

Formation Git Collaboratif Branch'n'Merge

<http://www.math.u-bordeaux.fr/~lfacq/GitBranchMerge.pdf>

©2024 Laurent FACQ - CNRS - IMB UMR 5251

Février 2024 v0.9

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de l'auteur et de son institution d'origine continuent à y figurer, de même que le présent texte.

2 pauses de 5 minutes !

- ▶ pause à $H+1$, reprise à $H+1:05$
- ▶ pause à $H+2$, reprise à $H+2:05$

à vous de revenir à l'heure !

Objectifs de cette session

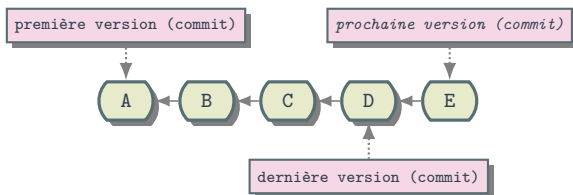
- ▶ comprendre en profondeur et maîtriser la notion de **branche** pour pouvoir travailler à plusieurs sur un projet
- ▶ nous verrons régulièrement les aspects implémentation car cela aide à comprendre précisément, et à démystifier, le fonctionnement des branches
- ▶ nous allons volontairement présenter des éléments partiels pour que cela reste simple pour une première approche.
 - ▶ `!/` il existe plusieurs façons de faire les mêmes choses, nous n'en verrons qu'une
 - ▶ `!/` les commandes git ont beaucoup d'options, nous ne verrons que quelques formes
 - ▶ `!/` nous ne parlerons pas de rebase

Pourquoi les branches ? à quoi ça sert ? (usage)

- ▶ les branches permettent de faire évoluer le projet dans plusieurs directions en même temps.
 - ▶ pour explorer diverses variantes
 - ▶ pour tester des évolutions possibles
 - ▶ pour travailler à plusieurs sans se gêner (chacun sa/ses branche(s))
 - ▶ pour maintenir plusieurs versions [publiques] du projet
 - ▶ ...
- ▶ ... et de reporter facilement tout ou partie des modifications apportées à une branche sur une autre branche
- ▶ toutes ces versions cohabitent en même temps
- ▶ vous pouvez aller d'une branche à l'autre à tout moment

Rappel sur le chaînage des versions (commits)

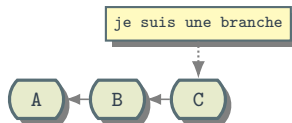
- ▶ chaque version pointe vers sa version parente :
ex: ici, 4 versions (commits) A,B,C,D
B pointe vers son père A, C vers B, ...
la prochaine version E pointera vers D



- ▶ jusqu'ici vous avez probablement travaillé sur une seule "chaîne" de version, ou *branche*.

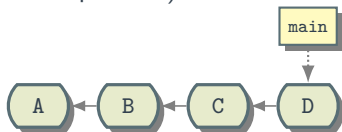
Qu'est ce qu'une branche ?

- ▶ une branche est juste un pointeur (une étiquette) vers une version (commit), version qui est elle même dans une chaîne de versions (commits), contenu dans un dépôt



Qu'est-ce qu'une branche ? (fonctionnement)

- ▶ une **branche** est juste une **étiquette** qui **pointe** sur un **commit** et qui **avance automatiquement** au gré des commits pour toujours pointer sur le dernier commit ajouté à cette branche
- ▶ la branche par défaut s'appelle généralement **main** (ou **master** historiquement)

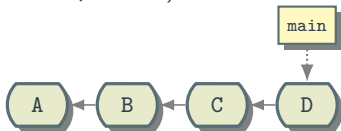


après un nouveau commit (E) en suivant :

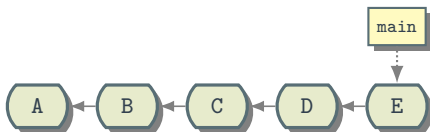


Qu'est-ce qu'une branche ? (fonctionnement)

- ▶ une **branche** est juste une **étiquette** qui **pointe** sur un **commit** et qui **avance automatiquement** au gré des commits pour toujours pointer sur le dernier commit ajouté à cette branche
- ▶ la branche par défaut s'appelle généralement **main** (ou **master** historiquement)

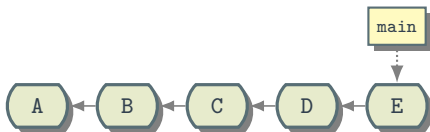


après un nouveau commit (E) en suivant :



Sous le capot : implémentation d'une branche

- ▶ techniquement, une branche est juste **un fichier texte contenant la référence (hash) d'un commit**



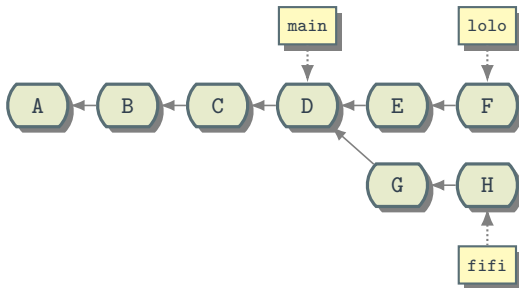
ex: pour la branche main

- Le Fichier `.git/refs/heads/main` contient

`"216ee559a85015e43b623bb11fc5fbc108ce3c77"` (E)

Et avec plusieurs branches ?

- ▶ on peut en avoir plusieurs branches "en parallèle"
- ▶ ex. avec 2 personnes :



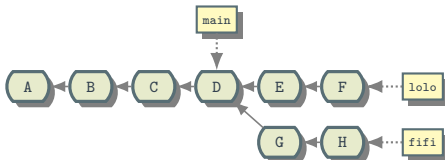
3 branches : main, fifi, lolo

- ▶ chacun peut travailler en autonomie dans sa branche personnelle, commiter autant qu'il le souhaite dans sa branche, puis fusionner (reporter les modifications), à posteriori, dans une autre branche quand les modifications sont mûres

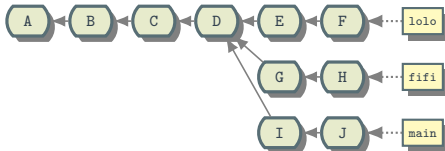
ici on peut dire que les branches **fifi** et **lolo** sont en avance sur **main**

Branche en avance ?

- ici on peut dire que les branches **fifi** et **lolo** sont en avance sur **main**

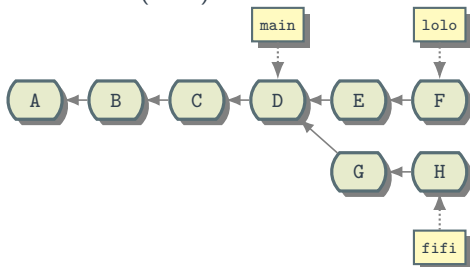


- elles sont dans le prolongement direct de la branche **main**
 - autrement dit : tous les ancêtres de **main** sont des ancêtres de **fifi** et **lolo**
- ici, en revanche, aucune branche n'est en avance sur une autre :



Sous le capot : implémentation des branches

- ▶ rappel : techniquement, une branche est juste un fichier contenant la référence (hash) d'un commit



- ex:
- Le Fichier `.git/refs/heads/main` contient
"216ee559a85015e43b623bb11fc5fbc108ce3c77" (D)
 - Le Fichier `.git/refs/heads/fifi` contient
"5d4359b12ed5a7f306e82ba30663b629bb74b818" (H)
 - Le Fichier `.git/refs/heads/lolo` contient
"2ded0fa9d313a93179c84d5e818a5336c9fdeb01" (F)

Comment savoir sur quelle branche nous sommes ?

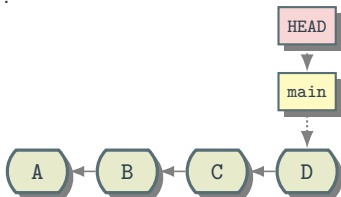
- ▶ comme il peut y avoir plusieurs branches dans notre dépôt/projet
- ▶ il faut un moyen pour mémoriser sur quelle branche ou quel commit nous sommes en train de travailler
- ▶ c.a.d savoir *quel commit de quelle branche* a été initialement recopiée dans notre répertoire courant et notre index

- ▶ c'est à cet endroit que sera naturellement inséré le prochain commit

- ▶ c'est la raison d'être de **HEAD**...

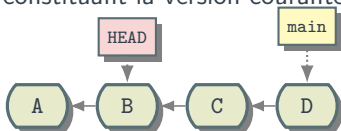
HEAD : la référence sur la branche/commit courant 1/2

- ▶ **HEAD** est une référence, *dans votre dépôt local*, pointant toujours (indirectement ou directement) sur le commit courant qui a été initialement recopié "dans le répertoire de travail et dans l'index"
- ▶ c'est à l'endroit pointé par **HEAD** que sera naturellement inséré le prochain commit
- ▶ **cas 1** (classique) : **HEAD** → **branche** → **commit (hash)**
la plus part du temps, **HEAD** pointe sur la branche courante (sur laquelle vous travaillez) qui pointe elle même sur son dernier commit :



HEAD : la référence sur la branche/commit courant 2/2

- ▶ **HEAD** est une référence - *dans votre dépôt local* - pointant toujours (indirectement ou directement) sur le commit courant qui a été initialement recopié "dans le répertoire de travail et dans l'index"
- ▶ c'est à l'endroit pointé par **HEAD** que sera naturellement ajouté le prochain commit
- ▶ **cas 2 (detached HEAD) : HEAD → commit (hash)**
HEAD peut aussi être dans l'état **détachée** - c.a.d détachée de toute branche - et pointer directement sur un commit (hash) constituant la version courante



- ▶ !! dans ce cas, en l'état, on peut regarder les fichiers mais on ne peut pas directement enregistrer des modifications (commits) car il n'y a pas de branche pour les recevoir

Sous le capot : implémentation de HEAD

- ▶ HEAD, comme les branches, est juste un fichier contenant la référence à une branche ou à un commit (hash)

- ▶ **cas 1 : HEAD → branche → commit (hash)**



ex: `git checkout main` # se placer sur la branche main

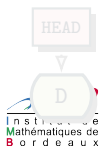
- Fichier `.git/HEAD` contient
"`ref: refs/heads/main`"

- Fichier `.git/refs/heads/main` contient
"`216ee559a85015e43b623bb11fc5fbc108ce3c77`"

- ▶ **cas 2 : HEAD → commit (hash) (*detached HEAD*)**

`git checkout 216ee559` # se placer sur le commit 216ee559

- Fichier `.git/HEAD` contient
"`216ee559a85015e43b623bb11fc5fbc108ce3c77`"



Sous le capot : implémentation de HEAD

- ▶ HEAD, comme les branches, est juste un fichier contenant la référence à une branche ou à un commit (hash)

- ▶ **cas 1 : HEAD → branche → commit (hash)**



ex: `git checkout main` # se placer sur la branche main



- Fichier `.git/HEAD` contient `"ref: refs/heads/main"`



- Fichier `.git/refs/heads/main` contient `"216ee559a85015e43b623bb11fc5fbc108ce3c77"`

- ▶ **cas 2 : HEAD → commit (hash) (*detached HEAD*)**

`git checkout 216ee559` # se placer sur le commit 216ee559



- Fichier `.git/HEAD` contient `"216ee559a85015e43b623bb11fc5fbc108ce3c77"`

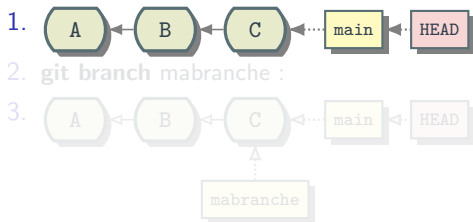
Comment créer une branche : 3 cas

- ▶ en 2 étapes : créer une branche, puis se positionner dessus
 - ▶ en 1 étape : créer une branche ET se positionner directement dessus (le plus pratiqué)
 - ▶ créer une branche sur un commit quelconque i.e. depuis une situation detached head (très similaire aux cas précédents)
-
- * dans les 3 cas on va créer la branche à l'endroit où nous sommes (comportement par défaut)
 - * "se positionner dessus" = faire pointer HEAD dessus = faire de cette branche la branche courante
 - * note: répertoire de travail et index restent inchangés car on est toujours sur le même commit

en 2 étapes : Créer - puis aller - sur une branche

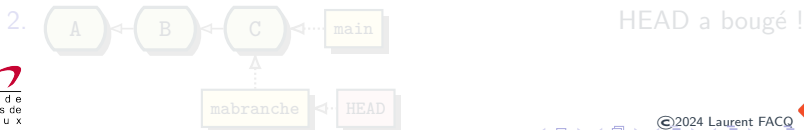
► Créer une branche : **git branch NOMBRANCHE**

- crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit



► Aller sur la branche : **git checkout NOMBRANCHE**

1. `git checkout mabranche :`



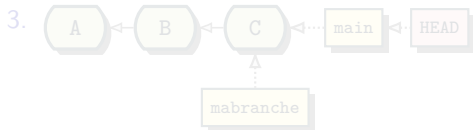
en 2 étapes : Créer - puis aller - sur une branche

▶ Créer une branche : **git branch NOMBRANCHE**

- ▶ crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit

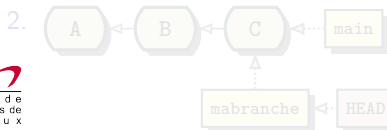


2. **git branch mabranche** :



▶ Aller sur la branche : **git checkout NOMBRANCHE**

1. **git checkout mabranche** :

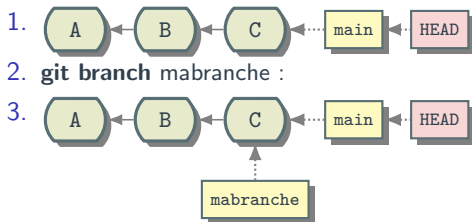


HEAD a bougé !

en 2 étapes : Créer - puis aller - sur une branche

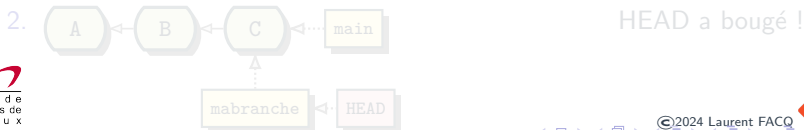
► Créer une branche : **git branch NOMBRANCHE**

- crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit



► Aller sur la branche : **git checkout NOMBRANCHE**

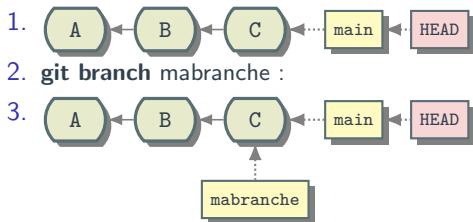
1. **git checkout mabranche :**



en 2 étapes : Créer - puis aller - sur une branche

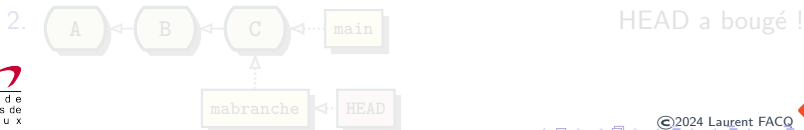
► Créer une branche : **git branch NOMBRANCHE**

- crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit



► Aller sur la branche : **git checkout NOMBRANCHE**

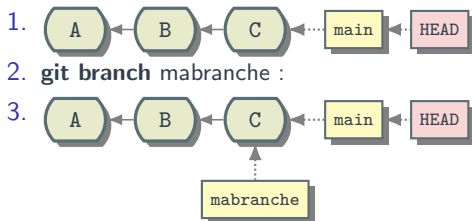
1. **git checkout mabranche :**



en 2 étapes : Créer - puis aller - sur une branche

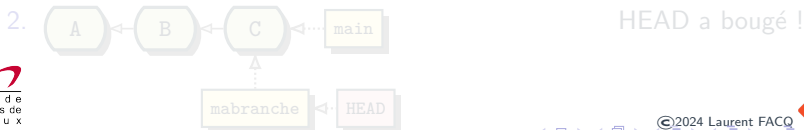
► Créer une branche : **git branch NOMBRANCHE**

- crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit



► Aller sur la branche : **git checkout NOMBRANCHE**

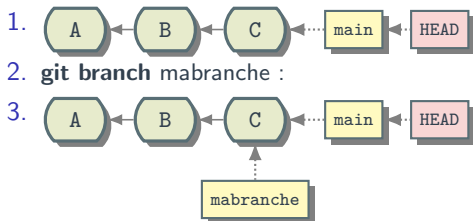
1. **git checkout mabranche :**



en 2 étapes : Créer - puis aller - sur une branche

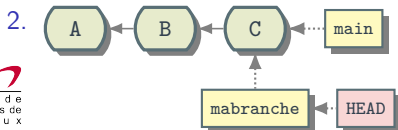
► Créer une branche : **git branch NOMBRANCHE**

- crée une branche "à l'endroit où nous sommes" (HEAD), sans bouger. *NOMBRANCHE* : une nouvelle branche qui pointe sur le même commit



► Aller sur la branche : **git checkout NOMBRANCHE**

1. **git checkout mabranche :**



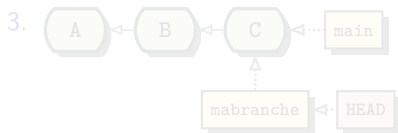
HEAD a bougé !

en une étape : Créer ET aller sur une branche

- ▶ généralement on fait les 2 opérations en une seule étape
- ▶ "on crée une branche et on va dessus pour travailler"
- ▶ Créer et aller sur une branche : **git checkout -b NOMBRANCHE**



2. git checkout -b mabranche :



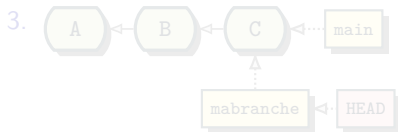
HEAD pointe déjà sur la nouvelle branche "mabranche"

en une étape : Créer ET aller sur une branche

- ▶ généralement on fait les 2 opérations en une seule étape
- ▶ "on crée une branche et on va dessus pour travailler"
- ▶ Créer et aller sur une branche : **git checkout -b NOMBRANCHE**



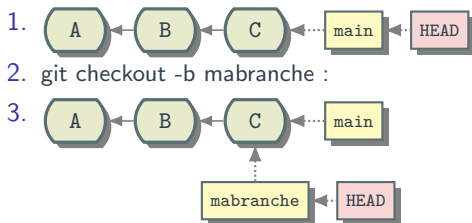
2. git checkout -b mabranche :



HEAD pointe déjà sur la nouvelle branche "mabranche"

en une étape : Créer ET aller sur une branche

- ▶ généralement on fait les 2 opérations en une seule étape
- ▶ "on crée une branche et on va dessus pour travailler"
- ▶ Créer et aller sur une branche : **git checkout -b NOMBRANCHE**



HEAD pointe déjà sur la nouvelle branche "mabranche"

Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (`git checkout HASHB`)
- ▶ Créer une branche : **`git branch NOMBRANCHE`**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

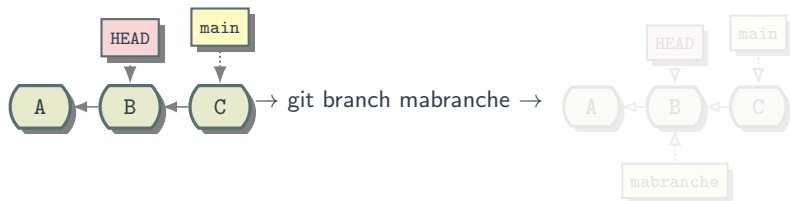


- ▶ Aller sur la branche : **`git checkout NOMBRANCHE`**



Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (`git checkout HASHB`)
- ▶ Créer une branche : **`git branch NOMBRANCHE`**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

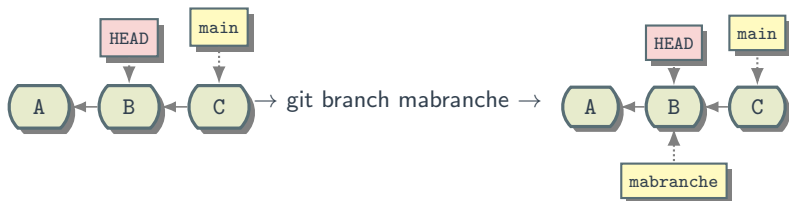


- ▶ Aller sur la branche : `git checkout NOMBRANCHE`



Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer une branche : **git branch NOMBRANCHE**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

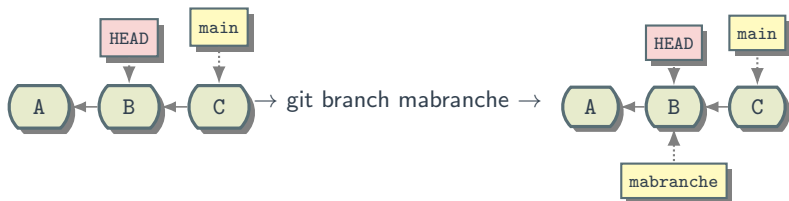


- ▶ Aller sur la branche : **git checkout NOMBRANCHE**

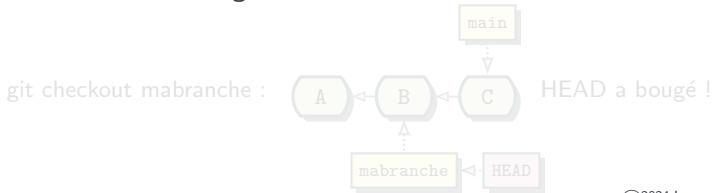


Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer une branche : **git branch NOMBRANCHE**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

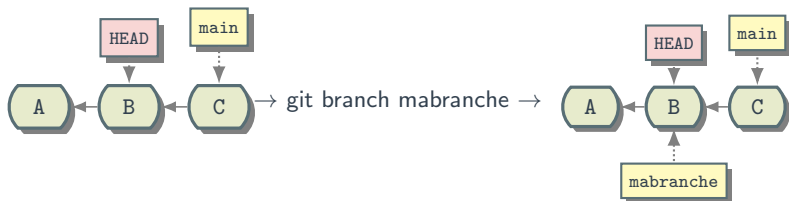


- ▶ Aller sur la branche : **git checkout NOMBRANCHE**

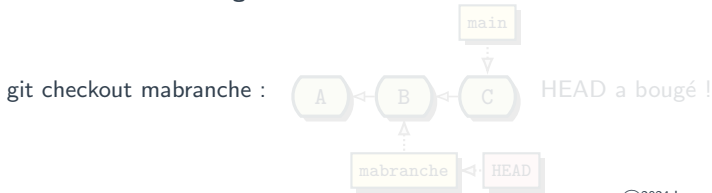


Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer une branche : **git branch NOMBRANCHE**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

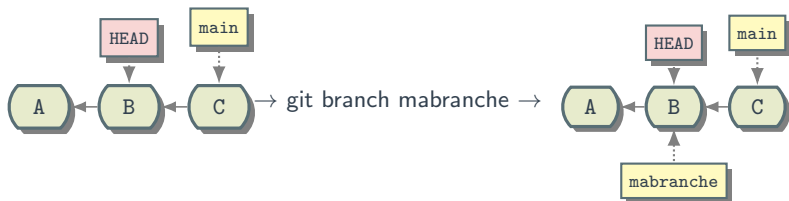


- ▶ Aller sur la branche : **git checkout NOMBRANCHE**

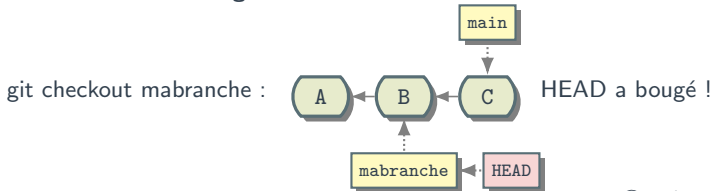


Créer -puis aller- sur une branche depuis un état détaché

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer une branche : **git branch NOMBRANCHE**
 - ▶ crée une nouvelle branche "à l'endroit où nous sommes"
 - ▶ c.a.d une nouvelle branche qui pointe sur le même commit que HEAD

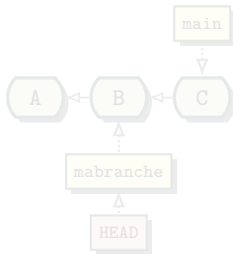
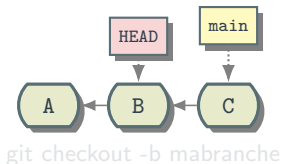


- ▶ Aller sur la branche : **git checkout NOMBRANCHE**



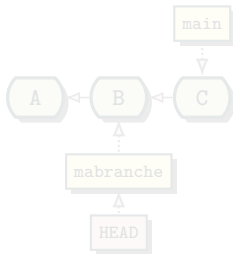
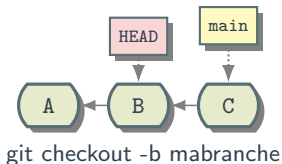
Créer et aller sur une branche depuis un état détaché en une seule étape

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer et aller sur une branche : **git checkout -b NOMBRANCHE**



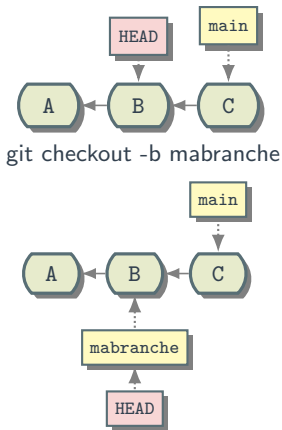
Créer et aller sur une branche depuis un état détaché en une seule étape

- ▶ depuis un état détaché (git checkout HASHB)
- ▶ Créer et aller sur une branche : **git checkout -b NOMBRANCHE**



Créer et aller sur une branche depuis un état détaché en une seule étape

- ▶ depuis un état détaché (`git checkout HASHB`)
- ▶ Créer et aller sur une branche : **`git checkout -b NOMBRANCHE`**



Que signifie " Aller sur une branche" ?

- ▶ se placer sur une branche avec **git checkout NOMBRANCHE**
revient à :
 - 1- stocker le nom de la branche dans le **pointeur courant HEAD**
 - ▶ les prochains commit s'empileront sur cette branche, en faisant avancer le pointeur de branche
 - 2- récupérer dans le dépôt local tous les fichiers/répertoires correspondants à ce commit et les placer dans le **répertoire de travail** et dans **l'index**
 - ▶ autrement dit : tous les fichiers/répertoires suivis/contrôlés par git sont remis dans l'état demandé
 - ▶ les fichiers non suivis sont in-touchés
 - ▶ !! git refuse le checkout s'il existe des fichiers *suivis et modifiés* (non comités) dans le répertoire courant... !! (sauf s'il n'y a pas de conflit : dans ce cas, le fichier est considéré comme déjà modifié)

Que signifie " Aller sur une branche" ?

- ▶ se placer sur une branche avec **git checkout NOMBRANCHE** revient à :
 - 1- stocker le nom de la branche dans le **pointeur courant HEAD**
 - ▶ les prochains commit s'empileront sur cette branche, en faisant avancer le pointeur de branche
 - 2- récupérer dans le dépôt local tous les fichiers/répertoires correspondants à ce commit et les placer dans le **répertoire de travail** et dans **l'index**
 - ▶ autrement dit : tous les fichiers/répertoires suivis/contrôlés par git sont remis dans l'état demandé
 - ▶ les fichiers non suivis sont in-touchés
 - ▶ !! git refuse le checkout s'il existe des fichiers *suivis et modifiés* (non comités) dans le répertoire courant... !! (sauf s'il n'y a pas de conflit : dans ce cas, le fichier est considéré comme déjà modifié)

Que signifie "Aller sur un commit" ?

- ▶ se placer sur un commit avec **git checkout HASH** revient exactement au même que le checkout sur une branche à la différence suivante :
 - ▶ HEAD pointe sur un commit (hash)
 - ▶ !! on se retrouve en mode détaché : (*detached HEAD*)
 - ▶ si l'on veut juste regarder la version, tout est ok
 - ▶ mais si l'on veut faire des modifications et ajouter un commit à cet endroit, il faudra créer une nouvelle branche

Récapitulatif - Opérations de base sur les branches

- ▶ Créer une branche : `git branch NOMBRANCHE`
 - ▶ Aller sur une branche : `git checkout NOMBRANCHE`
 - ▶ Créer et aller sur une branche : `git checkout -b NOMBRANCHE`
 - ▶ Supprimer localement une branche : `git branch -d NOMBRANCHE`
-
- ▶ Lister les branches locales (*=courante) : `git branch`
 - ▶ Lister toutes les branches locales/distantes : `git branch -a`

Supprimer localement une branche

`git branch -d NOMBRANCHE`

- ▶ Concrètement - dans un premier temps - ne fait que supprimer le pointeur (effacer le fichier qui pointe) sur un commit
(les commits de la branches ne peuvent être effacés que si plus rien ne pointe dessus)
- ▶ autorisé uniquement si la branche a déjà été fusionnée ailleurs
(protection)
- ▶ sinon, possibilité de forcer avec
`git branch -d --force NOMBRANCHE`
mais risque de perdre les commits sur lesquels plus rien ne pointe
(garbage collector)
- ▶ l'effacement effectif peut être forcé avec la commande **`git gc`** ou se fait de manière transparente avec l'appel à certaines commandes git.

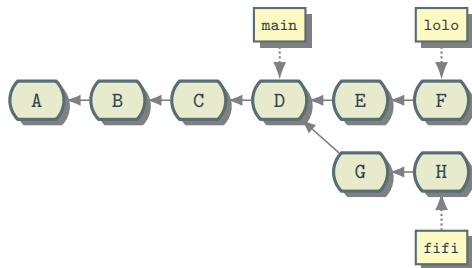
Intermède avant les TPs

- ▶ choisir l'éditeur de texte qui sera lancé.
exemples avec la commande emacs :
 - ▶ dans votre environnement (typiquement `$HOME/.bashrc_profile`)
ajouter :
`export EDITOR=emacs`
 - ▶ ou juste pour git :
`git config --global core.editor emacs`
- ▶ outils graphique pour résoudre les conflits :
`git mergetool`
- ▶ visualisation graphique des branches : `gitk gitg`

TP 1 : créer des branches !

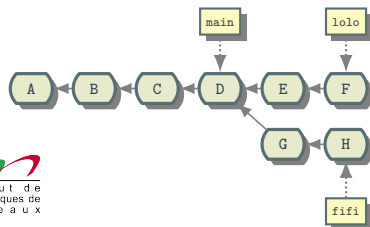
opération préliminaire. exécuter la commande :
git config --global --add push.default current

- ▶ le but de ce TP est de créer, **le plus simplement et le plus minimalement possible**, la structure de dépôt suivante :



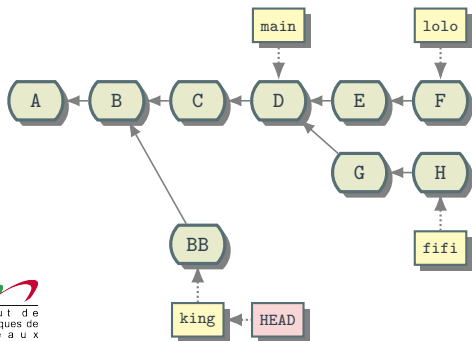
TP 1 : créer des branches ! (solution)

```
mkdir tp1 ; cd tp1 ; git init # ou cloner un dépôt distant
echo A >> a; git add a; git commit -m 'A'
echo B >> a; git add a; git commit -m 'B'
echo C >> a; git add a; git commit -m 'C'
echo D >> a; git add a; git commit -m 'D'
git checkout -b lolo
echo E >> a; git add a; git commit -m 'E'
echo F >> a; git add a; git commit -m 'F'
git checkout main # on revient sur D
git checkout -b fifi
echo G >> a; git add a; git commit -m 'G'
echo H >> a; git add a; git commit -m 'H'
```



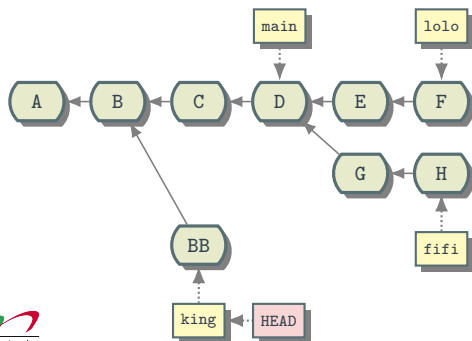
TP 1.1 : detached HEAD

- ▶ le but de ce TP est de tester l'état **detached HEAD** et de savoir le gérer
- ▶ se positionner sur l'état B (c.a.d faire de B le commit courant)
- ▶ créer un nouvelle branche "king" à partir de B
- ▶ créer un nouveau commit BB
- ▶ objectif = arriver à cet état :



TP 1.1 : detached HEAD (solution)

```
git log # reperer le HASH du commit B
git checkout HASHB
git status # HEAD is detached !!
git checkout -b king # creation de la branche king a cet endroit
echo BB >> a; git add a; git commit -m 'BB'
```



Fusion de branches

- ▶ principe : reporter les modifications d'une branche sur une autre
- ▶ le terme fusion (*merge*) peut être trompeur :
 - ▶ on a toujours deux branches après un "merge", dont une (la branche receveuse) qui "évolue" (reçoit un ou plusieurs commits)
- ▶ 2 cas principaux en détails :
 - 1- cas général avec création d'un commit issu de la fusion
 - 2- cas simple du "fast forward" (FF)

Fusionner une branche sur une autre - Cas Général (importer des modifications)

- ▶ permet d'importer sur une branche, dans un seul nouveau commit, les modifications qui ont été apportées sur une autre branche (depuis leur ancêtre commun)
- ▶ en 2 étapes :
 - ▶ `git checkout TARGETBRANCH` : on commence par se placer sur la branche receveuse (destination, *target branch*)
 - ▶ `git merge SOURCEBRANCH` : puis on crée un nouveau commit, sur la branche courante, résultat de la fusion entre la branche courante TARGETBRANCH et la branche source SOURCEBRANCH.
- ▶ ce commit, issu d'une fusion, a 2 commits parents
- ▶ au final :
 - ▶ la branche courante TARGETBRANCH (destination) a avancé d'un commit
 - ▶ la branche SOURCEBRANCHE (source) est **inchangée**
 - ▶ on a toujours les 2 branches
 - ▶ `git log` montre **chronologiquement tous** les commits ancêtres des 2 branches

Fusionner une branche sur une autre - Cas Général (importer des modifications)

- ▶ permet d'importer sur une branche, dans un seul nouveau commit, les modifications qui ont été apportées sur une autre branche (depuis leur ancêtre commun)
- ▶ en 2 étapes :
 - ▶ **git checkout TARGETBRANCH** : on commence par se placer sur la branche receveuse (destination, *target branch*)
 - ▶ **git merge SOURCEBRANCH** : puis on crée un nouveau commit, sur la branche courante, résultat de la fusion entre la branche courante TARGETBRANCH et la branche source SOURCEBRANCH.
- ▶ ce commit, issu d'une fusion, a 2 commits parents
- ▶ au final :
 - ▶ la branche courante TARGETBRANCH (destination) a avancé d'un commit
 - ▶ la branche SOURCEBRANCHE (source) est **inchangée**
 - ▶ on a toujours les 2 branches
 - ▶ **git log** montre **chronologiquement tous** les commits ancêtres des 2 branches

Fusionner une branche sur une autre - Cas Général (importer des modifications)

- ▶ permet d'importer sur une branche, dans un seul nouveau commit, les modifications qui ont été apportées sur une autre branche (depuis leur ancêtre commun)
- ▶ en 2 étapes :
 - ▶ **git checkout TARGETBRANCH** : on commence par se placer sur la branche receveuse (destination, *target branch*)
 - ▶ **git merge SOURCEBRANCH** : puis on crée un nouveau commit, sur la branche courante, résultat de la fusion entre la branche courante TARGETBRANCH et la branche source SOURCEBRANCH.
- ▶ ce commit, issu d'une fusion, a 2 commits parents
- ▶ au final :
 - ▶ la branche courante TARGETBRANCH (destination) a avancé d'un commit
 - ▶ la branche SOURCEBRANCHE (source) est **inchangée**
 - ▶ on a toujours les 2 branches
 - ▶ **git log** montre **chronologiquement tous** les commits ancêtres des 2 branches

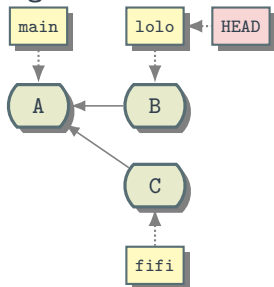
Fusionner une branche sur une autre - Cas Général (importer des modifications)

- ▶ permet d'importer sur une branche, dans un seul nouveau commit, les modifications qui ont été apportées sur une autre branche (depuis leur ancêtre commun)
- ▶ en 2 étapes :
 - ▶ **git checkout TARGETBRANCH** : on commence par se placer sur la branche receveuse (destination, *target branch*)
 - ▶ **git merge SOURCEBRANCH** : puis on crée un nouveau commit, sur la branche courante, résultat de la fusion entre la branche courante TARGETBRANCH et la branche source SOURCEBRANCH.
- ▶ ce commit, issu d'une fusion, a 2 commits parents
- ▶ au final :
 - ▶ la branche courante TARGETBRANCH (destination) a avancé d'un commit
 - ▶ la branche SOURCEBRANCHE (source) est **inchangée**
 - ▶ on a toujours les 2 branches
 - ▶ git log montre **chronologiquement tous** les commits ancêtres des 2 branches

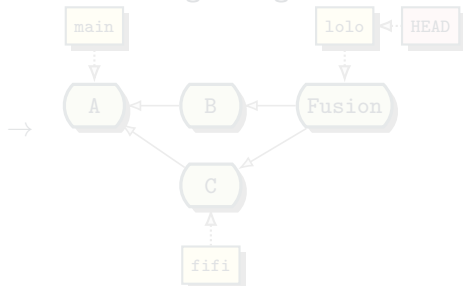
Fusion de branches : cas général

- ▶ ex: la branche courante/receveuse "lolo" prend en compte les modifications de la branche "fifi" (source)

git checkout lolo :



git merge fifi

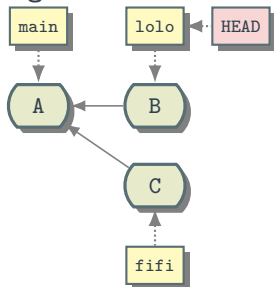


- ▶ un nouveau "commit de fusion" est créé sur la branche courante/receveuse (lolo)
- ▶ la branche source (fifi) est inchangée

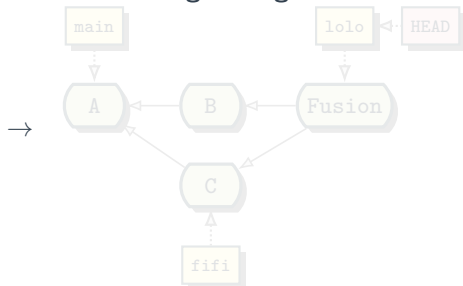
Fusion de branches : cas général

- ▶ ex: la branche courante/receveuse "lolo" prend en compte les modifications de la branche "fifi" (source)

git checkout lolo :



git merge fifi

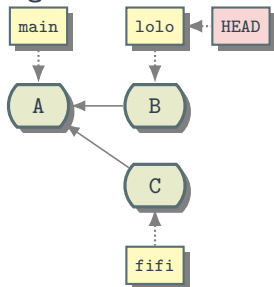


- ▶ un nouveau "commit de fusion" est créé sur la branche courante/receveuse (lolo)
- ▶ la branche source (fifi) est inchangée

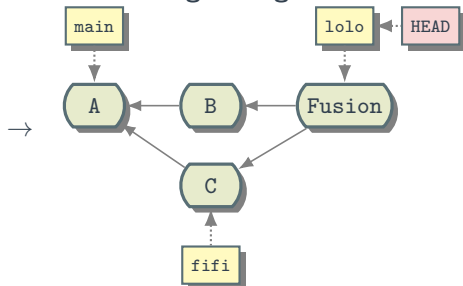
Fusion de branches : cas général

- ▶ ex: la branche courante/receveuse "lolo" prend en compte les modifications de la branche "fifi" (source)

git checkout lolo :



git merge fifi

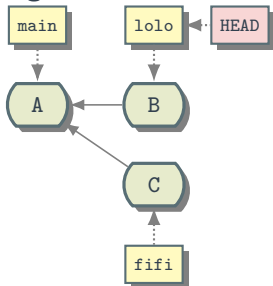


- ▶ un nouveau "commit de fusion" est créé sur la branche courante/receveuse (lolo)
- ▶ la branche source (fifi) est inchangée

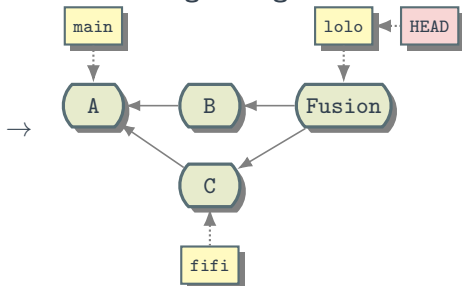
Fusion de branches : cas général

- ▶ ex: la branche courante/receveuse "lolo" prend en compte les modifications de la branche "fifi" (source)

git checkout lolo :



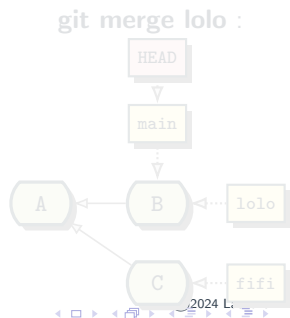
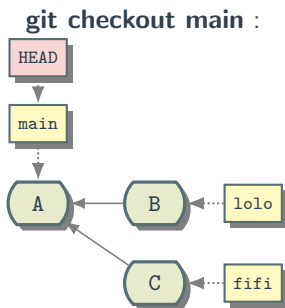
git merge fifi



- ▶ un nouveau "commit de fusion" est créé sur la branche courante/receveuse (lolo)
- ▶ la branche source (fifi) est inchangée

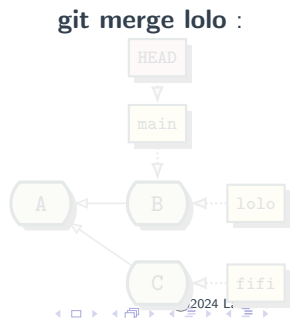
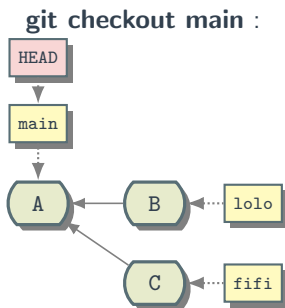
Fusion de branches : cas particulier du "Fast Forward" (FF)

- ▶ dans le cas où
 - ▶ la **branche source est directement dans le prolongement de la branche receveuse** : il suffit de faire avancer le pointeur de la branche receveuse.
 - ▶ ou autrement dit : **la branche source est en avance sur la branche receveuse**
- ▶ c'est le cas du **fast forward** : aucun commit n'est créé !
- ▶ ex. fusion de "lolo" dans "main". seuls les pointeurs avancent :



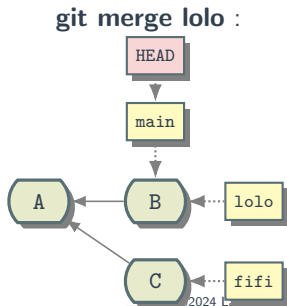
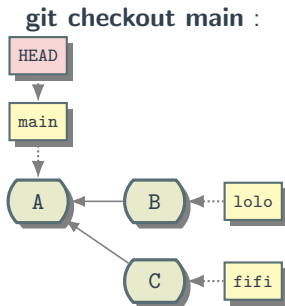
Fusion de branches : cas particulier du "Fast Forward" (FF)

- ▶ dans le cas où
 - ▶ **la branche source est directement dans le prolongement de la branche receveuse** : il suffit de faire avancer le pointeur de la branche receveuse.
 - ▶ ou autrement dit : **la branche source est en avance sur la branche receveuse**
- ▶ c'est le cas du **fast forward** : aucun commit n'est créé !
- ▶ ex. fusion de "lolo" dans "main". seuls les pointeurs avancent :



Fusion de branches : cas particulier du "Fast Forward" (FF)

- ▶ dans le cas où
 - ▶ la **branche source est directement dans le prolongement de la branche receveuse** : il suffit de faire avancer le pointeur de la branche receveuse.
 - ▶ ou autrement dit : **la branche source est en avance sur la branche receveuse**
- ▶ c'est le cas du **fast forward** : aucun commit n'est créé !
- ▶ ex. fusion de "lolo" dans "main". seuls les pointeurs avancent :

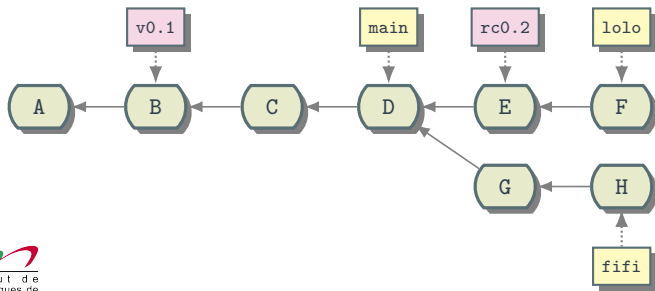


branches, dépôt, synchronos

- ▶ nouvelle vision d'un dépôt git
- ▶ branches et dépôts (distant)
- ▶ branches et synchronisation de dépôts distants
 - ▶ fetch
 - ▶ pull
 - ▶ push

Nouvelle vision d'un dépôt git

- ▶ un dépôt git n'est qu'un gros tas de commits : A, B, C, D, ... , (une base de données indexée par des hashes) chaînés entre eux, avec "des étiquettes" (références) qui pointent vers certains commits remarquables :
 - ▶ les branches (en jaune) : qui pointent vers un commit et qui avancent au grès des "git commits" (**main**, **fifi**, **lolo**)
 - ▶ les tags (en rose) : sorte d'alias, qui pointent vers un hash et *qui ne bougent que si on les déplace explicitement* (**v1**, **rc0.2**)



Le dépôt distant et ses branches distantes

- ▶ jusqu'à présent nous avons joué avec les branches locales c.a.d contenues dans notre dépôt local
- ▶ les branches locales ont vocation à être synchronisées avec le dépôt distant, qui est un autre tas de commits avec des étiquettes (*références*)
- ▶ les branches locales sont habituellement associées avec les branches distantes de même nom (branches **upstream** ou **remote-tracking branch**) mais préfixé du nom du dépôt distant (origin) :
ex: (local) main <-correspond-à-> origin/main (distant)
lolo <-correspond-à-> origin/lolo
fifi <-correspond-à-> origin/fifi
- ▶ au niveau implémentation on va retrouver une copie des branches distantes (les hash) dans le sous répertoire refs/remotes :

```
.git/refs/remotes/origin/main  
.git/refs/remotes/origin/fifi  
.git/refs/remotes/origin/lolo
```

Le dépôt distant et ses branches distantes

- ▶ jusqu'à présent nous avons joué avec les branches locales c.a.d contenues dans notre dépôt local
- ▶ les branches locales ont vocation à être synchronisées avec le dépôt distant, qui est un autre tas de commits avec des étiquettes (*références*)
- ▶ les branches locales sont habituellement associées avec les branches distantes de même nom (branches **upstream** ou **remote-tracking branch**) mais préfixé du nom du dépôt distant (origin) :
ex: (local) main <-correspond-à-> origin/main (distant)
lolo <-correspond-à-> origin/lolo
fifi <-correspond-à-> origin/fifi
- ▶ au niveau implémentation on va retrouver une copie des branches distantes (les hash) dans le sous répertoire refs/remotes :

```
.git/refs/remotes/origin/main  
.git/refs/remotes/origin/fifi  
.git/refs/remotes/origin/lolo
```

Le dépôt distant et ses branches distantes

- ▶ jusqu'à présent nous avons joué avec les branches locales c.a.d contenues dans notre dépôt local
- ▶ les branches locales ont vocation à être synchronisées avec le dépôt distant, qui est un autre tas de commits avec des étiquettes (*références*)
- ▶ les branches locales sont habituellement associées avec les branches distantes de même nom (branches **upstream** ou **remote-tracking branch**) mais préfixé du nom du dépôt distant (origin) :
ex: (local) main <-correspond-à-> origin/main (distant)
lolo <-correspond-à-> origin/lolo
fifi <-correspond-à-> origin/fifi
- ▶ au niveau implémentation on va retrouver une copie des branches distantes (les hash) dans le sous répertoire refs/remotes :

```
.git/refs/remotes/origin/main  
.git/refs/remotes/origin/fifi  
.git/refs/remotes/origin/lolo
```


Remarque / Nouvelle branche locale, branche distantes et git push

- ▶ la première fois que vous "poussez" une branche
- ▶ il est nécessaire de définir le nom de la branche remote
- ▶ sauf si vous avez la configuration globale :
 - ▶ `git config --global --add push.default current`
- ▶ sinon, besoin de préciser le nom de la branche distante (upstream / remote-tracking branch) à associer, avec
 - ▶ `git push --set-upstream NOM_DEPOT_REMOTE NOM_BRANCHE`
 - ▶ ex: `git push --set-upstream origin mabranche`
- ▶ une branche distante (ex: `origin/nom-branche`) peut être récupérée localement en faisant :
 - ▶ `git checkout nom-branche`

Synchronisation de dépôts et fusion de branches distantes

- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier les commits manquantes de part et d'autres - il n'y a jamais de conflit !
 - ▶ **les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergées de part et d'autre.
- ▶ **git fetch** synchronise uniquement le dépôt local en récupérant toutes les commits disponibles sur le dépôt distant et en recopiant (dans `.git/refs/remotes/origin/...`) les valeurs de toutes les branches distantes. Pas de fusion ... donc jamais de conflit avec cette commande !
git fetch affiche un résumé des modifications (branches concernées)
- ▶ **git merge** (sans argument) fusionne, dans la branche locale courante, sa branche distante correspondante (ex: main avec `origin/main`)

git pull = git fetch + git merge

Synchronisation de dépôts et fusion de branches distantes

- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier les commits manquantes de part et d'autres - il n'y a jamais de conflit !
 - ▶ **les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergées de part et d'autre.
 - ▶ **git fetch** synchronise uniquement le dépôt local en récupérant toutes les commits disponibles sur le dépôt distant et en recopiant (dans `.git/refs/remotes/origin/...`) les valeurs de toutes les branches distantes. Pas de fusion ... donc jamais de conflit avec cette commande !
git fetch affiche un résumé des modifications (branches concernées)
 - ▶ **git merge** (sans argument) fusionne, dans la branche locale courante, sa branche distante correspondante (ex: main avec `origin/main`)
- `git pull = git fetch + git merge`

Synchronisation de dépôts et fusion de branches distantes

- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier les commits manquantes de part et d'autres - il n'y a jamais de conflit !
 - ▶ **les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergées de part et d'autre.
- ▶ **git fetch** synchronise uniquement le dépôt local en récupérant toutes les commits disponibles sur le dépôt distant et en recopiant (dans `.git/refs/remotes/origin/...`) les valeurs de toutes les branches distantes. Pas de fusion ... donc jamais de conflit avec cette commande !
git fetch affiche un résumé des modifications (branches concernées)
- ▶ **git merge** (sans argument) fusionne, dans la branche locale courante, sa branche distante correspondante (ex: main avec `origin/main`)

`git pull = git fetch + git merge`

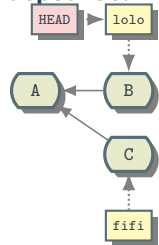
Synchronisation de dépôts et fusion de branches distantes

- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier les commits manquantes de part et d'autres - il n'y a jamais de conflit !
 - ▶ **les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergées de part et d'autre.
- ▶ **git fetch** synchronise uniquement le dépôt local en récupérant toutes les commits disponibles sur le dépôt distant et en recopiant (dans `.git/refs/remotes/origin/...`) les valeurs de toutes les branches distantes. Pas de fusion ... donc jamais de conflit avec cette commande !
git fetch affiche un résumé des modifications (branches concernées)
- ▶ **git merge** (sans argument) fusionne, dans la branche locale courante, sa branche distante correspondante (ex: main avec origin/main)

git pull = git fetch + git merge

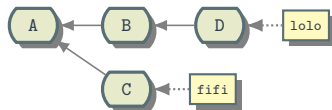
Synchronisation vers le dépôt local : **git fetch**

dépôt local



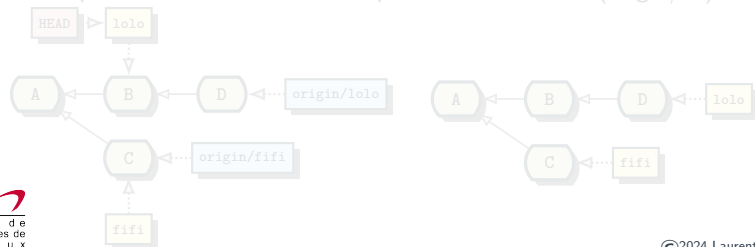
dépôt distant "origin"

commit D en plus !



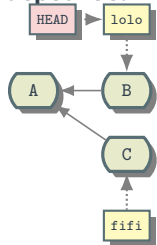
← git fetch ←

copie des commits et des étiquettes de branches (origin/...)



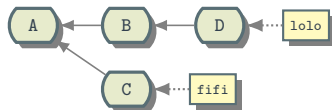
Synchronisation vers le dépôt local : **git fetch**

dépôt local



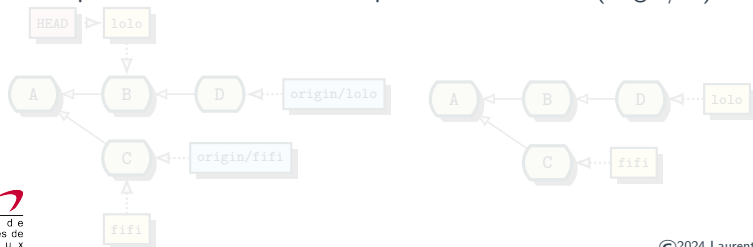
dépôt distant "origin"

commit D en plus !



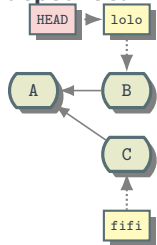
← **git fetch** ←

copie des commits et des étiquettes de branches (origin/...)



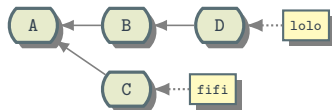
Synchronisation vers le dépôt local : **git fetch**

dépôt local



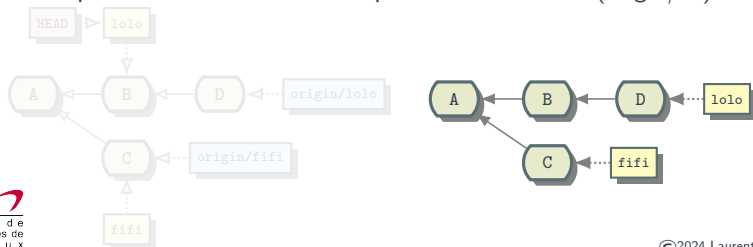
dépôt distant "origin"

commit D en plus !



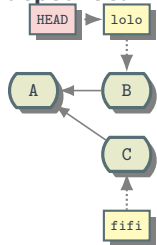
← **git fetch** ←

copie des commits et des étiquettes de branches (origin/...)



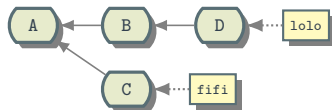
Synchronisation vers le dépôt local : **git fetch**

dépôt local



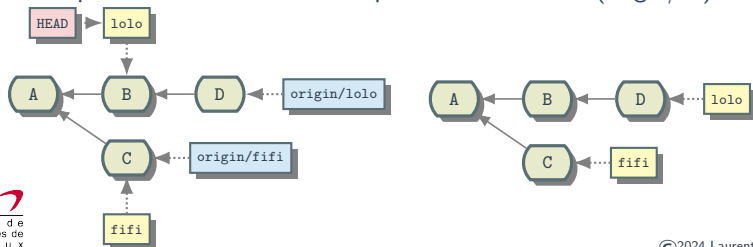
dépôt distant "origin"

commit D en plus !



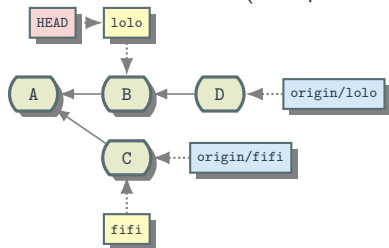
← **git fetch** ←

copie des commits et des étiquettes de branches (origin/...)



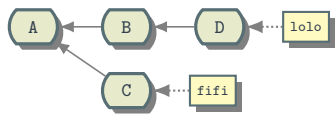
Fusion au sein du dépôt local : **git merge** (fast forward)

dépôt local



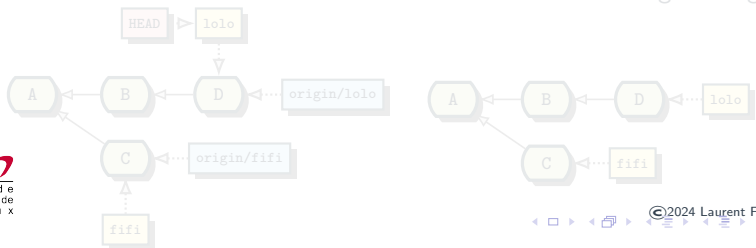
dépôt distant "origin"

(état précédent "après git fetch")



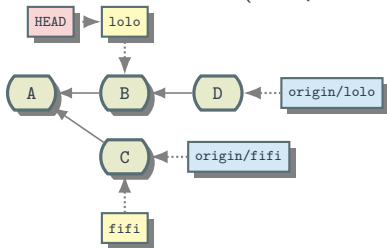
← git merge ←

branche courante : "lolo" fusionnée en FF avec son homologue "origin/lolo"



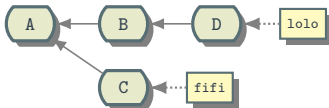
Fusion au sein du dépôt local : **git merge** (fast forward)

dépôt local



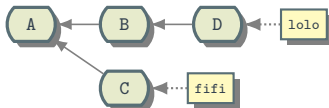
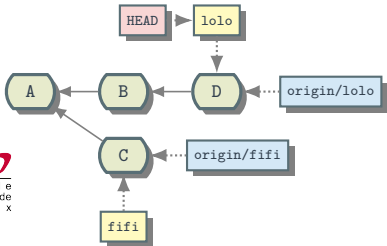
dépôt distant "origin"

(état précédent "après git fetch")



← **git merge** ←

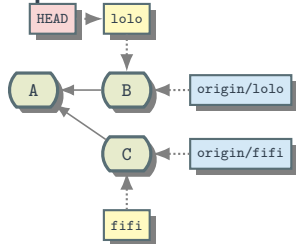
branche courante : "lolo" fusionnée en FF avec son homologue "origin/lolo"



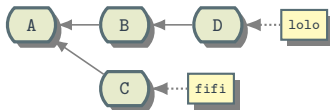
Synchronisation/Fusion vers le dépôt local :

`git pull = git fetch + git merge`

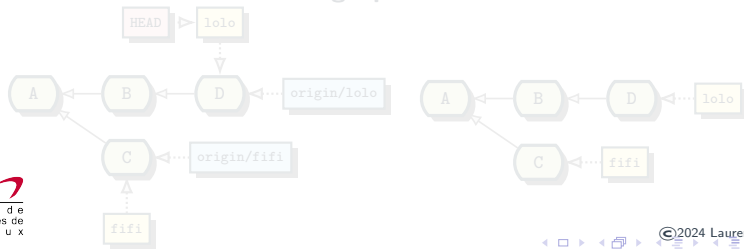
dépôt local



dépôt distant "origin"
commit D en plus !



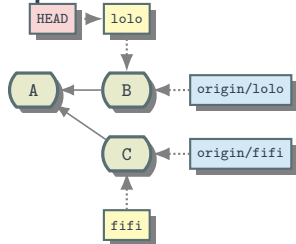
← git pull ←



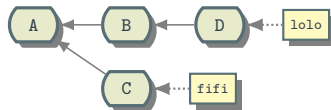
Synchronisation/Fusion vers le dépôt local :

`git pull = git fetch + git merge`

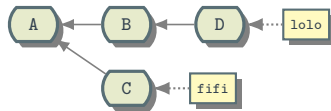
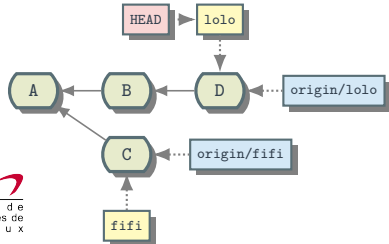
dépôt local



dépôt distant "origin"
commit D en plus !



← `git pull` ←



Synchronisation de dépôts et fusion de branches distantes

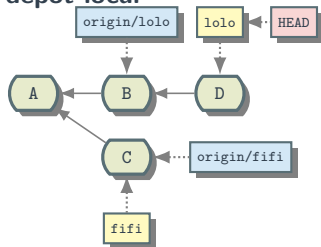
- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier (dans le sens demandé) les commits manquantes coté destination - il n'y a jamais de conflit.
 - ▶ **les étiquettes (références) et en particulier les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergé de part et d'autre.
- ▶ **git push = "le contraire de git pull" = "fetch + merge qui serait fait vers le dépôt distant"** = synchronise le dépôt distant à partir du dépôt local et provoque la fusion de la branche local courante sur son homologue distant mais **en fast forward seulement !**

Synchronisation de dépôts et fusion de branches distantes

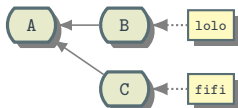
- ▶ synchroniser deux dépôts revient à synchroniser 2 choses :
 - ▶ **le gros tas de commits** : il suffit de recopier (dans le sens demandé) les commits manquantes coté destination - il n'y a jamais de conflit.
 - ▶ **les étiquettes (références) et en particulier les pointeurs de branches** : c'est là qu'il peut y avoir des fusions (et des conflits) à gérer quand les branches ont divergé de part et d'autre.
- ▶ **git push = "le contraire de git pull" = "fetch + merge qui serait fait vers le dépôt distant"** = synchronise le dépôt distant à partir du dépôt local et provoque la fusion de la branche local courante sur son homologue distant mais **en fast forward seulement !**

Synchronisation/Fusion vers le dépôt distant : **git push**

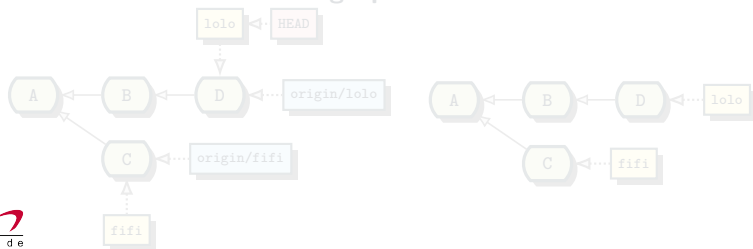
dépôt local



dépôt distant "origin"
commit D en moins !

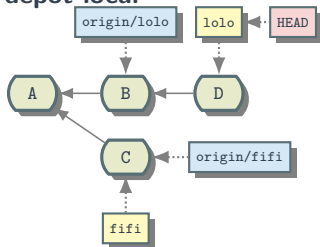


→ git push →



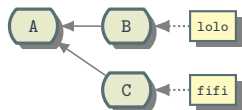
Synchronisation/Fusion vers le dépôt distant : **git push**

dépôt local

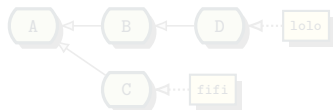
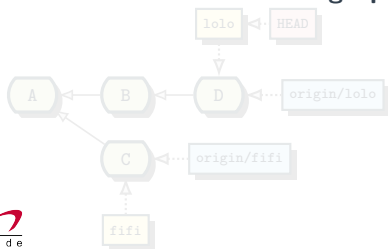


dépôt distant "origin"

commit D en moins !

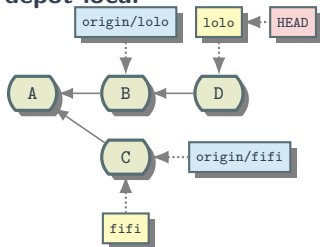


→ git push →



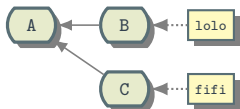
Synchronisation/Fusion vers le dépôt distant : **git push**

dépôt local

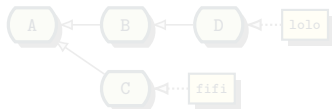
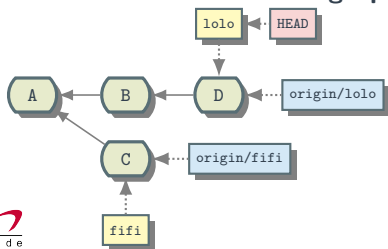


dépôt distant "origin"

commit D en moins !

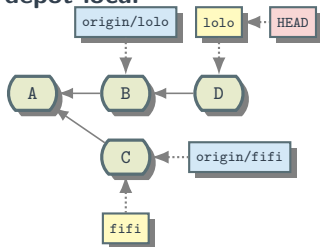


→ git push →

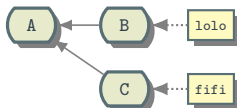


Synchronisation/Fusion vers le dépôt distant : git push

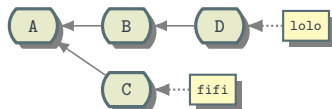
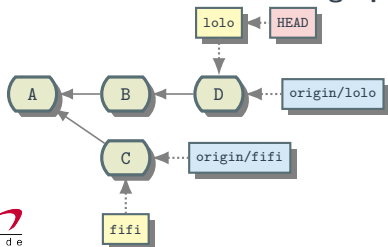
dépôt local



dépôt distant "origin"
commit D en moins !



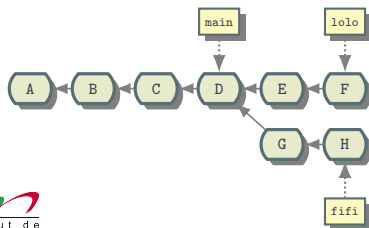
→ git push →



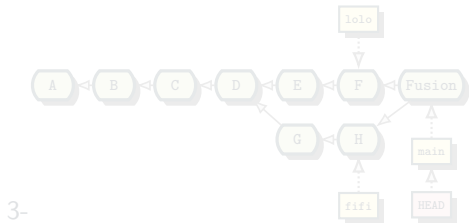
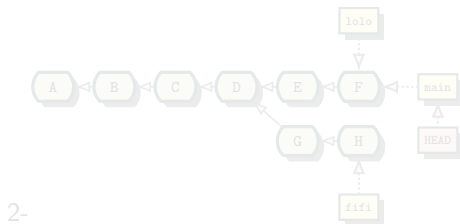
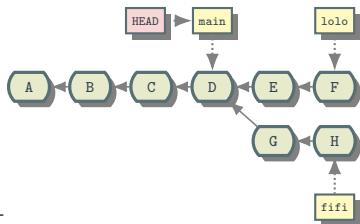
TP 2 : quelques fusions !

- ▶ 1- les modifications de **lolo** étant concluantes, vous allez les merger dans **main**
- ▶ 2- puis, les modifications de **fifi** étant également mûres, vous allez les merger dans **main**
- ▶ 3- dessinez le graphe résultat avec en particulier la position des 3 branches

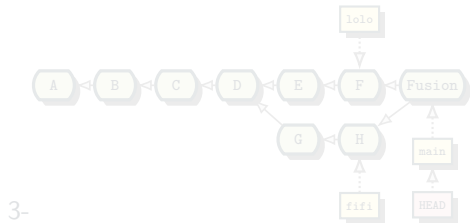
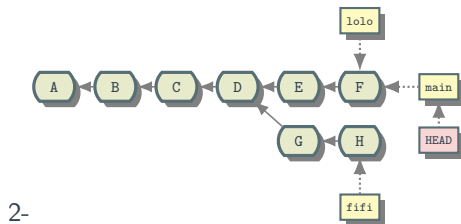
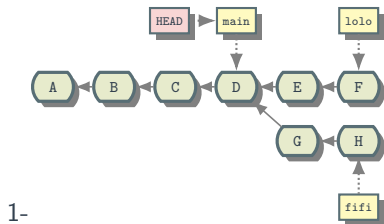
Challenge supplémentaire : essayez de prévoir à l'avance quels types des merge vont avoir lieu et si des problèmes risquent apparaître...



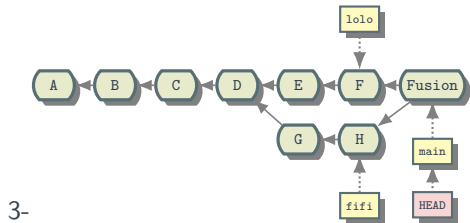
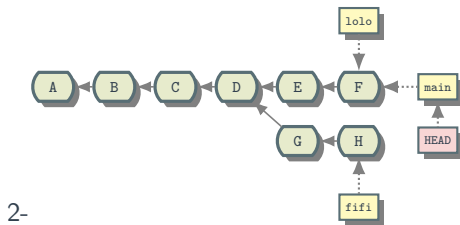
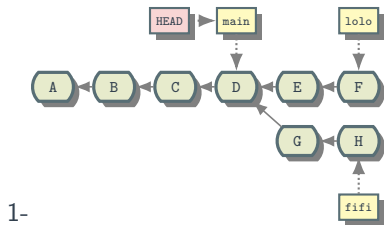
TP 2 : quelques fusions ! (solutions)



TP 2 : quelques fusions ! (solutions)



TP 2 : quelques fusions ! (solutions)



TP 2 : quelques fusions ! (solutions)

```
git checkout main          # retour sur main
git merge lolo             # en fast forward !
git merge fifi            # aie... un conflit :(
# éditer le fichier 'a' à votre convenance
git add a
git commit -m 'Fusion fifi dans main'
```


TP 3 : git poème - 2 à 4 joueurs (expérimental!)

- ▶ constituer un groupe de $N=2$ à 4 participants
- ▶ on tire au sort celui qui va créer le projet commun
- ▶ il crée le projet et donne les droits **maintainer** à tous les participants (ou 'developer' en levant la protection sur la branche 'main' dans settings/repo/protected)
- ▶ un autre participant crée le fichier poesie.txt avec $N*3$ lignes identiques : "SUJET VERBE COMPLÉMENT" (à 2 joueurs, on crée $2*3=6$ lignes identiques)
- ▶ à chaque tour :
 - ▶ on s'assure de bien (re)partir de la dernière version
 - ▶ chacun modifie ce fichier en remplaçant 3 mots clefs, sur 3 lignes différentes (un par ligne) par de vrais mots qu'il choisit, en tenant compte du type de mot à remplacer (ex: VERBE devient mange)
 - ▶ on propage le fichier sur le serveur - chacun son tour de pousser en premier
 - ▶ on attends que tout le monde ait terminé pour passer au tour suivant

TP 3 : git poème (solution)

```
### joueurs A et B
# joueur A crée le dépôt
# puis joueur B initialise le poeme :
git clone ... ; cd poeme
echo 'SUJET VERBE COMPLEMENT' >> poeme.txt ; # N fois !
echo 'SUJET VERBE COMPLEMENT' >> poeme.txt ; # N fois !
echo 'SUJET VERBE COMPLEMENT' >> poeme.txt ; # N fois !
git add poeme.txt
git commit -m 'poeme initial' ; git push
# joueur A :
git fetch ; # à faire jusqu'à voir des modifications de B !
# joueurs A et B :
# BEGIN tant que "partie pas finie" :
git pull # pour être sûr d'avoir la dernière version
## action: remplacer 3 mots clefs au hasard dans le poeme.txt
git add poeme.txt ; git commit -m 'ma poesie' ;
## si je suis premier à pousser :
    git push
## si non
git fetch # à répéter jusqu'à voir les modifications de l'autre joueur
git pull # régler le conflit éventuel (edit/add/commit)
git push
## attendre la fin du tour
# END
```

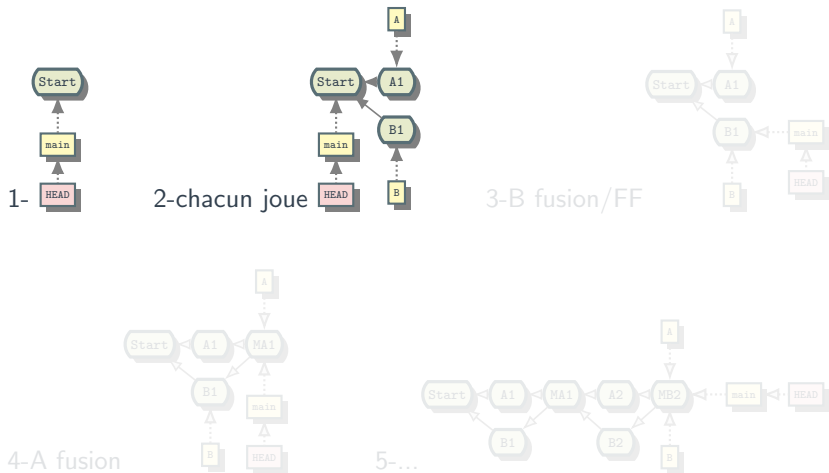
TP 4 : git branch poeme - 2 à 4 joueurs (expérimental!)

- ▶ même jeu que tout à l'heure avec le même dépôt mais **dans un nouveau fichier poeme2.txt** et avec la nuance suivante :
- ▶ **chacun se crée sa propre branche**
- ▶ **chacun ne joue (n'écrit) que dans sa branche** et doit ensuite propager ses modifications dans la branche main
- ▶ puis récupérer, dans sa propre branche, les modifications effectués par les autres joueurs dans main
- ▶ note: le fichier 'poeme2.txt' est initialement crée dans main
- ▶ note: on pousse également sa propre branche sur le serveur

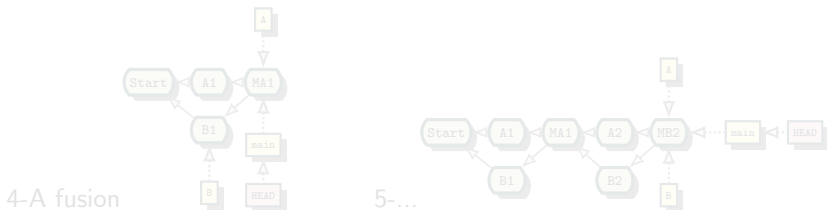
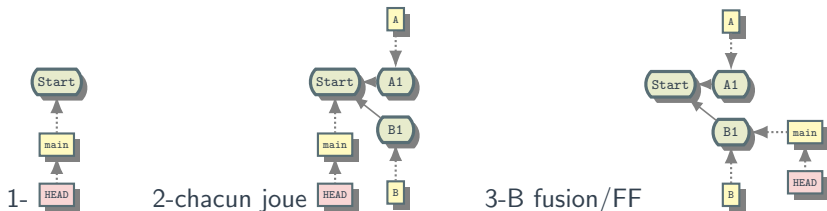
TP 4 : git branch poème (solution) TODO



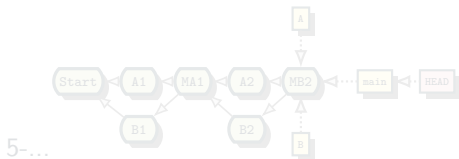
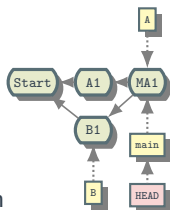
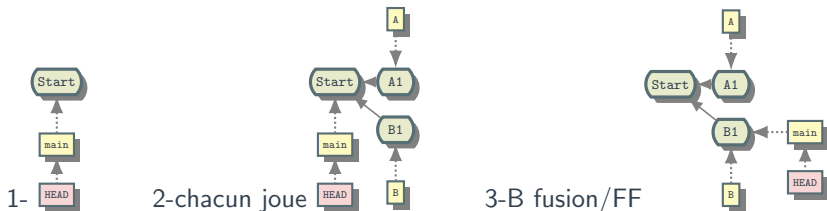
TP 4 : git branch poème (solution) TODO



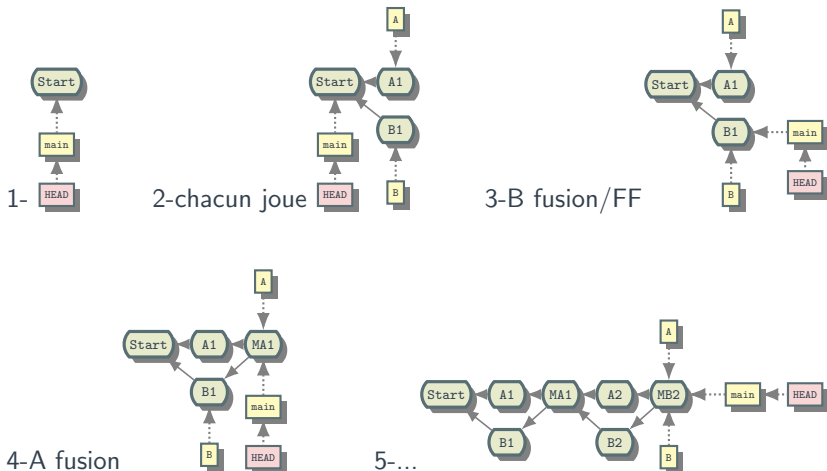
TP 4 : git branch poème (solution) TODO



TP 4 : git branch poème (solution) TODO



TP 4 : git branch poème (solution) TODO



TP 4 : git branch poème (solution)

```
# joueur A crée le dépôt distant
# initialisation - joueurs A et B clonent le dépôt :
git clone ... ; cd poeme
# puis joueur B initialise le poeme2.txt :
echo 'SUJET VERBE COMPLEMENT' >> poeme2.txt # 3*N fois !
git add poeme2.txt
git commit -m 'branch poeme initial' ; git push
git branch B # création de la branche B à partir de main
# joueur A - mise à jour et initialisation :
git fetch # à faire jusqu'à voir des modifications de B dans main !
git pull # récupérer la dernière version de main
git branch A # création de la branche A à partir de main
### joueurs A et B - dans la suite, on remplace MABRANCHE par A ou B selon le joueur
# BEGIN tant que "partie pas finie" :
git checkout main ; git pull # mise à jour de main en Fast Forward
git checkout MABRANCHE ; git merge main # on revient sur notre branche et on intègre main en Fast Forward
## action: dans ma branche, remplacer 3 mots clefs au hasard dans le poeme2.txt
git add poeme2.txt ; git commit -m 'ma poesie' ; git push
git checkout main # on revient sur main (local)
git merge MABRANCHE # on joue dans main (local)
## si je suis premier à pousser :
git push
## si non
git fetch # à répéter jusqu'à voir les modifications du joueur précédent
git pull # régler le conflit éventuel (edit/add/commit)
git push # mise à jour de main avec notre modification
## attendre la fin du tour
# END
```

TP 5 : git tac toe - 2 joueurs (trop compliqué!)

- ▶ constituer un groupe de 2 participants : A et B
- ▶ A, le perdant du chifoumi crée un dépôt et donne les droits **maintainer** à B (ou 'developer' en levant la protection sur la branche 'main' dans settings/repo/protected)
- ▶ chacun clone le dépôt
- ▶ on communique juste pour se synchroniser avant de jouer le coup suivant
- ▶ B crée un fichier 'game' contenant exactement 5 lignes de 5 caractères '+'
- ▶ A et B doivent jouer chaque coup en remplaçant un '+' par leur lettre (A ou B) et pousser le résultat
- ▶ le but du jeux est d'aligner 4 lettres, à chaque tour, chacun joue une fois
- ▶ si A et B jouent au même endroit... le coup est annulé et il est interdit de rejouer la même case au coup suivant
- ▶ on joue dès que l'on est prêt (après avoir *pullé* les modifications de l'autre), mais c'est à chacun son tour de pusher le premier (A commence).

TP 5 : git tac toe (solution)

```
# joueurs A et B
git clone ... ; cd git-tac-toe
# joueur A :
echo '+++++' >> game ; echo '+++++' >> game ;
echo '+++++' >> game ; echo '+++++' >> game ;
echo '+++++' >> game ;
git add game
git commit -m 'game created' ; git push
# joueur B :
git fetch ; # à faire jusqu'à voir des modifications de A !
git pull
# joueurs A et B :
# BEGIN tant que "partie pas finie" :
## action: jouer un coup dans le fichier game (remplacer un + par votre lettre)
git add game ; git commit -m 'mon coup' ;
## si je suis premier à jouer :
    git push
    git fetch # à répéter jusqu'à voir les modifications de l'autre joueur
    git pull
## si je suis deuxième à jouer :
git fetch # à répéter jusqu'à voir les modifications de l'autre joueur
git pull # régler le conflit éventuel (edit/add/commit)
git push
# END
```

TP 6 : git tac branch - 2 joueurs (bien trop compliqué!)

- ▶ même jeu que tout à l'heure avec la nuance suivante :
- ▶ chacun se crée une branche (branche A pour A et B pour B)
- ▶ chacun ne joue que dans sa branche...
- ▶ note: le fichier 'game' est initialement crée dans main par A

TP 6 : git tac branch (solution)

```
# joueurs A et B
git clone ... ; cd git-tac-toe
# joueur A :
echo '+++++' >> game ; echo '+++++' >> game ;
echo '+++++' >> game ; echo '+++++' >> game ;
echo '+++++' >> game ;
git add game
git commit -m 'game created' ; git push
# joueur B :
git fetch ; # à répéter jusqu'à voir des modifications de A !
git pull
git checkout -b B
# joueur A
git checkout -b A
# joueurs A et B :
# tant que "partie pas finie" :
## jouer un coup dans le fichier game (remplacer un + par votre lettre)
git add game ; git commit -m 'mon coup' ;
## si je suis premier à jouer/pusher :
    git push
    git fetch # à répéter jusqu'à voir les modifications de l'autre joueur
    git checkout 0 ; git pull # remplacer 0 par la lettre de l'autre joueur, M la votre
    git checkout M ; git merge 0
    git push
## si je suis deuxième à jouer/pusher :
git fetch # à répéter jusqu'à voir les modifications de l'autre joueur
git checkout 0 ; git pull # remplacer 0 par la lettre de l'autre joueur, M la votre
git checkout M ; git merge 0
git push
```

