt local, vous pouvez utiliser `git tag -d <nom-d-etiquette>`
- La suppression à distance se fait avec le paramètre `delete` de cette manière `git push origin --delete <nom-d-etiquette>`

## Les alias

Git permet de définir un alias pour chaque commande en utilisant `git config`.

Voici des exemples d'alias courant:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
```

## 3. Les branches

# Introduction

- Créer une branche signifie **diverger de la ligne principale** de développement et continuer à travailler sans impacter cette ligne
- Git gère les branches de manière **légère** et permet de réaliser les opérations de manière **quasi instantanée** et de basculer entre les branches aussi rapidement
- Git **encourage** des méthodes qui privilégient la **création** et la **fusion** fréquentes de branches

# Fonctionnement des branches

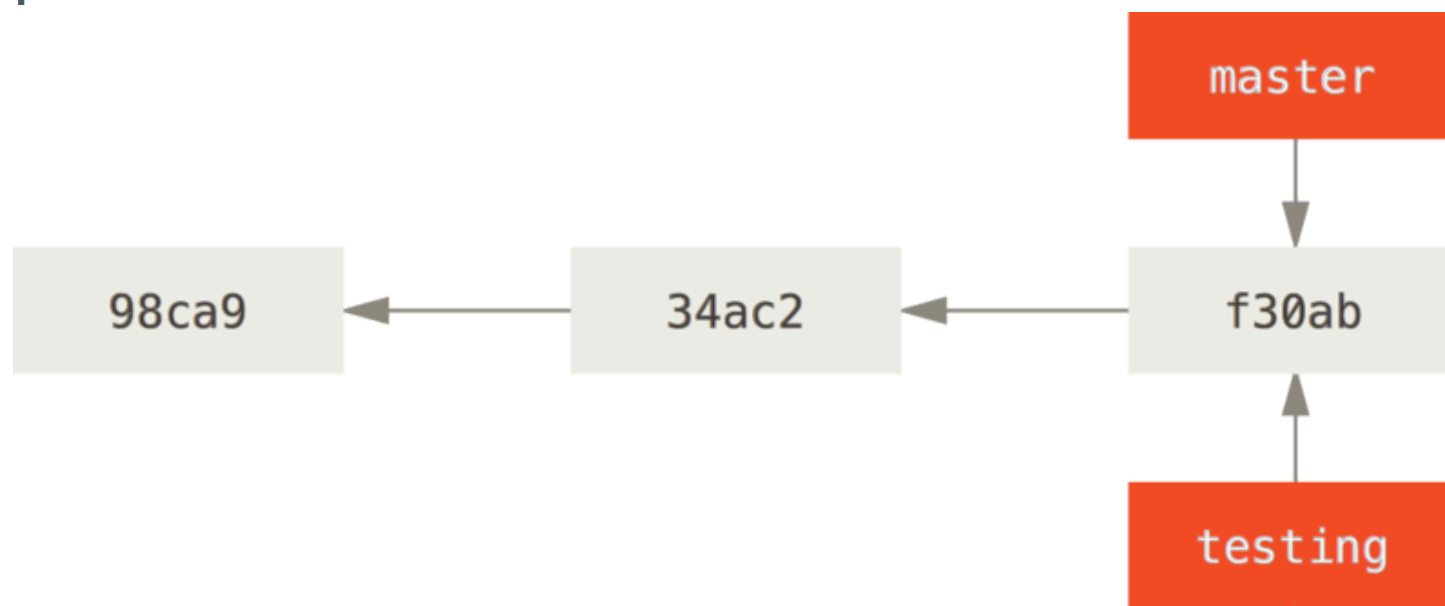
- Les branches Git sont en fait un **pointeur** vers un snapshot de vos modifications.
- Lorsque vous souhaitez **ajouter une nouvelle fonctionnalité** ou **corriger un bogue** - quelle que soit sa taille - vous **créez une nouvelle branche** pour encapsuler vos modifications.
- Il est ainsi plus difficile pour un code instable d'être fusionné dans la base de code principale, et cela vous donne la possibilité de nettoyer l'historique de votre futur avant de le fusionner dans la branche principale.

## Commandes de base

- `git branch` permet de lister l'ensemble des branches du dépôt
- `git branch <branch>` crée une nouvelle branche mais ne se déplace pas sur celle-ci
- `git checkout -b <new-branch>` : créer une branche et s'y déplacer
- `git branch -d <branch>` supprime la branche de manière sécurisée
- `git branch -D <branch>` force la suppression de la branche
- `git branch -m <branch>` renomme la branche actuelle
- `git branch -a` liste les branches distantes

## Création de branche

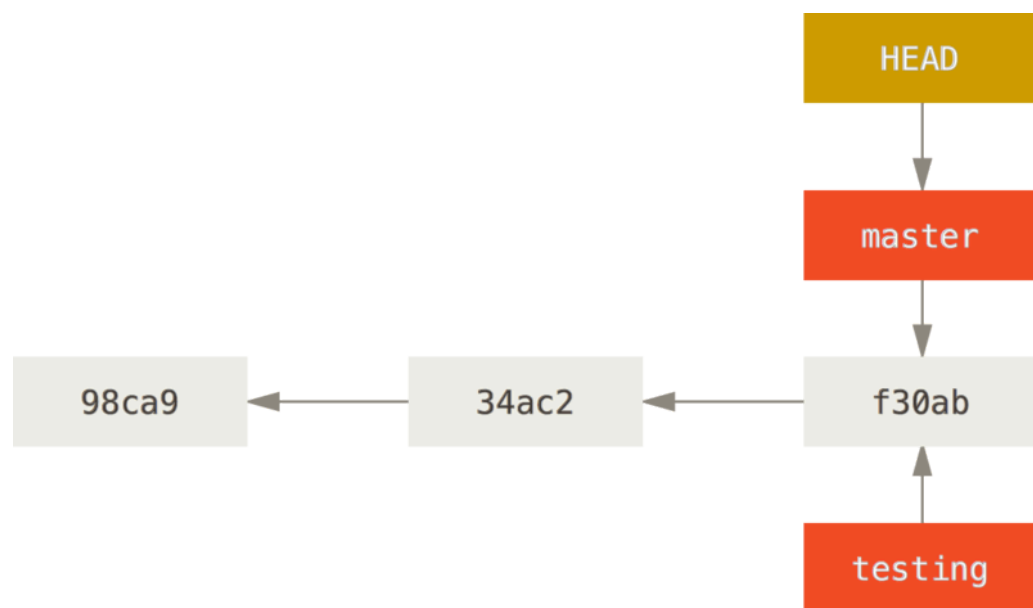
`git branch testing` créé une nouvelle branche, c'est à dire un pointeur sur le commit actuel.



# HEAD

Git conserve la position de la branche sur laquelle on se situe à l'aide d'un pointeur spécial appelé **HEAD**.

Il s'agit simplement d'un pointeur sur la branche locale où l'on se situe.





## Connaître la position du head

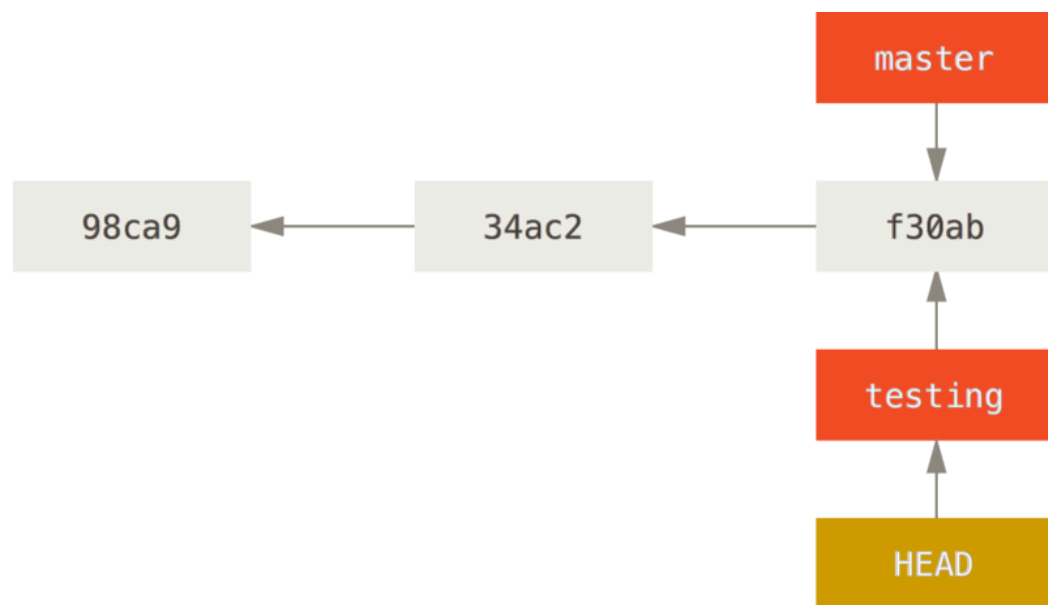
A l'aide de la commande `git log` via l'option `--decorate` on peut facilement savoir où se situe le pointeur `HEAD`

```
$ git log --oneline --decorate
f30ab (HEAD, master, test) add feature #32 - ability to add new
34ac2 fixed bug #ch1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Ici on peut voir que le `HEAD` se situe sur le commit `f30ab` au même titre que les branches `master` et `test`

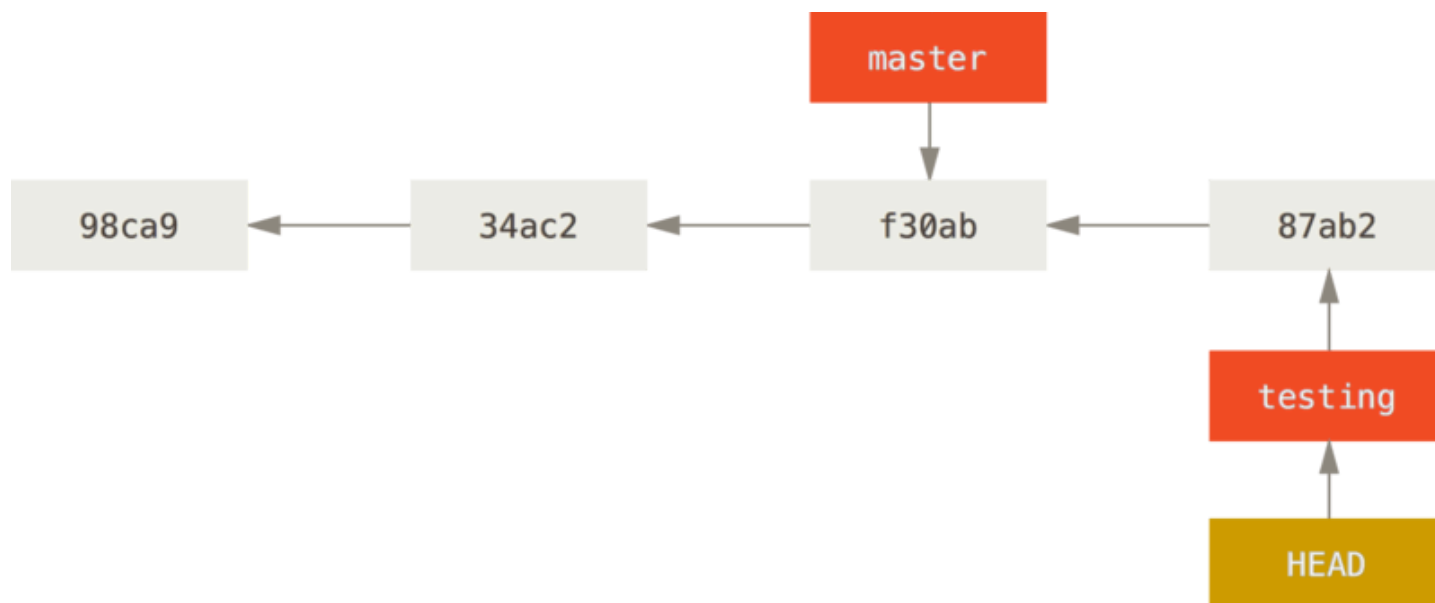
## Basculer de branches

- Pour basculer sur une branche existante, il suffit de lancer la commande `git checkout`.
- En exécutant `git checkout testing` on obtient:



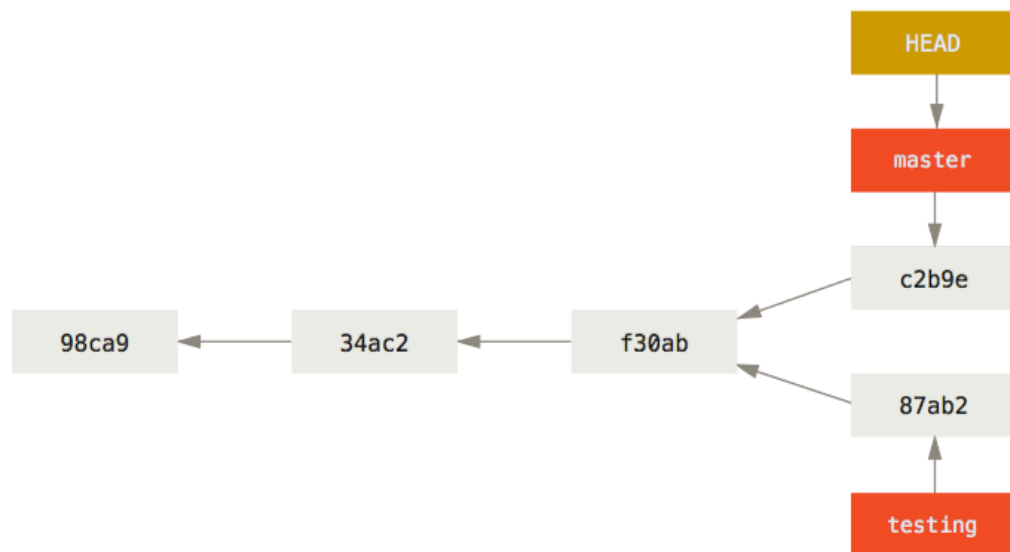
## Créer un commit sur une branche

Lorsque l'on crée un nouveau commit sur une branche, celle-ci se déplace depuis le commit d'origine en gardant le même historique. Les autres branches elles, ne changent pas de position.



## Divergence d'historique

Si l'on réalise un commit sur la branche master, on se retrouve avec des divergences d'historiques. Il faudra donc finir par fusionner les deux branches pour garder un historique commun.

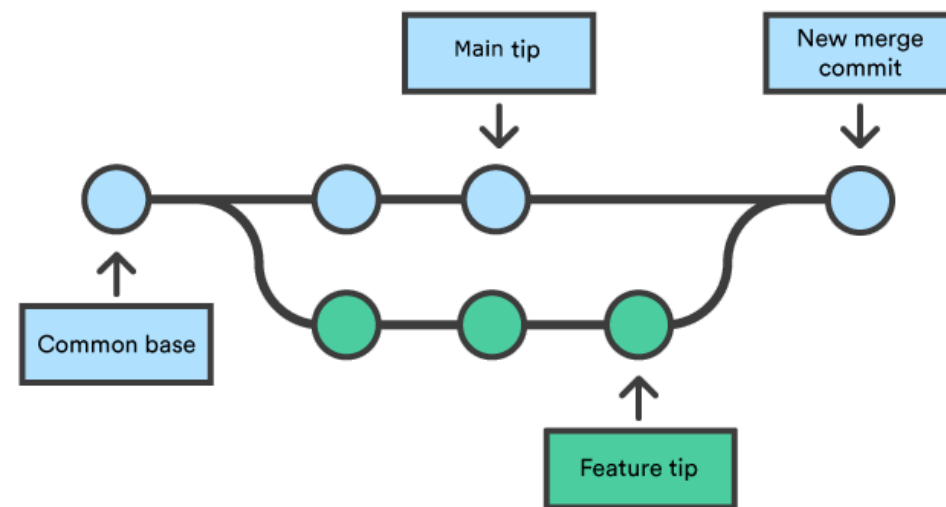
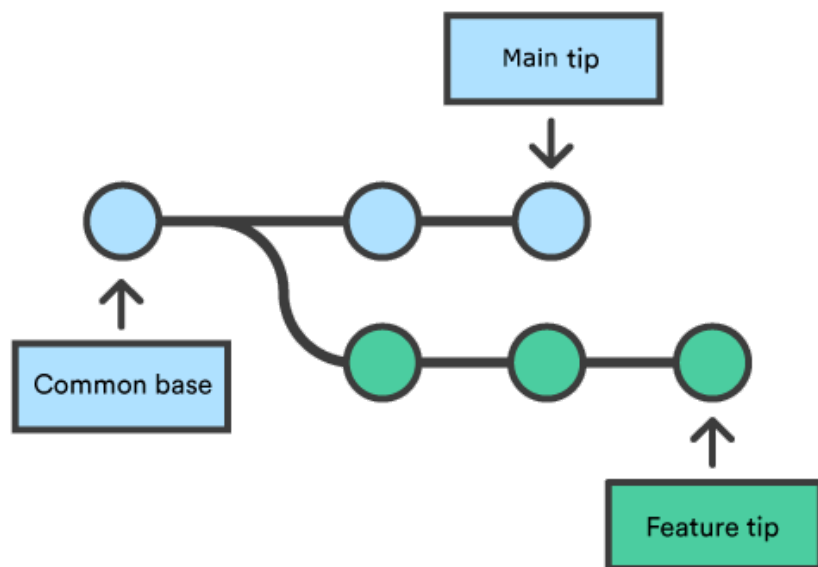


## Fusion de branches

- La fusion est le moyen utilisé par Git pour reconstituer un historique forké.
- La commande `git merge` permet de prendre des historiques de développement indépendants créés sur une branche pour les intégrer dans une seule branche.
- Pour fusionner une branche, on se place sur la branche destinataire et on fusionne la branche de destinateur.
- Exemple: `git merge feature/navbar`

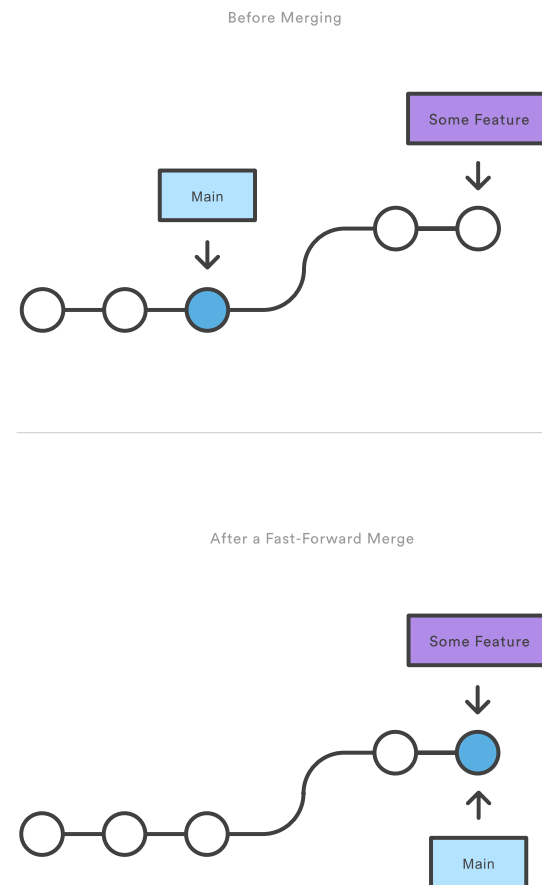
# Comment fonctionne les merge

Lorsque l'on fusionne deux branches, git tente automatiquement de fusionner les deux historiques.



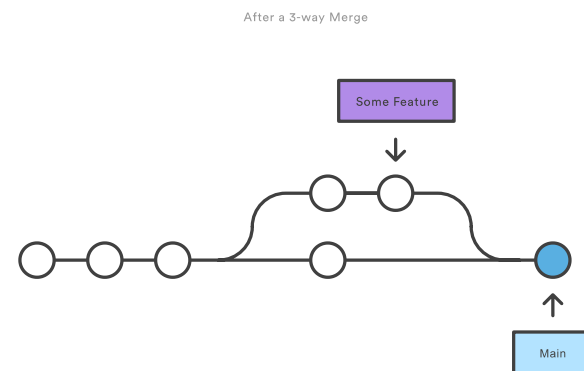
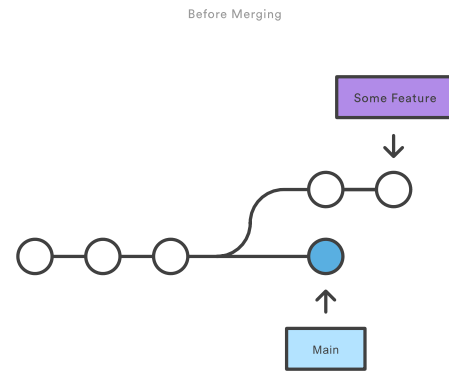
# Merge Fast Forward

- Le **merge Fast Forward (FF)** se produit lorsque la branche de base vers laquelle on fusionne n'a pas reçu de nouveaux commits depuis la dernière mise à jour de la branche que l'on souhaite fusionner
- Au lieu de créer un commit de fusion, git déplace le pointeur vers l'avant



# Merge de branches divergentes

- Lorsqu'il n'y a pas de chemin linéaire vers la branche cible, Git n'a pas d'autre choix que de les combiner via une fusion à 3 voies
- Les fusions à 3 voies utilisent un commit dédié pour relier les deux historiques





## Résolution de conflits

Si les deux branches que vous essayez de fusionner ont toutes deux modifié la même partie du même fichier, **Git ne sera pas en mesure de déterminer quelle version utiliser.**

Lorsqu'une telle situation se produit, Git s'arrête juste avant le commit de fusion afin que vous puissiez **résoudre les conflits manuellement.**

```
On branch main
Unmerged paths:
(use "git add/rm ..." as appropriate to mark resolution)
both modified: hello.py
```

## Présentation d'un conflit

Lorsque Git rencontre un conflit lors d'une fusion, il modifie le contenu des fichiers affectés avec des indicateurs visuels qui marquent les deux côtés du contenu conflictuel. Ces indicateurs visuels sont les suivants : <<<<<<, =====, et >>>>>>.

```
here is some content not affected by the conflict
<<<<<< main
this is conflicted text from main
=====
this is conflicted text from feature branch
>>>>>> feature branch;
```

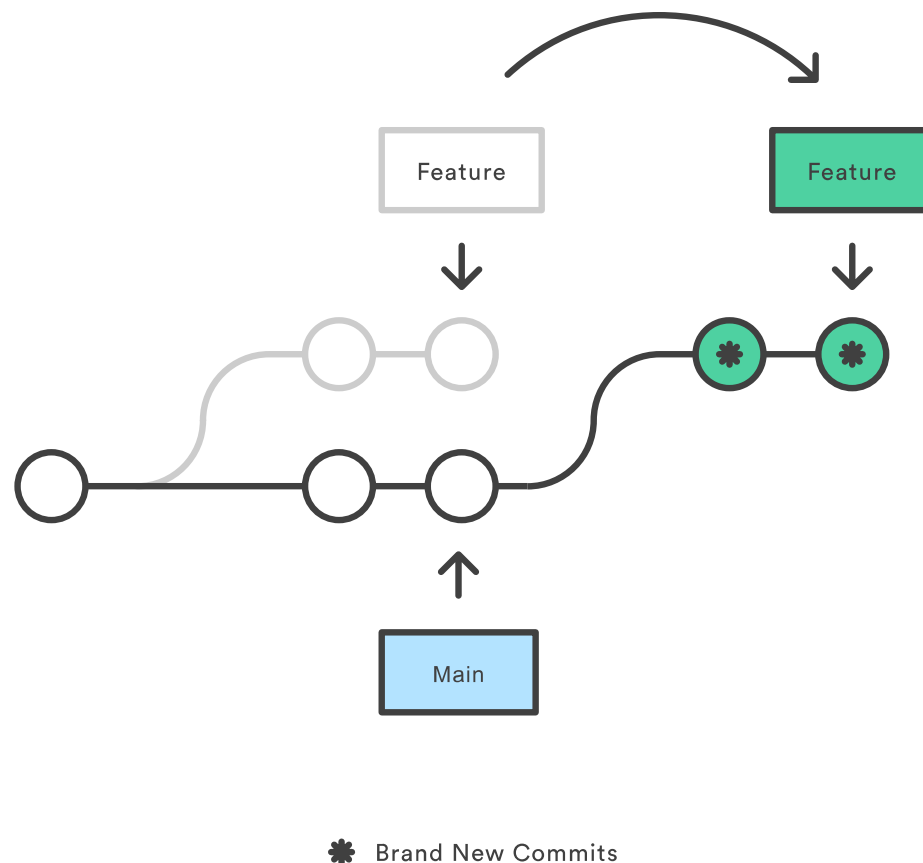
## Résolution du conflit

- L'utilisation de `git status` permet de connaître les éléments en conflits
- Une fois le(s) fichier(s) corrigés, il suffit de créer un nouveau commit pour corriger les problèmes liés au merge
- La plupart des IDE fournissent des outils graphiques pour résoudre les conflits

## Git rebase

- Le rebase est le processus de **déplacement** ou de combinaison d'**une séquence de commits** vers un nouveau commit de base
- Du point de vue du contenu, le rebase consiste à **changer la base d'une branche d'un commit à un autre**, ce qui donne l'impression que l'on créé une branche à partir d'un commit différent
- En interne, Git accomplit cela en **créant de nouveaux commits** et en les appliquant à la base spécifiée

# Visuel git rebase



# Manipulation git rebase

```
git checkout feature  
git rebase main
```

Ceci déplace la branche entière de la `feature` pour commencer à l'extrémité de la branche `main`, incorporant effectivement tous les nouveaux commits dans la branche `main`.

Mais, au lieu d'utiliser un commit de **merge**, **rebase** réécrit l'historique du projet en créant de nouveaux commits pour chaque commit de la branche d'origine.

## La règle d'or du rebase

La règle d'or de `git rebase` est de ne **jamais** l'utiliser sur les **branches publiques**.

Posez-vous toujours la question suivante : "*Est-ce que quelqu'un d'autre regarde cette branche ?*"

Si la réponse est oui, enlevez vos mains du clavier et commencez à réfléchir à un moyen non destructif d'effectuer vos modifications (par exemple, la commande `git revert`)

# Merge vs Rebase

## Rebase

- Historique propre et linéaire, sans commits de fusion inutiles

## Merge

- Historique complet du projet en évitant le risque de réécrire les commits publics



## 4. Synchronisation à distance

## Travailler avec un dépôt distant

- Les dépôts distants sont des versions d'un projet qui sont hébergées sur Internet ou le réseau d'entreprise.
- Gérer des dépôts distants inclut savoir comment **ajouter des dépôts distants**, **effacer des dépôts distants** qui ne sont plus valides, **gérer des branches distantes** et les définir comme suivies ou non, et plus encore.

# Afficher des dépôts distants

- Pour visualiser les serveurs distants que enregistrés, il suffit d'exécuter la commande `git remote`
- En général lorsque l'on clone un dépôt, on dispose au moins d'une référence nommée `origin`
- `git remote -v` permet d'afficher le nom ainsi que l'URL des dépôts

## Ajouter un dépôt distant

- Lors de la récréation d'un référence git, il est possible d'ajouter l'origin d'un dépôt distant avec la commande:
  - `git remote add [nomcourt] [url]`

# Récupérer les données distantes

- `git fetch [remote-name]` permet de récupérer toutes les références de toutes les branches du dépôt
- `fetch` récupère les données dans le dépôt local mais sous sa propre branche, sans les fusionner
- `git pull` récupère et fusionne automatiquement une branche distante dans la branche locale

## Git fetch

- La commande `git fetch` télécharge **les commits, les fichiers et les refs** d'un dépôt distant dans le dépôt local
- `git fetch` télécharge le contenu distant mais ne met pas à jour l'état de travail du repo local, laissant le travail actuel intact
- `git fetch --all`: récupère les informations de tous les dépôts distants
- Pour mettre à jour les changements il faudra effectuer `git merge origin/main` dans la branche locale

## Git pull

- `git pull` intègre les modifications d'un dépôt distant dans la branche actuelle
- Dans son mode par défaut, `git pull` est l'abréviation de `git fetch` suivi de `git merge FETCH_HEAD`
- Avec l'option `--rebase`, il lance `git rebase` au lieu de `git merge`

## Envoyer les données sur le dépôt distant

- `git push <nom-distant> <nom-de-branche>` permet de pousser les modifications locales sur le dépôt distant
- Si des modifications ont été effectuées sur la branche entre temps il faudra d'abord les récupérer et les intégrer



# Inspecter et éditer un dépôt distant

- `git remote show [nom-distant]` permet de visualiser plus d'informations à propos d'un dépôt distant
- `git remote rename [old] [new]` pour modifier le nom court d'un dépôt distant
- `git remote rm [nom]` Pour supprimer un dépôt distant du registre

## 5. Les outils de Git

## Git stash

- `git stash` met temporairement de côté (ou cache) les modifications apportées à votre copie de travail afin de les appliquer à nouveau plus tard
- La mise en cache est pratique si vous avez besoin de changer rapidement de contexte (branche) et de travailler sur autre chose, mais que vous êtes à mi-chemin d'une modification de code et que vous n'êtes pas tout à fait prêt à la valider

## Commandes git stash

- `git stash` : Enregistre l'état actuel du répertoire de travail et de l'index, puis rétablit le répertoire de travail pour correspondre au commit HEAD
- `git stash list` : Affiche la liste des stashes existants
- `git stash show` : Montre la différence du stash
- `git stash save <message>` : Stashe les modifications avec un message personnalisé
- `git stash pop` : supprime le dernier stash et l'applique
- `git stash apply` : applique le dernier stash

## Git cherry-pick

- `git cherry-pick` est une commande puissante qui permet à des commits d'être pris par référence et ajoutés au `HEAD` de travail actuel
- Le cherry picking est l'action de choisir un commit d'une branche et de l'appliquer à une autre
- `git cherry-pick` peut être utile pour annuler des changements
- Dans la plupart des cas, l'utilisation de merge est à préférer

# Utilisation de git cherry-pick

```
a - b - c - d    Main
      \
      e - f - g  Feature
```

On souhaite appliquer les modifications du commit '**f**' de la branche **feature** à la branche **main**: `git cherry-pick f`.

```
a - b - c - d - f    Main
      \
      e - f - g  Feature
```

La branche **main** dispose désormais uniquement des modifications du commit **f**.

## 6. Git workflow

# Introduction aux workflows

- Un workflow est comme une recette ou une recommandation sur la façon d'utiliser Git pour accomplir un travail de manière cohérente et productive
- Les workflows encouragent les développeurs et les équipes DevOps à utiliser Git de manière efficace et cohérente
- Ici nous allons présenter deux workflows:
  1. Gitflow
  2. Trunk-based development



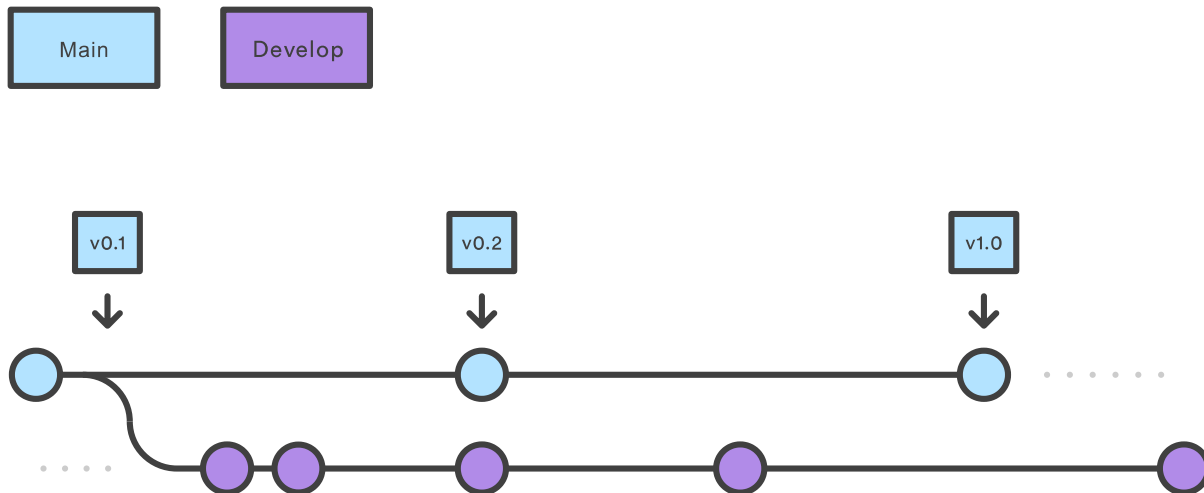
# Gitflow: Introduction

- Gitflow est un modèle qui implique l'utilisation de **branches de fonctionnalités** et de multiples **branches primaires**
- Il a été publié pour la première fois et popularisé par Vincent Driessen sur le site [nvie](#)
- Cette méthodologie attribue des **rôles très spécifiques aux différentes** branches et définit comment et quand elles doivent interagir

# Gitflow: Main et Develop

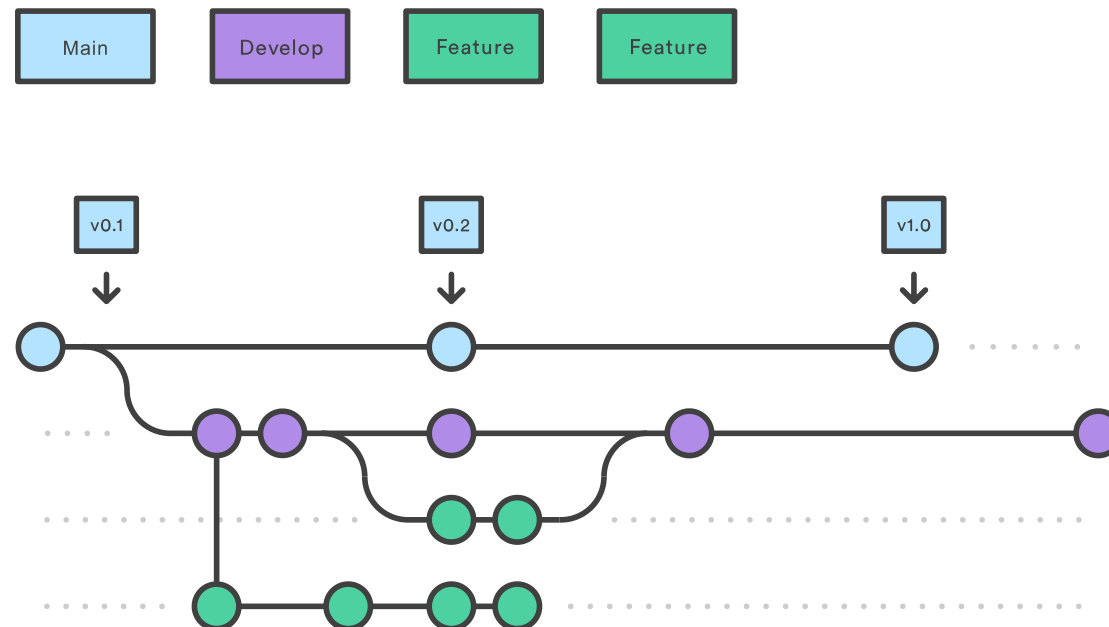
Gitflow utilise deux branches pour enregistrer l'historique du projet:

- La branche **main** stocke l'historique de la version officielle
- la branche **develop** sert de branche d'intégration pour les fonctionnalités



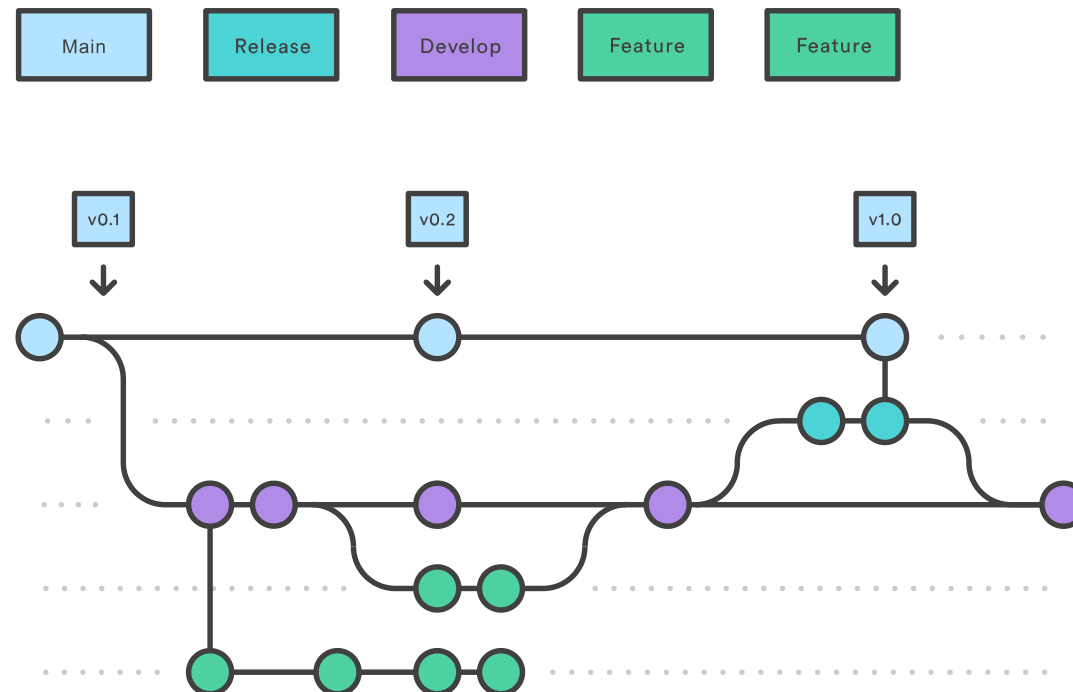
# Gitflow: Feature

- Chaque **feature** réside sur sa propre branche, qui peut être poussée vers le dépôt central
- les branches de **feature** utilisent **develop** comme branche parente
- Lorsqu'une fonctionnalité est terminée, elle est fusionnée dans **develop**



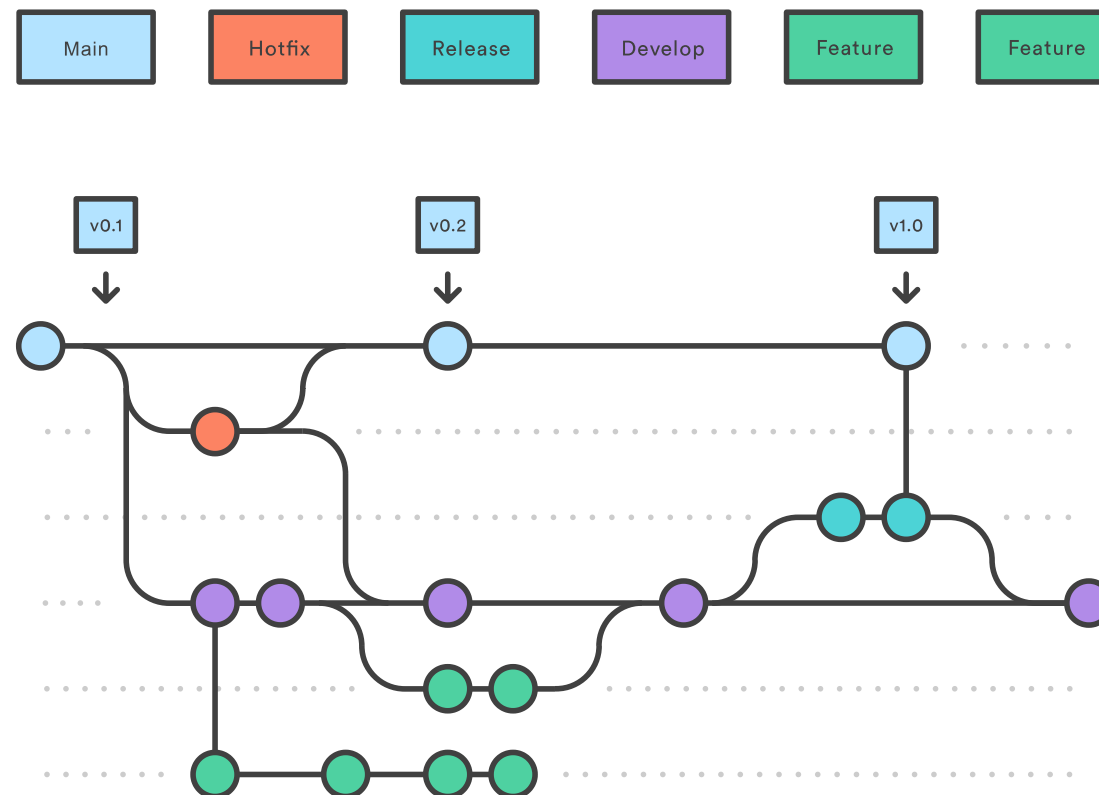
# Gitflow: Release

- Créée lorsque la branche `develop` est complète en termes de fonctionnalités
- On y ajoute uniquement des correctifs de bugs ou des modifications de configuration
- Une fois prête, elle est fusionnée dans la branche `main` avec un numéro de version attribué



# Gitflow: Hotfix

- Utilisée pour les correctifs d'urgence ou les correctifs critiques dans le code de production
- Créée à partir de `main`
- Une fois terminée elle est fusionnée à la fois dans `main` et `develop`



## Gitflow: utilitaire

- L'installation de git se fait avec l'utilitaire `git-flow` qui permet la mise en place rapide de Gitflow
- Une documentation est disponible à cette [adresse](#)
- Git-flow est une solution basée sur les fusions (merge). Elle n'effectue pas de rebase sur les branches de fonctionnalités

## Trunk-based development: principe

- Le trunk-base development est un workflow où les développeurs collaborent sur le code dans **une seule branche** appelée "*tronc*"
- Elle repose sur le fait d'éviter de créer d'autres branches de développement de longue durée
- Elle permet de se focaliser sur l'intégration continue et les livraisons fréquentes

## Trunk-based development: histoire

- Le trunk-based development fait référence à un arbre dont le tronc est plus développé que ses branches
- Il s'agit d'un workflow apparu en **1980** puis popularisé dans les années 1990 par des entreprises Google et Facebook
- C'est un modèle majoritairement favorisé par les pratiques **DevOps**



# Trunk-based development: petite équipe



Dans ce workflow, chaque commiter (généralement un duo en pair programming) effectue de petits commits directement dans le tronc (ou la main) avec une étape de pré-intégration qui effectue une phase compilation.

# Trunk-based development: grande échelle

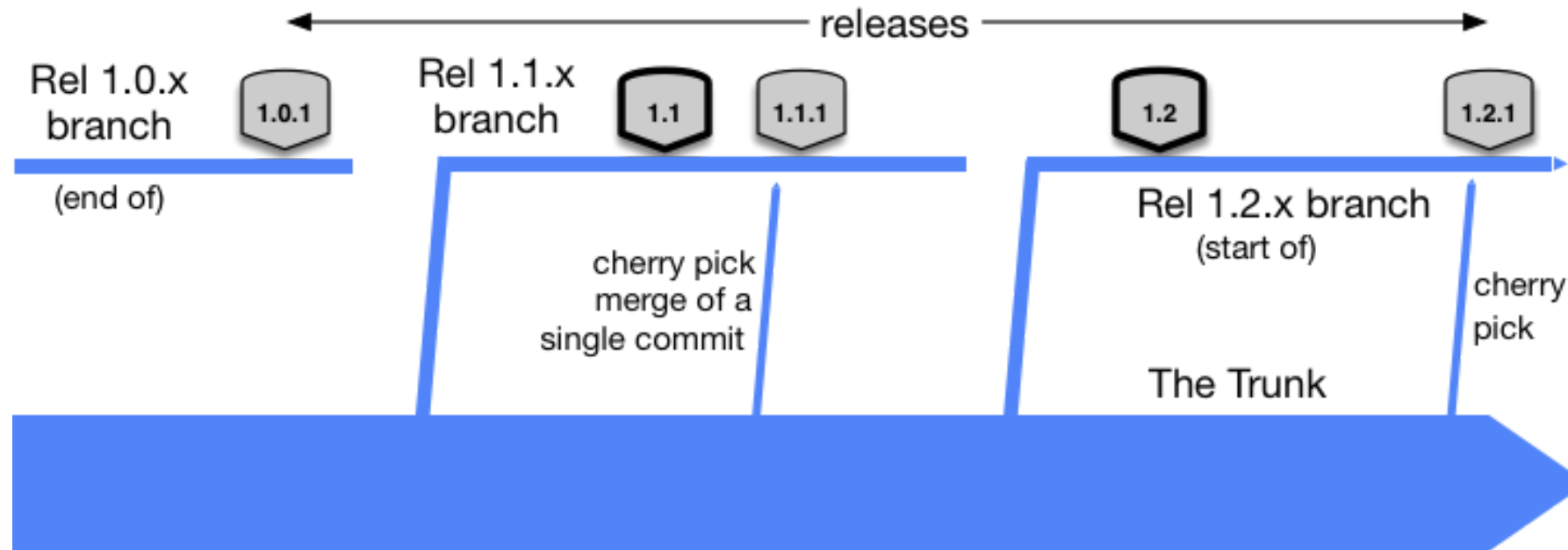


A grande échelle on réalise des branches de fonctionnalités de courte durée : une personne sur quelques jours (max) et passant par une **revue de code** de type **Pull-Request** et une automatisation de build avant "d'intégrer" (fusionner) dans le tronc (ou main).

## Trunk-based development: release

- Si une équipe met en place des versions de production mensuelles, elle devra également mettre en place des versions de correction de bogues entre les versions planifiées
- Pour faciliter cela, il est courant pour les équipes de développement créé une branche de release juste à temps, par exemple quelques jours avant la sortie
- Cette branche devient un endroit stable, étant donné que les développeurs continuent d'envoyer leurs modifications dans le tronc régulièrement

# Trunk-based development: release



**Merci pour votre attention**

**Des questions ?**