A Java course

Table of Contents

1. Presentation

Présentation[1]

L'enseignant

- Olivier Cailloux
- olivier.cailloux@dauphine.fr
- Coordonnées : cf. annuaire de Dauphine
- Développeur sur projets de recherche
- Enseignant chercheur au LAMSADE

1 / 8

Objectifs pédagogiques

- Programmer de « vraies » applications
- De qualité
- Fournir et utiliser des composants réutilisables
- Conception objet
- Prise en main d'outils de dév avancés :
  - Eclipse ;
  - Maven ;
  - git (livraisons exclusivement via GitHub)

2 / 8

Intérêt pratique

- Technologies omniprésentes et très demandées (15 In-Demand Tech-Focused (And Tech-Adjacent) Skills And Specialties, 2022, Forbes Technology Council)
- Qu'on soit programmeur, qu'on discute avec des programmeurs
- Décomposition en responsabilités, en sous-problèmes
- Respect des spécifications
- Utile au-delà de la programmation

3 / 8

[1] https://github.com/oliviercailloux/java-course/raw/main/L3/Pr%C3%A9sentation%20du%20cours%20Objet/presentation.pdf

L'enseignant Obj. pédagogiques Mise en œuvre Attendu

Prérequis

- Notions algorithmiques élémentaires (boucles, structures de listes, d'arbres...)
- Familiarité avec un langage tel que C++ ou Python
- Manipulation de votre système d'exploitation : installation de logiciels, navigation dans le système de fichiers, démarrage de programmes
- Capacité à comprendre des textes en anglais liés à l'informatique
- Un ordinateur fonctionnel

4 / 8

L'enseignant Obj. pédagogiques Mise en œuvre Attendu

Évaluation

100% CC

- Tests réguliers en séance (annoncés)
- (Possible selon avancement) Devoirs maison / Remises de projet
- (Possible selon avancement) QCMs
- Aggrégation des notes reçues au long de l'année
- Pondération augmente au fil de l'année

5 / 8

L'enseignant Obj. pédagogiques Mise en œuvre Attendu

Évaluation des tests

- Code doit compiler
- Livraison via git
- Respect précis des instructions
- Évaluation généralement automatique
- Points pour chaque aspect fonctionnel

6 / 8

L'enseignant Obj. pédagogiques Mise en œuvre Attendu

Contenu

- Syntaxe Java
- Programmation objets : responsabilités ; techniques
- Maven
- Éléments d'ingénierie : programmation par contrat ; patrons de conception...
- Collections
- Tests unitaires
- Utilisation de bibliothèques tierces
- Exceptions
- Logging
- Et plus selon demandes

7 / 8

L'enseignant Obj. pédagogiques Mise en œuvre Attendu

Travail attendu

- $\{[(25\text{ h} / \text{ECTS}) \times 5\text{ ECTS}] - 51\text{ h}\} / 16$ inter-séances
- 5 heures de travail entre chaque séance en moyenne
- Prenez des notes
- Poursuivre les exercices chez vous, cf. page GitHub du cours

8 / 8

2. Shell

2.1. Introduction

A shell (also called a terminal or a command line interface) permits to invoke programs by typing commands.

- Under Linux, use BASH (Bourne-again Shell), for example.
- Under Windows, choose one of these routes.
  - Install git[2]; use Git BASH (recommended for beginners);
  - use (Windows) PowerShell[3], which is probably installed already (recommended for power users).

[2] https://git-scm.com/download
[3] https://docs.microsoft.com/powershell/scripting/install/installing-windows-powershell

- (Different shells admit slightly different syntax and provide slightly different capabilities, and commands sometimes differ, but the commands we will need for this course are the same in most classical shells, except when indicated.)

2.2. Basics

Read Introduction to command line[4] for the very basics of using a shell. Use ↑ (keyboard up arrow[5]) to reuse previous commands. (This tutorial[6] could help as well.)

At the end of this part you are supposed to be able to use a shell to, at least, change directory and list files.

2.3. About arguments

A shell command contains (typically) a program name, followed by arguments, separated by spaces. Example: `touch afile anotherfile` calls the program `touch` with two arguments. This command creates two empty files, `afile` and `anotherfile`, if they do not exist already.

Any argument can be surrounded by quotes, thus, the commands `touch "afile" "anotherfile"` or `touch afile anotherfile` or `touch "afile" anotherfile` are equivalent to the previous one. *However*, to form a single argument containing spaces, quotes are *mandatory*. Example: `touch "a file" "another file"` contains two arguments: `a file` and `another file`. Thus, this command creates two files, named `a file` and `another file`, with spaces in their names. The command `touch a file another file` (without quotes) would instead create four files.

Here are other examples for better understanding what an argument is: the command `ls -a` has one argument, `-a`. It is equivalent to `ls "-a"`. The command `ls --color=always "a file"` has two arguments. It is equivalent to `ls "--color=always" "a file"`. The command[7] `git config --type bool --get core.filemode` has five arguments (considering `git` as the program name and `config` as its first argument).

2.4. Exercice

Open a shell, navigate (using `cd`) to some different place of your choice on your hard disk, create a file `my file.txt` there (using the shell), delete it with your graphical file explorer, check in the terminal that it has disappeared (using the shell).

2.5. Relative and absolute paths

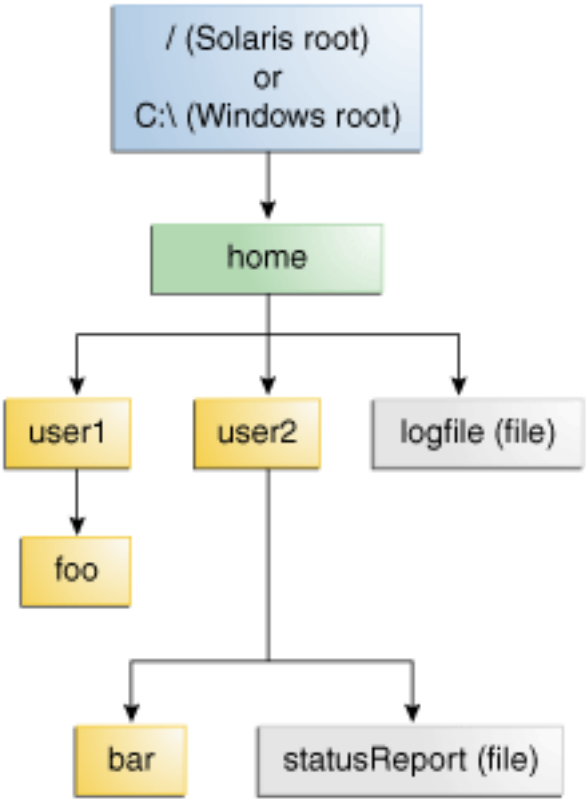
It is often necessary to refer to files or directories as arguments to commands in the terminal. You do this usually by using absolute or relative file or directory paths.

A file or a directory stored on your hard disk has an absolute “path” (or, less technically, “name”) that refers to it unambiguously. For example, under Linux (or MacOS): `/home/user2/statusReport`; and under Windows: `C:\Documents\myprojectdirectory\README.txt`. (Windows uses backslashes instead of slashes to separate path names and a slightly different naming scheme.) Examples in this course follow the Linux-like naming.

The term “path” comes from the fact that this way of referring to files correspond to following a path in a tree that represents the hierarchy of files on your file system. In the file system displayed below (from Oracle tutorial[8]), a file has the path `/home/user1/foo`, for example.

[4] https://tutorial.djangogirls.org/en/intro_to_command_line/
[5] https://en.wikipedia.org/wiki/Arrow_keys
[6] https://www.lamsade.dauphine.fr/~bnegrevergne/ens/Unix/static/TP_Shell_Unix.pdf
[7] https://git-scm.com/docs/git-config
[8] https://docs.oracle.com/javase/tutorial/essential/io/path.html

Figure 1. A tree representing a file system



A file or a directory can also be referred to using a path that is *relative* to another directory. For example, relative to the directory `/home/user2/`, a relative path for the file in the above example is `statusReport`. A path relative to `/home/` is `user2/statusReport`. Relative paths can also use `..` segments to "climb up" one level in the hierarchy. That is, relative to `/home/user2/bar/`, a path of the example file is `../statusReport`. The mechanism for referring to directories using relative paths is similar. For example, a relative path to refer to the directory `/home/user2/bar/`, relative to `/home/`, is `user2/bar/`. A relative path never starts with a `/`, an absolute path always does (this is also true under Windows, if replacing `/` with \ and neglecting the drive letter).

Referring to some file or directory as an argument of a command typed in a terminal can usually be done using its absolute path or its path relative to your current directory. For example, if you are in the directory `/home/user2/`, you can use both `/home/user2/statusReport` or `statusReport` to refer to the same file.

You can use the special name `.` to refer to the current directory. For example, you can use `ls somepath` to list the content of the directory specified by `somepath`, and thus typing `ls .` lists the content of your current directory.

This course uses Linux-like commands (in particular, uses forward slashes to separate paths), which you should be able to use in any of the environments listed here above: Git BASH emulates a Linux environment and PowerShell authorizes[9] this use.

2.5.1. Exercice

Open a shell, navigate (using `cd`) to some directory of your choice `D1`. Use `ls` to list the content of a directory `D2` that is not a subdirectory of `D1`, using an absolute name for `D2`, then using a relative name. Still from `D1`, create a file in `D2`, using `touch`, by using an absolute file name as argument, then using a relative file name as argument.

[9] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_path_syntax

3. Git

3.1. Presentation

Présentation[10]

Git

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 15 février 2022

LAMSADE UMR CNRS 7243 — DAUPHINE UNIVERSITÉ PARIS — PSL RESEARCH UNIVERSITY

Présentation Commits Branches Serveurs distants Divers

1 / 12

Présentation Commits Branches Serveurs distants Divers

Git

- Contrôle de version (VCS, SCM)
  - Conserver l'historique
  - Fusionner des changements parallèles
- Un ou plusieurs contributeurs
- Pour tous types de projet : code, images, présentations, article...
- VCS souvent centralisés : historique sur un serveur distant
- Git ?
- Créé par ?

2 / 12

Présentation Commits Branches Serveurs distants Divers

Git

- Contrôle de version (VCS, SCM)
  - Conserver l'historique
  - Fusionner des changements parallèles
- Un ou plusieurs contributeurs
- Pour tous types de projet : code, images, présentations, article...
- VCS souvent centralisés : historique sur un serveur distant
- Git ? Local (!) ; usage local, centralisé ou distribué ⇒ tout le monde a une copie complète de l'historique
- Créé par ?

2 / 12

Présentation Commits Branches Serveurs distants Divers

Git

- Contrôle de version (VCS, SCM)
  - Conserver l'historique
  - Fusionner des changements parallèles
- Un ou plusieurs contributeurs
- Pour tous types de projet : code, images, présentations, article...
- VCS souvent centralisés : historique sur un serveur distant
- Git ? Local (!) ; usage local, centralisé ou distribué ⇒ tout le monde a une copie complète de l'historique
- Créé par ? Linus Torvalds (?)

2 / 12

Présentation Commits Branches Serveurs distants Divers

Git

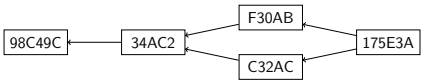
- Contrôle de version (VCS, SCM)
  - Conserver l'historique
  - Fusionner des changements parallèles
- Un ou plusieurs contributeurs
- Pour tous types de projet : code, images, présentations, article...
- VCS souvent centralisés : historique sur un serveur distant
- Git ? Local (!) ; usage local, centralisé ou distribué ⇒ tout le monde a une copie complète de l'historique
- Créé par ? Linus Torvalds (?) Créateur du noyau Linux

2 / 12

[10] https://github.com/oliviercailloux/java-course/raw/main/Git/Pr%C3%A9sentation/presentation.pdf

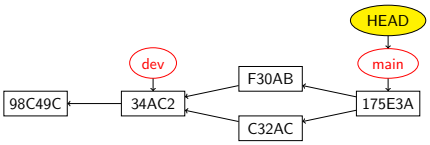
Commits et historique

- Blob : capture d'un fichier à un moment donné
- Commit : identifié par un hash SHA-1
  - Contient : structure de répertoires ; *blobs* ; auteur...
- Histoire : un DAG de « commits »
- Conservée dans un *dépôt* (repository)



Circuler dans l'historique

- Références git (git refs) : branches et références spéciales
- Références pointent vers des commits ou vers d'autres refs
- Branche : pointe un commit
- HEAD : pointe un commit, et généralement une branche ; appelés *actuels*
- Indique le commit d'où est issu la version actuelle
- Circuler en utilisant la commande `checkout <branche>`



Work dir (WD)

- Histoire conservée *localement* dans `.git` à la racine du projet
- WD (« work dir ») : version du projet (fichiers et sous-répert.)
- Interaction avec sous-rép. `.git` via commandes git

```
/root
    /.git
    /rép1
        /fich1
    /fich2
```


Préparer un commit

```
Work dir          Index          HEAD
  /rép1                            /rép1
    /fich1'                          /fich1
  /fich2'        /fich2'           /fich2
  /fich3
```

- *Index* : changements à apporter au prochain commit
- *HEAD* : état capturé dans le commit référencé
- Initialisation nouveau dépôt ?
- Juste après un commit ?

Préparer un commit

```
Work dir          Index          HEAD
  /rép1                            /rép1
    /fich1'                          /fich1
  /fich2'        /fich2'           /fich2
  /fich3
```

- *Index* : changements à apporter au prochain commit
- *HEAD* : état capturé dans le commit référencé
- Initialisation nouveau dépôt ? Index et HEAD vides
- Juste après un commit ?

Préparer un commit

```
Work dir          Index          HEAD
  /rép1                            /rép1
    /fich1'                          /fich1
  /fich2'        /fich2'           /fich2
  /fich3
```

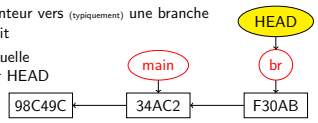
- *Index* : changements à apporter au prochain commit
- *HEAD* : état capturé dans le commit référencé
- Initialisation nouveau dépôt ? Index et HEAD vides
- Juste après un commit ? Index vide

Préparer un commit : commandes

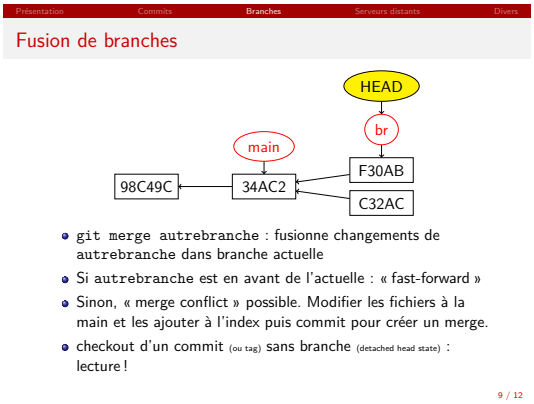
- `git add fichier` : blob mis dans index (« staged »)
- `git status` : liste untracked, tracked-modified, staged
- `git status --short` (sauf merge conflict) : idx VS HEAD ; WD VS idx.
- `git diff` : WD VS index
- `git diff --staged` : index VS HEAD
- `git commit` : commenter et expédier ! (Renvoie son id SHA-1)
- `git commit -v` : voir l'index en détail
- NB : commit bouge la branche actuelle

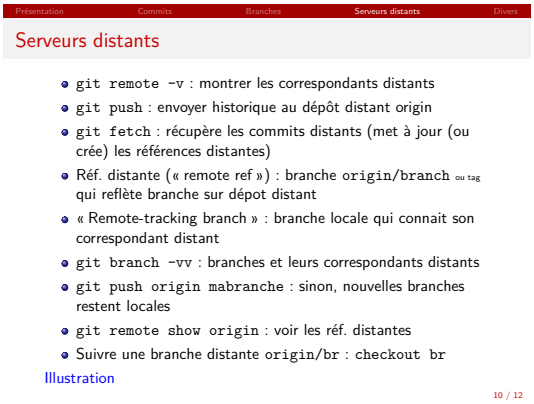
Branches et HEAD

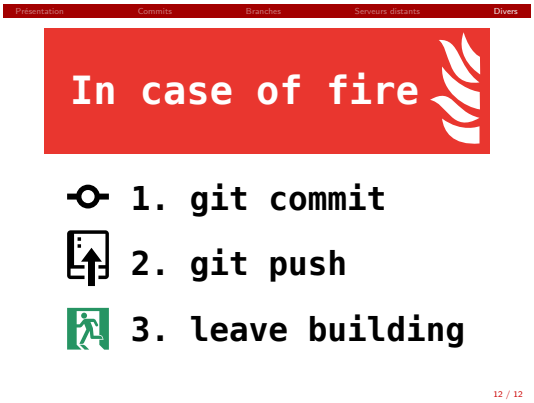
- Branche : pointeur vers un commit
- HEAD : pointeur vers (typiquement) une branche et un commit
- Branche actuelle désignée par HEAD



- commit : avance HEAD et branche actuelle
- `git branch truc` : crée branche `truc`. HEAD inchangé !
- `git checkout truc` : change HEAD et met à jour WD







3.2. Introduction

Prerequisites:

- Install git[11].
- We will start learning with the command line interface of git. If you are not used to using a shell, read the Shell[12] document. Under Windows, use Git BASH which provides completion. (Power users may prefer to use PowerShell[13] with posh-git.)

Then see:

- présentation[14] (or read below),
- configure[15] git,
- step-by-step exercices[16],
- Git branching 1[17], Git branching 2[18], Git branching 3[19]
- best practices[20].

[11] https://git-scm.com/download
[12] https://github.com/oliviercailloux/java-course/blob/main/Git/Shell.adoc
[13] https://www.develves.net/blogs/asd/articles/using-git-with-powershell-on-windows-10/
[14] https://raw.githubusercontent.com/oliviercailloux/java-course/main/Git/Pr%C3%A9sentation/presentation.pdf
[15] https://github.com/oliviercailloux/java-course/blob/main/Git/README.adoc#configure-git
[16] https://github.com/oliviercailloux/java-course/blob/main/Git/Step-by-step.adoc
[17] https://github.com/oliviercailloux/java-course/blob/main/Git/Git%20branching%201.adoc
[18] https://github.com/oliviercailloux/java-course/blob/main/Git/Git%20branching%202.adoc
[19] https://github.com/oliviercailloux/java-course/blob/main/Git/Git%20branching%203.adoc
[20] https://github.com/oliviercailloux/java-course/blob/main/Git/Best%20practices.adoc

3.3. Learning the basics

There are two ways to learn the basics of Git: the frustrating and long way, and the nice and short way. The frustrating and long way is the one you find yourself into if you do not read anything about git (because you do not have time) and just try to deal with it by running commands that you found on the web, that you do not fully understand, that you supposed would achieve just what you need, and that instead created a mess that you ignore how to repair.

To save time, read the Pro Git[21] book. For the basics, you really only need to read the following sections.

- 1.3[22] What is Git?
- 1.6[23] Getting Started - First-Time Git Setup
- 2.1[24] Getting a Git Repository
- 2.2[25] Recording Changes to the Repository
- 2.3[26] Viewing the Commit History
- 2.5[27] Working with Remotes
- 3.1[28] Git Branching - Branches in a Nutshell
- 3.2[29] Git Branching - Basic Branching and Merging
- 3.5[30] Git Branching - Remote Branches

Hint: do not try to remember all the shortcut commands and options git provides. You just need those ones: `git config --global` … (just for the initial configuration); `git clone <url>` or `git init` to start the fun; `git status`, `git diff`, `git add <files>`, `git commit` and `git merge` to enrich your local history; `git branch <name>` to create branches; `git log` and `git checkout <branch/commit>` to navigate your history; `git fetch` and `git push` to synchronize with your remote repository. You can learn the rest when and if you need it.

3.4. Configure git

Git can be configured by associating string values to “options”. An option can be configured locally (for a given repository) or globally (for every time you use git on that system).

- Type `git config --global --list` to see which options are currently configured globally

Here we want to associate your name as a value to the option `user.name`. Git will use this to sign your commits.

- Type `git config --global --get user.name` to see the value currently associated to the option `user.name`
- Type `git config --global --add user.name MyUserName` to associate the value `MyUserName` to the option `user.name`

[21] https://git-scm.com/book
[22] https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F
[23] https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
[24] https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository
[25] https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
[26] https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History
[27] https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes
[28] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[29] https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging
[30] https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches

- Check again the value currently associated to the option `user.name`

For more info: initial setup[31]; GitHub usage of `user.name`[32] and `user.email`[33]

3.5. About authentication on GitHub

Authentication fails if you use your GitHub password. You must use a personal access token instead. See Cloning with HTTPS URLs[34], follow the instructions to create a “fine-grained personal access token”.

3.6. References

- The git Cheat sheets[35],
- The git name[36].
- Learn Git Branching tutorial[37] and live demo[38]
- Git-SCM[39]: Videos; Cheat Sheets; Book Pro Git[40] (free, as in speech and beer)
- Videos (I haven’t watched any of those): see Git-SCM videos[41]; Videos by Tower[42]
- Working with git: A quick and useful guide[43] about workflow on GitHub; a branching model[44]; prefer fetch then merge[45] to pull; the scout pattern[46] for merging
- GUIs: I recommend using the one integrated with your IDE; other options include Git Cola[47] (in particular `git-cola dag`); I’ve been recommended GitKraken[48] (but it is only free for public repos[49]; or through GitHub Student Pack[50])

[31] https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
[32] https://docs.github.com/en/get-started/getting-started-with-git/setting-your-username-in-git#about-git-usernames
[33] https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/setting-your-commit-email-address
[34] https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls
[35] https://github.github.com/training-kit/
[36] https://git.wiki.kernel.org/index.php/Git_FAQ#Why_the_.27Git.27_name.3F
[37] https://learngitbranching.js.org/
[38] https://learngitbranching.js.org/?NODEMO
[39] https://git-scm.com/
[40] https://git-scm.com/book
[41] https://git-scm.com/videos
[42] https://www.git-tower.com/learn/git/videos
[43] https://guides.github.com/introduction/flow/
[44] https://nvie.com/posts/a-successful-git-branching-model/
[45] https://longair.net/blog/2009/04/16/git-fetch-and-merge/
[46] http://think-like-a-git.net/sections/testing-out-merges/the-scout-pattern.html
[47] https://git-cola.github.io/
[48] https://www.gitkraken.com/
[49] https://www.gitkraken.com/pricing#git-gui-features
[50] https://help.gitkraken.com/gitkraken-client/gitkraken-edu-pack/