**M1 informatique – module GL et environnement pro. 1**

**TP1** : Rappels Java, git et maven

# Exercice de modélisation d'un panier de fruits

## **Partie 1 :** Utilisations **simples** de git via GitHub et les lignes de commande

### 1) Création d'un dépôt distant avec GitHub *(via un navigateur web)*

Se connecter ("sign in") ou s'inscrire ("sign up" / "register") à un compte **GitHub** (https://github.com/). Y créer un **nouveau dépôt** (repository) **distant vide** nommé **TPGL** (dans lequel on va versionner des classes Java). Cocher la case intitulée *"Initialiser le dépôt avec un README"* et associer le fichier **.gitignore** avec le langage **Java** (pour ne pas stocker les fichiers Java de compilation, de log, d'archivage).

### 2) Obtenir en local une copie du dépôt distant *(via un terminal – ligne de commande)*

Pour obtenir **une copie locale** du dépôt distant **TPGL** qu'on vient de créer sur GitHub, on va lancer (à partir du **répertoire** local choisi pour stocker le contenu des TPs de Génie Logiciel) :

> git **clone** https://github.com/votre-login/TPGL.git *//pour obtenir la copie du dépôt distant*
*//ceci créé un **répertoire TPGL** dans le répertoire courant (contenant README.md et .gitignore)*

**Se placer dans le répertoire TPGL via la ligne de commande adéquate, puis :**

*//Par défaut, le mot-clé "**origin**" est associé à l'url du dépôt distant (ci-dessus) pour pouvoir y*
*//faire référence de manière plus concise, c'est ce qu'on observe en tapant :*

> git **remote** -v *//précise pour chaque mot-clé (dont origin) son url associée*

> git **status** *//informations sur l'**état** des dépôts et indications des choses à faire :*
*//**A UTILISER LE PLUS SOUVENT POSSIBLE ! Rien à faire actuellement !***

### 3) Test de la classe Orange.java *(à inclure dans votre répertoire TPGL)*

a. Récupérer la classe **Orange.java** sur **Plubel**. Celle-ci contient deux attributs : ***prix*** et ***origine*** représentant respectivement le prix unitaire (en euros : de type double) et le pays d'origine de l'orange (représenté par une chaîne de caractères). Dans le ***constructeur*** à 2 paramètres, on a fait en sorte qu'il ne soit pas possible de créer des oranges avec un **prix négatif**.

b. Écrire une méthode « **main** » (dans la classe **Orange**), avec quelques tests simples des différentes méthodes fournies dans la classe. **Vérifier en ligne de commande** que la **compilation** (javac Orange.java) et **l'exécution** (java Orange) se passent bien.

### 4) Synchronisation des dépôts local et distant *(via un terminal – ligne de commande)*

a. Pour "**versionner**" la classe Orange dans le dépôt **local**, lancer les lignes de commandes suivantes, toujours à partir du **répertoire TPGL** :

> git **status** *//indique que le fichier Orange.java est non suivi*

> git **add** Orange.java *//pour **indexer** le fichier Orange.java (**git add -A** s'il y en a plusieurs)*

> git **status** *//indique que le fichier Orange.java est indexé*

> git **commit** -m "premier commit" *//pour **valider** les modifications (créer un instantané)*
*//un message vous indique que le commit a bien été fait*

> git **status** *//rien à valider (en local) mais indique qu'on est en avance sur origin/master*

*//si vous essayez maintenant de modifier le fichier Orange.java et faites ensuite :*

> git **diff** *//différences entre le dernier commit, l'index, et le répertoire de travail courant*

*// il sera indiqué la ligne de la modification effectuée : **A UTILISER REGULIEREMENT***

Pour plus d'informations sur une commande git : **man git "commande"** *(en ligne de commande)*

b. On veut maintenant **synchroniser** les dépôts local et distant, pour cela, il faut lancer (toujours à partir du **répertoire TPGL**) :

*//Depuis peu, GitHub n'accepte plus les mots de passe des comptes lors de l'authentification des opérations Git. Vous devez maintenant ajouter un PAT (Personal Access Token) en suivant les instructions suivantes : https://exerror.com/remote-support-for-password-authentication-was-removed-on-august-13-2021-please-use-a-personal-access-token-instead/*

> git **remote** set-url origin https://***PAT***@github.com/***votre-login***/TPGL

> git **push** origin main *//pour "pousser" les changements effectués en local sur le dépôt distant*

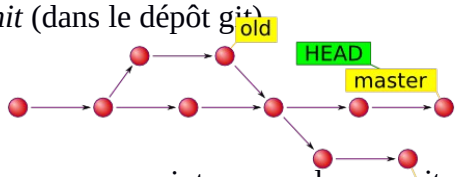
> git **status** *//rien à valider (en local) et indique que votre branche est à jour avec origin/main*

## 5) Création de branches avec git *(via un terminal – ligne de commande)*

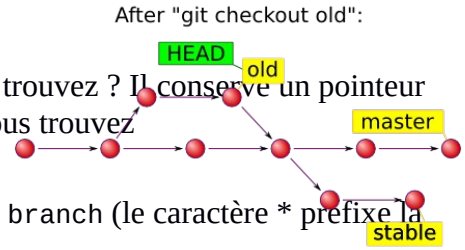
Voici un schéma expliquant le **principe des branches dans git** (*où **master** est devenu **main***) :

- **branche** = pointeur mobile léger vers un des objets *commit* (dans le dépôt git)

- branche par défaut : **main** (code prêt à être déployé)



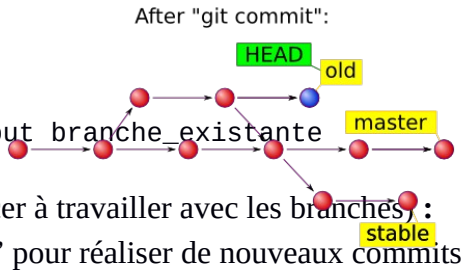

- créer une **nouvelle branche** (`git branch nom`) = créer un nouveau pointeur vers le *commit* actuel




- comment git connaît-il la branche sur laquelle vous vous trouvez ? Il conserve un pointeur spécial : **`HEAD`** = pointeur sur la branche locale où vous vous trouvez

- pour connaître la liste des branches existantes : `git branch` (le caractère * prefixe la branche en cours)

`git branch -a` (voir les branches locales et distantes)




- pour **basculer vers une branche existante :** `git checkout branche_existante`

**Réalisez maintenant les étapes suivantes** (pour commencer à travailler avec les branches) **:**

a. Créer une **branche locale** nommée "nouveau_fruit" pour réaliser de nouveaux commits :

> git **branch** nouveau_fruit //création de la branche

> git branch //pour savoir (grâce au symbole *) sur quelle branche on se situe (en local)

> git **checkout** nouveau_fruit //pour basculer vers la nouvelle branche

//les 2 commandes en gras ci-dessus s'écrivent aussi : git checkout **-b** nouveau_fruit

b. Écrire une nouvelle classe représentant un **nouveau fruit** (toujours dans TPGL), ayant les mêmes attributs et méthodes que la classe **Orange** : chaque binôme se verra attribuer un fruit différent parmi : Fraise, Cerise, Pomme, Poire, Banane, Kiwi, Framboise, Ananas, ...

c. Lancer les **commandes git** suivantes pour faire un suivi des modifications sur la branche courante, puis fusionner sur la branche *main* (en local) et mettre à jour le dépôt distant :

> git status //indique que le fichier « Fruit ».java est non suivi

> git add « Fruit ».java *//pour **indexer** le nouveau fichier créé*

> git status *//indique que ce fichier est indexé*

> git commit -m "ajout Fruit" *//pour **valider** les modifications (créer un instantané)*

> git status *//indique : sur la branche nouveau_fruit, rien à valider*

> git **checkout** main *//se déplacer sur la branche main (principale) avant la fusion*

> git **merge** nouveau_fruit *//pour fusionner la branche nouveau_fruit avec master*
*//le message indique **fast-forward** lors de la fusion (avance rapide) : git a simplement //déplacé le pointeur (vers l'avant)*

> git **branch -d** nouveau_fruit *//supprimer la branche locale nouveau_fruit*

> git **pull** *//pour mettre à jour le répertoire de travail local avec les données du dépôt //distant : **RIEN A METTRE A JOUR ACTUELLEMENT***

> git **push** origin main *//aller "pousser" les changements effectués sur le dépôt distant*

> git **log** *//pour voir l'historique des commits réalisés et sur quoi pointe **HEAD***

## Partie 2 : Travail à plusieurs sur un même dépôt GitHub et conflits

### 1) Obtenir en local une copie d'un nouveau dépôt distant (partagé par tout le TP)

Pour obtenir **une copie locale** de ce nouveau dépôt distant, lancer (à partir du **répertoire** local choisi pour stocker ce dépôt partagé par tout le groupe de TP) :

> git **clone** https://github.com/roudetUb/GLPanierTP1.git ***//POUR LE GROUPE TP1***

→ *modifier en **GLPanierTP2.git** ou **GLPanierTP3.git …** pour les autres groupes*

**Puis se placer dans le répertoire créé via la ligne de commande adéquate et :**

> git **status** *//informations sur l'**état** du dépôt et indications des choses à faire*

### 2) Compléter la classe Panier.java en créant une nouvelle branche git

Dans le nouveau dépôt, vous trouverez des nouvelles classes dont **Panier.java** et une nouvelle interface **Fruit.java**. Chaque binôme devra compléter certaines méthodes de la classe **Panier** en veillant aux **conflits** possibles. **Demander les instructions à l'enseignant ...**

**Réalisez les étapes suivantes :**

a. Créer une **branche locale** nommée "creation_panier" pour réaliser de nouveaux commits :

> git **branch** creation_panier //création de la branche

> git branch //pour savoir (grâce au symbole *) sur quelle branche on se situe (en local)

> git **checkout** creation_panier //pour basculer vers la nouvelle branche
//les 2 commandes en gras ci-dessus s'écrivent aussi : git checkout **-b** creation_panier

b. Écrire la/les méthode(s) vous concernant dans la classe **Panier.java**, puis ajouter la classe correspondant au **nouveau fruit** créé précédemment (en la faisant dériver de **Fruit.java**).

c. Dans la méthode **"main"** de la classe **Panier**, réaliser quelques tests simples pour s'assurer que la/les méthode(s) écrite(s) précédemment respecte(nt) bien les consignes indiquées.

d. Lancer les **commandes git** suivantes pour faire un suivi des modifications sur la branche courante, puis fusionner sur la branche *main* (en local) et mettre à jour le dépôt distant :

> git status //indique que le fichier Panier.java est non suivi

> git add Panier.java *//pour **indexer** le fichier Panier.java*

> git status *//indique que le fichier Panier.java est indexé*

> git commit -m "ajout Panier" *//pour **valider** les modifications (créer un instantané)*

> git status *//indique : sur la branche creation_panier, rien à valider*

> git **checkout** main *//se déplacer sur la branche main (principale) avant la fusion*

> git **merge** creation_panier *//pour fusionner la branche creation_panier avec master*

*//le message indique **fast-forward** lors de la fusion (avance rapide) : git a simplement //déplacé le pointeur (vers l'avant)*

> git **branch -d** creation_panier *//supprimer la branche locale creation_panier*

> git **pull** *//met à jour le répertoire de travail local avec les données du dépôt distant*

*//**pull** automatise la mise à jour des données mais peut entraîner de nombreux conflits si //beaucoup de fichiers ont été modifiés (car regroupe les commandes git **fetch** et **merge**)*

*//**fetch** permet de garder son répertoire de travail à jour et de contrôler le moment où l'on //souhaite fusionner les données.*

*> git **remote** set-url origin https://<PAT>@github.com/roudetUb/GLPanierTP1(2 ou 3)*

> git **push** origin main *//aller "pousser" les changements effectués sur le dépôt distant*

> git **log** *//pour voir l'historique des commits réalisés et sur quoi pointe **HEAD***

### 3) Fusions et conflits via GitHub

Le but est ici d'évoquer les **fusions** (merge) de *commits*. Une fusion peut être **conflictuelle** lorsqu'au moins 2 utilisateurs travaillent en parallèle sur les mêmes lignes du même fichier.

Voici ce qui peut se produire, lorsque vous tenterez de lancer (si par exemple vous « *poussez* » tous votre classe **Orange** avec vos tests personnels sur le dépôt distant) :

> **git push** //on vous demandera certainement de faire d'abord un `git pull` qui pourra déclencher un **conflit**, affichant un message du type :

*Fusion automatique de Panier.java*
*CONFLIT (contenu) : Conflit de fusion dans Panier.java*
*La fusion automatique a échoué ; réglez les conflits et validez le résultat.*

Git n'a pas automatiquement créé le ***commit* de fusion**. Il a arrêté le processus le temps que vous résolviez le conflit. Si vous voulez vérifier, à tout moment après l'apparition du conflit, quels fichiers n'ont pas été fusionnés, vous pouvez lancer la commande `git status`.

Pour **résoudre le conflit** manuellement, ouvrez le fichier Panier.java avec *gedit* (ou utilisez **`git mergetool -t meld`** pour ouvrir une interface). Ce fichier doit se présenter de la façon suivante :

**<<<<<<<<<< commit de utilisateur1**
**# version de utilisateur1**
**========**
**# version de utilisateur2**
**>>>>>>>>>> commit de utilisateur2**

Les **flèches gauches** (<< suivies du nom de la branche sur laquelle on a fait la fusion) indiquent qu'il s'agit du côté **local** (jusqu'aux symboles ===) et les **autres (>>)** le côté **distant**, suivi de l'identifiant du commit distant concerné (donc le dernier en date sur le dépôt distant et sur la branche concernée).

Modifiez le fichier pour **ne garder qu'une seule** des 2 versions (ou un mélange des 2 !), tout en retirant tous les symboles <<<, === et >>>. A ce stade, le **conflit** est considéré comme **résolu**.

Enfin ***commitez*** (`git add Panier.java`, puis `git commit -m "conflit de la methode ... resolu"`), puis **poussez** vers le dépôt distant, jusqu'à ce que **`git status`** vous indique que tout est à jour.

→ Aide sur la commande **merge** et les **branches** : https://git-scm.com/book/fr/v2

→ Aide conflits : https://www.atlassian.com/fr/git/tutorials/using-branches/merge-conflicts

## **Partie 3 :** Utilisation de git via GitHub et **Netbeans** + **maven**

Nous allons maintenant voir comment se servir des commandes git à partir de Netbeans, en se servant des classes Java créées en **Partie 1**. Nous allons, pour cela, nous **placer dans un nouveau répertoire** pour créer un **projet Netbeans** à partir des classes créées auparavant.

**1)** Dans **Netbeans**, choisir : **Team > Git > Clone...** puis indiquer l'adresse de votre **dépôt distant** https://github.com/votre-login/TPGL.git, votre login et mot de passe, puis spécifier le répertoire à créer pour votre dépôt local (par défaut "/home1/votre_loginIEM/NetBeansProjects/**TPGL**").

Après avoir appuyé sur le bouton "Next", on nous demande de sélectionner les branches distantes qui nous intéressent, puis "Next". On peut alors (dans la fenêtre suivante), choisir sur quelle branche se placer (Checkout branch) : choisir **main**, puis "Finish".

Une fois ces opérations terminées, une boîte de dialogue vous demande si vous voulez **créer un projet IDE** : répondre **oui** et **suivre pas à pas les explications ci-dessous :**
Créer un **nouveau projet Java avec Maven** (choisir **Java with Maven** > « Project from Archetype », puis sélectionner l'archétype « **maven-archetype-quickstart** » (dernière version) en gardant les options par défaut, puis indiquer dans la dernière fenêtre le **nom du projet** (**Panier**) et comme **GroupID** : « **fr.ufrsciencestech** », puis Terminer).

Netbeans a créé les **2 packages** suivants : « **Packages de sources** » et « **Packages de tests** », ainsi que les fichiers **App.java** (contenant le main) et **AppTest.java** (test unitaire JUnit).

Vous ne voyez pas apparaître vos 2 fichiers *Orange.java et « Fruit ».java* (dans l'arborescence de votre projet), mais ils sont pourtant bien présents. Dans l'explorateur de fichiers, placez-vous dans le répertoire que Netbeans a créé : vous voyez ces 2 fichiers **à côté du répertoire Panier**.

Comme ce sont des classes sources, il va falloir les déplacer dans le dossier :
**Panier/src/main/java/fr/ufrsciencestech/panier/**, à côté de App.java (qu'on peut maintenant supprimer).

Revenez maintenant dans Netbeans où la mise à jour a dû s'effectuer automatiquement et :

- **clic droit sur Orange.java** et sélectionner **Refactor > Move..** et cliquer sur le bouton **Refactor** (pour bien intégrer dans Netbeans le changement de répertoire de la classe), faire de même pour l'autre classe .java

- lancer un « **run** » (classe **Orange**) : plusieurs téléchargements vont s'effectuer (on peut aussi lancer la commande "`mvn package`" dans un terminal, lorsqu'on est situé à la racine du projet)

- vérifiez sur quelle branche git vous vous trouvez : **clic droit** sur le projet **> Git > Repository > Repository Browser** : une fenêtre indique l'ensemble des branches avec **en gras** celle sur laquelle on se trouve actuellement.

- faire un **clic droit** sur le projet **> Git > Show Changes** : va indiquer dans une fenêtre les différences entre le répertoire de travail (modifié par Netbeans) et le dépôt local (résultant du clone du dépôt distant)

- faire un **clic droit** sur le projet **> Git > Add** pour indexer les changements

- faire un **clic droit** sur le projet **> Git > Commit…** : la fenêtre qui s'ouvre indique les fichiers qui seront ajoutés au dépôt local et propose un message associé à ce *commit* (à remplacer par un message explicite).

- **clic droit** sur le projet **> Git > Show Changes** : indique qu'il n'y a plus de différences

- faire un **clic droit** sur le projet **> Git > Remote > Push ...** : pour aller pousser les modifications vers le dépôt distant (GitHub). Dans la partie *« Specify Git Repository Location: »*, indiquer votre *<PAT>* dans la zone « **User** » et laissez le mot de passe vide.

Vérifiez dans GitHub les modifications. On peut constater que même si vous avez déplacé précédemment les fichiers .java pour qu'ils ne figurent plus à la racine du dépôt, ils sont toujours présents dans GitHub, car vous avez un autre dépôt local qui y fait référence.

**2)** Rapatriez la classe **Panier** (précédemment créée en commun), en créant une nouvelle branche **ajouts_panier** via **Netbeans**. Lancer un « **run** » (classe **Panier**), puis faire un ***commit.***

**3)** Lorsque tous les tests passent, il est temps de **fusionner** sur la branche **main** pour passer le code en production. En **ligne de commande**, faire :

> git branch -a //et vérifier que la branche locale master n'existe pas

> git checkout main //on obtient le message : 'main est paramétrée pour suivre la branche //distante 'main' depuis 'origin'. Basculement sur la nouvelle branche 'main'

Puis dans **Netbeans**, faire :

- **clic droit** sur le projet **> Git > Branch/Tag > Merge Revision…** : puis dans la zone de texte "Revision", sélectionner la branche **ajouts_panier** puis cliquer sur le bouton **Merge.**

- **clic droit** sur le projet **> Git > Show Changes** : si aucune différence à signaler, c'est que tout s'est bien passé !

**4)** Il faudra enfin envoyer le projet **Panier** sur le dépôt distant (GitHub), pour pouvoir le partager et l'utiliser avec **Jenkins** (outil d'intégration continue). Pour cela, à partir d'un clic droit sur le projet **Panier**, faites :

- **Git > Remote > Pull from Upstream** (savoir s'il y a une nouvelle modification à charger sur GitHub, en cas d'utilisation collaborative),

- puis si "**No update**" est indiqué, faites **Git > Remote > Push to Upstream** pour aller mettre à jour le dépôt distant (GitHub) avec les nouveaux fichiers de votre projet **Panier**. **Vérifiez** sur GitHub.

**5)** Si vous souhaitez que votre projet Netbeans ne soit plus lié au dépôt ou changer de dépôt : **Menu Team**, puis **Disconnect...**

- Pour plus de simplicité, ajouter le **plugin Git Toolbar** dans Netbeans (menu **Tools > Plugins**)

## **Partie 4 :** Utilitaires Java divers

### 1) Archive JAR et maven

Dans un **projet Netbeans classique** (généré à partir d'un script de build **Ant**), à chaque nouvelle compilation du projet, un fichier **jar exécutable** est créé automatiquement dans le répertoire **dist/** du projet. Pour **créer le .jar** avec **maven** (dans **target/**), il faut soit :

- dans **Netbeans** : click droit sur le projet, **Run Maven > Goals...** et écrire : `clean package`
- ouvrir un **terminal** et exécuter "`mvn clean package`" (lorsqu'on est à la racine du projet)

Sans oublier d'ajouter ce qui suit dans votre **pom.xml** (après les balises <**build**> <**plugins**>) :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.1.2</version>
    <configuration>
      <archive>
       <manifest>
        <addClasspath>true</addClasspath>
        <mainClass>fr.ufrsciencestech.panier.Panier</mainClass>  <!-- classe contenant le main -->
       </manifest>
      </archive>
    </configuration>
</plugin>
```

Pour exécuter ce **.jar** en ligne de commande : **java -jar Panier-1.0-SNAPSHOT.jar**
Pour examinez le contenu du **.jar** (en l'extrayant) : **jar xfv Panier-1.0-SNAPSHOT.jar**.

### 2) Javadoc

Ajouter quelques commentaires dans vos classes (en format **Javadoc**) et générer les fichiers HTML correspondants (clic droit sur le projet puis “Generate Javadoc”). Les fichiers HTML créés sont placés dans le répertoire **target/site/apidocs/**.
Pour générer les fichiers HTML à chaque appel de la commande “mvn site”, aller voir ici :
https://maven.apache.org/plugins/maven-javadoc-plugin/examples/selective-javadocs-report.html

### 3) Refactoring

Comprendre l’intérêt du **menu “Refactor”** de **Netbeans** (pour renommer, déplacer, copier, supprimer sans danger, …) à l’aide du tutoriel suivant : http://wiki.netbeans.org/Refactoring

## **Annexe 1 :** Utilisation basique de Git et autres commandes de base

Tiré de : http://yannesposito.com/Scratch/fr/blog/2009-11-12-Git-for-n00b/
et de : https://openclassrooms.com/fr/courses/2342361-gerez-votre-code-avec-git-et-github

### 1) La façon immédiate de travailler avec Git

- récupérer les modifications des autres : **`git pull`**
- voir les détails de ces modifications : **`git log`**
- Plusieurs fois:
  - *Faire une modification atomique*
  - verifier le details de ses modifications : **`git status`** et **`git diff`**
  - indiquer si nécessaire que de nouveaux fichiers doivent être *versionnés :* **`git add [file]`**
  - enregistrer ses modifications : **`git commit -a -m "message"`**
  - envoyer ses modifications aux autres : **`git push`** (refaire un **`git pull`** si le push renvoie une erreur).

Voilà, avec ces quelques commandes vous pouvez utiliser Git sur un projet avec d’autres personnes. Même si c’est suffisant, il faut quand même connaître une chose avant de se lancer ; la gestion des *conflits*.

### 2) Autres commandes

#### 2.1) Revenir dans le temps

- pour toute l’arborescence :

```
$ git checkout
$ git revert
```

- revenir trois versions en arrière :

```
$ git uncommit 3
```

- revenir avant le dernier merge (s’il s’est mal passé) :

```
$ git revertbeforemerge
```

- pour un seul fichier :

```
$ git checkout file
$ git checkout VersionHash file
```

```
$ git checkout HEAD~3 file
```

### 2.2) Lister les différences entre chaque version

- Liste les fichiers en cours de modifications :

```
$ git status
```

- différences entre les fichiers de la dernière version et les fichiers locaux :

```
$ git diff
```

- liste les différences entre les fichier d'une certaine version et les fichiers locaux :

```
$ git diff VersionHash fichier
```

- nommer certaines versions pour s'y référer facilement :

```
$ git tag 'toto'
```

- afficher l'historique des modifications :

```
$ git log
$ git lg
$ git logfull
```

- savoir qui a fait quoi et quand :

```
$ git blame fichier
```

- gérer des conflits :

```
$ git conflict
```

### 2.3) Manipuler des branches

Un élément que vous allez être souvent amenés à utiliser lorsque vous travaillez sur un repo, ce sont les branches. Les branches permettent de travailler sur des versions de code qui divergent de la branche principale contenant votre code courant. Lorsque vous initialisez un dépôt Git, votre code est placé dans la branche principale appelée **master** par défaut.

- Pour **voir** les branches présentes dans votre dépôt Git :
`$ git branch` (ajoutera une **étoile** devant la branche dans laquelle vous êtes placés)

- pour **créer** une nouvelle branche :
`$ git branch nouvelle_branche`

- pour **changer** de branche courante :
`$ git checkout nouvelle_branche`

- pour créer une branche et s'y positionner :
`$ git checkout -b nouvelle_branche`

- pour **fusionner** des branches (ajouter dans une branche A les mises à jour faites dans une autre branche B) :
- se placer dans la branche A : `git checkout brancheA`
- fusionner ce qui a été fait dans B vers A : `git merge brancheB`
Les modifications faites sur les différentes branches doivent toujours être fusionnées dans la branche principale **master**.

### 2.4) Retrouver qui a fait une modification

Pour retrouver qui a modifié une ligne précise de code dans un projet, faire une recherche avec `git log` peut s'avérer compliqué, surtout si le projet contient beaucoup de commits. Il existe un autre moyen plus direct de retrouver qui a fait une modification particulière dans un fichier : `git blame nomdufichier.extension`

Cette commande liste toutes les modifications qui ont été faites sur le fichier, ligne par ligne. À chaque modification est associé la **signature** du commit correspondant. Par exemple :

```
^05b1233 (Marc Gauthier 2014-08-08 00:31:02 1) # Une liste
```

Pour retrouver pourquoi cette modification a été faite, vous avez deux possibilités :

1. Faire un `git log` et rechercher le commit dont la **signature** commence par `05b1233`,
2. utiliser la commande `git show` qui vous renvoie directement les détails du commit recherché en saisissant le début de sa **signature** : `git show 05b1233`

### 2.5) Ignorer des fichiers

Pour des raisons de sécurité et de clarté, il est important d'ignorer certains fichiers dans Git, tels que :

- tous les fichiers de configuration (config.xml, databases.yml, .env...)
- les fichiers et dossiers temporaires (tmp, temp/...)
- les fichiers inutiles comme ceux créés par votre IDE ou votre OS (.DS_Store, .project...)

**Le plus crucial** est de ne **JAMAIS** versionner une variable de configuration, que ce soit un mot de passe, une clé secrète ou quoi que ce soit de ce type. Versionner une telle variable conduirait à une large faille de sécurité, surtout si vous mettez votre code en ligne sur GitHub ! Si vous avez ce type de variables de configuration dans votre code, déplacez-les dans un fichier de configuration et **ignorez ce fichier dans Git** : en utilisant le fichier **.gitignore**.
Si le code a déjà été envoyé sur GitHub, partez du principe que quelqu'un a pu voir vos données de configuration et mettez-les à jour (changez votre mot de passe ou bien générez une nouvelle clé secrète).

**Créez le fichier .gitignore** pour y lister les fichiers que vous ne voulez pas versionner dans Git (les fichiers comprenant les variables de configuration, les clés d'APIs et autres clés secrètes, les mots de passe, etc.). Listez ces fichiers ligne par ligne dans **.gitignore** en indiquant leurs chemins complets, par exemple :

```
motsdepasse.txt
config/application.yml
```

Le fichier **.gitignore** doit être **tracké** comme vos autres fichiers dans Git : vous devez donc l'ajouter à l'index et le « committer ».

### 2.6) Éviter des commits superflus

Imaginez le scénario suivant : vous êtes en train de travailler sur une fonction, lorsque tout à coup une urgence survient et un collègue vous demande de résoudre un bug dans un autre fichier et/ou une autre branche.

Si vous faites un `commit` des modifications sur votre fonction à ce stade, cela va alourdir votre historique car vous n'avez pas terminé votre tâche. **Comment faire pour ne pas perdre vos modifications en cours avant de passer à l'urgence à traiter ?**
Vous pouvez **mettre de côté vos modifications en cours** avec la commande : **`git stash`**

Vous pouvez alors vous rendre dans la branche/le fichier que vous devez traiter à l'instant, finir et committer vos modifs. Une fois que vous avez réglé cette urgence, revenez sur la branche sur

laquelle vous étiez en train de travailler, et récupérez les modifications que vous aviez mises de côté avec la commande : `git stash pop`

Attention, `pop` vide votre `stash` des modifications que vous aviez rangées dedans. Donc une fois que vous avez récupéré ces modifications dans votre branche, il vous faut finir votre tâche et les committer ! (ou bien les remettre de côté en exécutant à nouveau la commande `git stash`).
Si vous voulez garder les modifications dans votre `stash`, vous pouvez utiliser `apply` à la place de `pop` : `git stash apply`

### 3) Aides et liens utiles

Mémos Git : https://www.lije-creative.com/memo-git-en-ligne-commande/
https://rogerdudler.github.io/git-guide/index.fr.html

Git et Netbeans : https://github.com/hugoscurti/INF2015-H14/wiki/Labo-03-:-Git-avec-NetBeans ou https://richardcarlier.com/article-408-git-et-netbeans.html (vidéo)

Git et Eclipse :
https://github.com/iblasquez/tuto_git/blob/master/egit/git_egit_tutoriel.md

Essayer les commandes Git et voir ce qu'elles font : https://learngitbranching.js.org/

Bon **tutoriel** pour utiliser Git via les lignes de commande :
https://github.com/hugoscurti/INF2015-H14/wiki/Labo-04-:-Git-(Partie-2)

Article intéressant : https://delicious-insights.com/fr/articles/bien-utiliser-git-merge-et-rebase/