A Java course

Table of Contents

1. Presentation

Présentation [1]

from-pdf::Présentation du cours Objet/presentation.pdf[end=-1]

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.

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

- 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).
- (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.

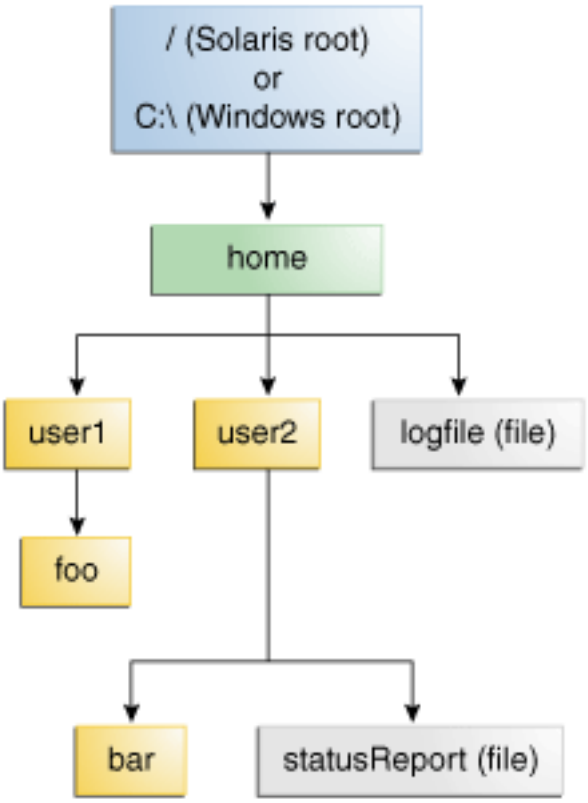
A file or a directory stored on your hard disk has an absolute “path” (or, less technically, “name”) that refers to it unambiguously. For example, under Linux (or MacOS): `/home/user2/statusReport`; and under Windows: `C:\Documents\myprojectdirectory\README.txt`. (Win-

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

dows 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.

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.

[8] https://docs.oracle.com/javase/tutorial/essential/io/path.html
[9] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_path_syntax

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.

3. Git

3.1. Presentation

Présentation [10]

from-pdf::../Git/Présentation/presentation.pdf[end=-1]

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].

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.

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

- 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 (you will see an (almost) empty list if you haven’t configured anything yet)

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` (which will probably be empty at first)
- Type `git config --global --add user.name MyUserName` to associate the value `MyUserName` to the option `user.name`
- 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”.

[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
[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

3.6. References

- The git Cheat sheets [35],
- The git name [36].
- Learn Git Branching tutorial [37] and free play [38] (alternative [39])
- Git-SCM [40]: Videos; Cheat Sheets; Book Pro Git [41] (free, as in speech and beer)
- Videos (I haven't watched any of those): see Git-SCM videos [42]; Videos by Tower [43]
- Working with git: A quick and useful guide [44] about workflow on GitHub; a branching model [45]; prefer fetch then merge [46] to pull; the scout pattern [47] for merging
- GUIs: I recommend using the one integrated with your IDE; other options include Git Cola [48] (in particular `git-cola dag`); I've been recommended GitKraken [49] (but it is only free for public repos [50]; or through GitHub Student Pack [51])

3.7. Step-by-step exercices

Some step by step exercices for git starters.

3.7.1. Local repository

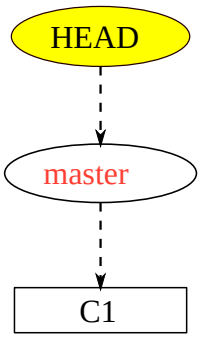
3.7.1.1. First commit

- Create a new directory `project` and put there a file `start.txt` with content `hello`.
- Initialize a new git repository in your directory.
  - Hint: check in the slides, or in the Cheat sheets, how to do this.
- Put `start.txt` in the index. Modify its content so that it contains `hello2`. Observe (using a git command) the difference between the version of this file in the workdir, in the index, and in the repository (all three should be different). Now place your new version of `start.txt` in the index.
- Send your first commit. (Where is it sent?) Check that your current situation conforms to Figure 2, "Right after the first commit" .
  - Hint: try both commands `git log` and `git log --graph --oneline --decorate --all`, understand how they differ.

[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://onlywei.github.io/explain-git-with-d3/
[40] https://git-scm.com/
[41] https://git-scm.com/book
[42] https://git-scm.com/videos
[43] https://www.git-tower.com/learn/git/videos
[44] https://guides.github.com/introduction/flow/
[45] https://nvie.com/posts/a-successful-git-branching-model/
[46] https://longair.net/blog/2009/04/16/git-fetch-and-merge/
[47] http://think-like-a-git.net/sections/testing-out-merges/the-scout-pattern.html
[48] https://git-cola.github.io/
[49] https://www.gitkraken.com/
[50] https://www.gitkraken.com/pricing#git-gui-features
[51] https://help.gitkraken.com/gitkraken-client/gitkraken-edu-pack/

- Also: install Git Cola [52] and use the command `git-cola dag` to view your history graphically.

Figure 2. Right after the first commit

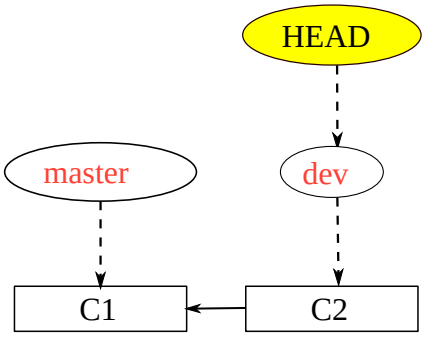


3.7.1.2. First branch

You have got a bold idea to solve some problem in your project. But you're not so sure it will work. You want to create a branch and modify things there in the meantime.

- Create a branch `dev`. Commit in this branch a file `bold.txt` containing `try 1`. Thus, your new commit contains two files (`bold.txt` and `start.txt`). Check that your current situation conforms to Figure 3, "A commit and another commit in a branch" .

Figure 3. A commit and another commit in a branch



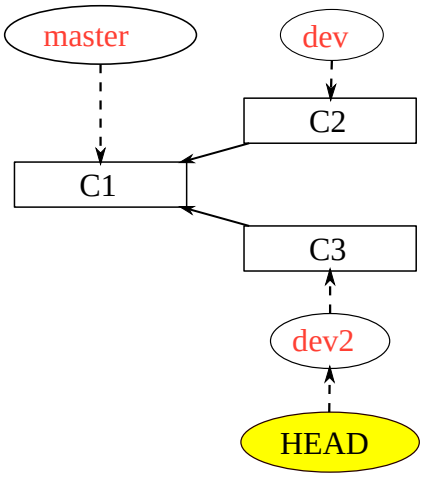
3.7.1.3. Second branch

Now that this (huge) work is done, you want to try an alternative approach. You're also not sure, so you'll do it in a different branch.

- Starting back from `master`, create a branch `dev2`. Commit in this branch a file `bold.txt` containing `alternative approach` and a file named `supplement.txt` containing `hello suppl`. Check that your current situation conforms to Figure 4, "Two branches plus master" .

[52] https://git-cola.github.io/

Figure 4. Two branches plus master



3.7.1.4. Merge

Thinking about it, you decide to merge both your ideas into `master`.

- Go back to branch `master` and merge `dev` into it. Before doing it, try to predict the resulting situation. Such a merge is called a fast-forward, try to guess the definition of this concept.
- Now let's merge `dev2` into `master` (now containing the changes from `dev`). First predict what situation will result. Try it and obtain a final result with everything merged into the `master` branch.

(Solutions diagrams are here: Merged1 [53], Merged2 [54]. Don't cheat! Solve the exercice before looking.)

3.7.2. Remote repository

- Clone your local repository into another directory of your same computer, in order to create another repository. Let's call your original repository R1, your new one (cloned) R2, and the respective work directories WD1 and WD2. This way we simulate multiple users, using only your computer.
  - Hint: proceed as if cloning from an url, but give it instead your local path to your repository.
- Now R2 should have remote repository named `origin` pointing to R1, and a remote-tracking branch `master`. (The term "remote" is ill-suited here, generally your remote would effectively be located on another computer.) Use git commands to see what `origin` refers to, as well as `master` and `origin/master`.
- Change the content of the file `bold.txt` into WD1 (add your first name). Commit into R1.
- Fetch the new informations from R1 into R2. Check that the last commit from R1 exists in the history of R2. Observe the difference between `git log` and `git log --all`.
- Predict what `master` and `origin/master` refer to in R2, and check. What is in the file `bold.txt` in R2? Why?
- Merge into R2 your last changes from R1.

3.7.3. Remote repository on GitHub

- Create a new remote remository, RG, on GitHub [55]. Connect R1 to RG.

[53] https://github.com/oliviercailloux/java-course/blob/main/Git/Merged1.svg
[54] https://github.com/oliviercailloux/java-course/blob/main/Git/Merged2.svg
[55] https://github.com/

    - Hint: understand and adapt the instructions given by GitHub.
- Send your local informations to RG. Check with a browser that they have reached GitHub.
- Clone RG into a new directory, R3. Modify a file and commit into R3. Send it to RG. Check online.
- Modify something in R1, commit. Try to send it to RG. Why is it refused? How can you effectively send your last modification online? Send it to your RG repository.
- Send your two branches `dev` and `dev2` to RG as well. Check that you see them online.

3.8. Git branching 1 exercice

A modified version of a graded exercice that was given a few years ago.

- Fork [56] this project [57] and clone your version locally.
- Check out the commit whose SHA-1 identifier *ends* with `6a341`. We will call this the "starting" commit.
- Create and switch to a new branch, `my-branch`, departing from the starting commit.
- Change something in the `pom.xml` file. Commit into your branch (so that your commit has the starting commit as parent).
- Bring the changes of the commit that originally followed the starting commit into `my-branch` (by merging).
- Do not forget to send all your commits and your branch to the GitHub repository. Check that you see them online.

You will be graded as follows.

- One of the commits that you created must have the starting commit as parent, and be an ancestor of the final position of `my-branch`.
- One of the commits that you created must have two parents: a commit that you created; and one that you did not create; both parents having as parent the starting commit. The commit must be the final position of the branch `my-branch`, or be an ancestor of it.
- You must have changed the pom in your commit that has the starting commit as a parent.
- Commits created using the GitHub web site do not count.

4. Graded exercices

This document explains how to submit your solutions for the graded exercices of this course.

- Each graded exercice uses its own "GitHub assignment" (a mechanism proposed by GitHub classroom to dispatch exercices in a class).
- Each assignment has its own name and link. Start by clicking the link and accept the assignment
    - This creates a GitHub repository that you own in my GitHub "organisation", `oliviercailloux-org`
    - Your repository will be created as `https://github.com/oliviercailloux-org/assignmentName-yourGitHubUserName`

[56] https://docs.github.com/en/get-started/quickstart/fork-a-repo
[57] https://github.com/oliviercailloux/google-or-tools-java

    - This repository will be initialized with some data
- Clone your new repository
    - You have to work on your local repository, and regularly push your additions to the remote
    - Both you and I have access to your remote repository. I use this access to grade your work
    - Note that I have no access to your local repository, so, whatever you do not push, I cannot grade
- Do whatever the assignment asks you to do (for example, add some code to some files, create new files, …) in your local repository
    - You can use pre-existing resources on your computer (course, your notes…) and on the internet. You can search on Google, StackOverflow, or any other engine that returns pre-existing answers. It is *absolutely forbidden* to use any resource that is generated on the fly, such as large language models (ChatGPT or others), post a question on a forum, …
- Push your work to your remote
    - Push regularly, do not wait the end of the allowed time, so as to prevent last minute problems
    - Remember that you need to push all branches that you want me to see
- Hint: Check with the GitHub web interface that your commits and branches have reached the remote repository
- Your last commit on time is graded. Also, your commits overtime are graded, and penalized linearly according to the delay, so that five minutes delay is penalized 100%. The best grade is kept. So, pushing after the delay can only improve your grade.
- When you are done, you may exit the classroom only if you push a tag `END` on the latest commit that you want to be considered.

4.1. Configuration

It is important that your git `user.name` equals [58] (exactly) your GitHub username, so that I can associate your contributions to you. This will be important when working on projects. I will also grade whether you configured your git correctly even before projects, as part of most graded exercice.

4.2. The `training` assignment

- Here is the link [59] for the “training” assignment.
- You should be able to predict the name of the GitHub repository that will be created when you will accept this assignment.
- You can use this to practice as much as you want.
- This repository will not be used to grade your work.
- In this assignment, you start with a file named `hello.txt` which contains `Hello, world.`

If you want to start fresh (for example, before starting a new exercice), delete [60] your “training” repository.

[58] https://github.com/oliviercailloux/java-course/blob/main/Git/README.adoc#Configure-git
[59] https://classroom.github.com/a/82sB-Te7
[60] https://docs.github.com/repositories/creating-and-managing-repositories/deleting-a-repository

4.3. Exercice: hello world

Accept the `training` assignment (link here above) and translate the content of the file `hello.txt` to say hello in another language of your choice. One prestige point to you if it is a language that I have not yet seen in other student works and you did not use external help for the translation. Make sure this is visible on your remote repository.

4.4. Permissions

You may add methods to classes and you may add classes but you may not change the provided headers (you may not add parameters for example) or change the interfaces. Moreover, your unit tests have to test the contracts as specified, they may not use supplementary methods or classes.

4.5. Trying again

I will sometimes grant a second chance. In those cases, the new grade will, unless noted otherwise, consider the number of lines that you changed in between both attempts. Here is a related comment from a previous year (in French).

Je vous attribuerai alors comme note une combinaison de l'ancienne et de la nouvelle note, dépendant de l'ampleur de vos modifications. Cette combinaison dépendra typiquement du **nombre de lignes modifiées**. Elles sont comptées en considérant la différence ensembliste symétrique entre les lignes de votre fichier original et les lignes de votre nouveau fichier (additionnées sur tous les fichiers modifiés). Les changements d'ordre entre les lignes ne comptent donc pas. Attention, les différences de simples caractères tels que les espaces, les tabulations, les fins de lignes comptent comme des différences comme les autres (voir end of lines [61] et formatting [62]). Si formater [63] votre code augmente son nombre de lignes, ces lignes supplémentaires sont ajoutées au nombre de lignes différentes. Lisez comparing commits [64] pour vérifier visuellement que vous n'introduisez pas de changements inutiles.

5. Syntax

Here we learn the basic syntax of Java, and train using jshell.

See: présentation [65].

5.1. Hint

In jshell, as with many shells, use ↑ (keyboard up arrow [66]) to recall previous commands.

5.2. Short exercices

You can achieve all these tasks with only the syntax seen in this presentation, in particular, without using explicit conversions from String to integer or conversely. Try it!

1. Assign your first name to a variable and use it to print `Hello <your-first-name>.`
2. Assign your age to a integer variable and use it to print `<your-age> is my age!`
3. Assign each digit of your age to different integer variables; use them to print `<your-age> is my age!`

[61] https://github.com/oliviercailloux/java-course/blob/main/Git/Best%20practices.adoc#end-of-lines
[62] https://github.com/oliviercailloux/java-course/blob/main/Style/Code.adoc#formatting
[63] https://code.visualstudio.com/docs/java/java-linting
[64] https://docs.github.com/en/github/committing-changes-to-your-project/comparing-commits
[65] https://raw.githubusercontent.com/oliviercailloux/java-course/master/Syntax/Pr%C3%A9sentation/presentation.pdf
[66] https://en.wikipedia.org/wiki/Arrow_keys

4. Create a boolean variable whose value is `true`, another one whose value is `false`, and use them to create an expression that evaluates to `true` and one that evaluates to `false`

5. Assign any number you like to an integer variable. Create an expression that evaluates to `true` if, and only if, that variable equals your age. (Test it by changing the value of the variable and running the expression again.)

6. Assign the value `"true"` to a variable of type `String`. Create an expression that evaluates to `true` if, and only if, the value of that variable is equal to the String `"true"`. Use the method `s1.equals(s2)` to test for equality of two strings `s1` and `s2`.

5.3. Methods

1. Assign 4659 and 23 to variables, and show the result of multiplying these variables

2. Compute the greatest divisor of 4659 that is different than 4659 (use the `%` operator and a loop) (to check your answer, look at its factors [67])

3. Define a method that returns the greatest divisor of 4659 that is different than 4659

4. Define a method `greatestDivisor` that accepts an integer parameter and returns its greatest divisor except itself; use it to show the greatest divisor of 4659.

5.4. Classes

In a text editor, define a class `MyMathClass` containing a method that accepts an integer parameter and returns its greatest divisor except itself, and a method that returns the smallest divisor of its parameter except one; and a class `MyBooleanClass` containing a method `xor` returning `true` iff exactly one of its two boolean parameters is `true`. Copy-paste this in jshell and call all these methods from jshell.

5.5. More about classes

Define a class `MovingThing` that has a static variable `currentSpeed` and a static variable `totalLength`; and a static method declared as `static void move(double time)` that computes the distance an object moving at `currentSpeed` moves in `time` and adds it to the current total length. Copy-paste this in jshell and use your class to check that it works well. For example, after calling `MovingThing.currentSpeed = 10` then `MovingThing.move(50)`, the variable `MovingThing.totalLength` should have a value of `500`.

Once this works, make all your variables private so that nobody but your own class can touch them. Modify your class to make it still useable by providing appropriate methods to get or set their values as required. Make sure to restrict access as much as reasonable, for example, an external user should not be able to modify the `totalLength` directly.

5.6. More exercices

Our goal here is to reach exercice 3.6 (here below), which requires to print out all numbers that have the maximum number of divisors. This requires to achieve exercice 3.2 first. Optionally, you may also want to attempt exercices 3.1, and 3.8 and 3.9 if you want some more advanced challenges.

- 3.2 [68]
- 3.6
- EE3.1, EE3.8, 3.9 (optional)

[67] https://www.wolframalpha.com/input?i2d=true&i=factor%5C%2840%294659%5C%2841%29
[68] https://math.hws.edu/javanotes/c3/exercises.html

The following two subsections are here for reference; we will come back to it later in the course.

5.7. Varargs

See Oracle doc about the varargs syntax: Varargs [69]

Exercice: call the static method `String. format` [70]`()` with no arguments, then with only one string as argument, then two strings, then three strings. Predict which calls will be accepted by the compiler. Explain in each case what parameters are effectively passed to the method, by considering the method declaration (hint: exactly two parameters are passed for each permissible call).

5.8. More syntax

- Interfaces [71]
- Classes as concept [72]: instance variables and instance methods
- Inheritance [73]: extending interfaces and classes

5.9. References

- jshell [74]
- The peculiarities of the JShell [75]

[69] https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html
[70] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#format(java.lang.String,java.lang.Object.%2E.)
[71] https://github.com/oliviercailloux/java-course/blob/main/Syntax/Interfaces.adoc
[72] https://github.com/oliviercailloux/java-course/blob/main/Syntax/Classes%20as%20concept.adoc
[73] https://github.com/oliviercailloux/java-course/blob/main/Syntax/Inheritance.adoc
[74] https://docs.oracle.com/en/java/javase/13/docs/specs/man/jshell.html
[75] https://arbitrary-but-fixed.net/teaching/java/jshell/2017/12/14/jshell-peculiarities.html