

Introduction à Java

Un langage compilé

Java est un langage de programmation généraliste orienté objet. Nous n'allons pas copier Wikipedia ici (voir [https://fr.wikipedia.org/wiki/Java_\(langage\)](https://fr.wikipedia.org/wiki/Java_(langage))), mais relever un point important par rapport aux langages que vous avez rencontrés jusqu'à maintenant (HTML, JavaScript, PHP). Java est un langage **statiquement typé** et **compilé** :

les variables sont définies avec un type de données (une variable "String" ne pourra contenir que des valeurs de type "String") ;

les fichiers sources devront être compilés (sorte de traduction) en fichiers "classes" avant d'être exécutés.

Cela oblige à bien définir les types des variables et à subir l'étape supplémentaire de compilation mais apporte plusieurs avantages :

- certaines erreurs sont détectées lors de la compilation et non lors de l'exécution (le code est plus "sûr") ;
- des optimisations plus poussées peuvent être mises en œuvre ;
- les classes générées sont indépendantes du système.

Mais... même dans le développement, tout n'est pas binaire ! Même si cela est moins visible, lors de l'exécution d'un code JavaScript (on parle alors de code interprété car "lu" au fur et à mesure des besoins), ce code peut être localement compilé pour être exécuté et accéléré. Le principe est juste moins poussé que pour le langage Java et la différence est historique, les langages interprétés étaient alors exécutés comme nous le faisons en lisant le code ! De même, des extensions à JavaScript peuvent apporter du typage statique. Et dans l'autre sens, beaucoup de frameworks rendent l'opération de compilation en Java quasi invisible... On peut donc relativiser ces différences !

Hands on!

Installation du JDK

Pour pouvoir utiliser le langage Java (exécuter des programmes Java), il faut avoir installé l'environnement d'exécution. Cet environnement apporte la "machine virtuelle" (le programme qui exécute le code compilé), mais aussi les librairies de base (afficher un texte à l'écran, ouvrir un fichier, ...). Le JRE (Java Runtime Environment) suffit pour cela. Dans notre cas, nous allons également créer des programmes en Java : il nous faut donc installer le JDK - Java Development Kit.

Des extensions existent pour le JDK, selon les librairies de base installées notamment. Nous allons utiliser JavaSE (Standard Edition), version 8 : suivez les instructions depuis <http://www.oracle.com/technetwork/java/index.html>

Un IDE ?

Différents IDE (ou même un simple éditeur de texte) sont utilisables pour écrire des programmes Java. Les références pour Java sont Eclipse (<http://www.eclipse.org/>), IntelliJ IDEA (<https://www.jetbrains.com/idea/>), Netbeans (<https://netbeans.org>)...

Nous n'entrons pas dans le débat des avantages des uns et des autres, mais ces IDE sont en général très complets, permettent le développement de gros projets et sont même parfois imposés dans des équipes pour l'utilisation de leurs fonctionnalités spécifiques. Le revers de la médaille est une courbe d'apprentissage plus raide et des mécanismes qui cachent parfois trop ce qui se passe pour un apprentissage.

Nous faisons donc le choix d'un IDE plus simple, Visual Studio Code. Trois observations :

- Vous pouvez utiliser celui que vous voulez !
- Vous trouverez en général des tutoriels qui utilisent les IDE cités ci-dessus
- Des plugins Java pour Visual Studio Code sont disponibles (voir plus bas)

Hello... World!

Comment y échapper ? Nous allons même en faire 4 versions ! L'objectif est d'appréhender les bases du développement en Java ; nous rentrerons dans le langage et ses principes juste après.

Hello 0

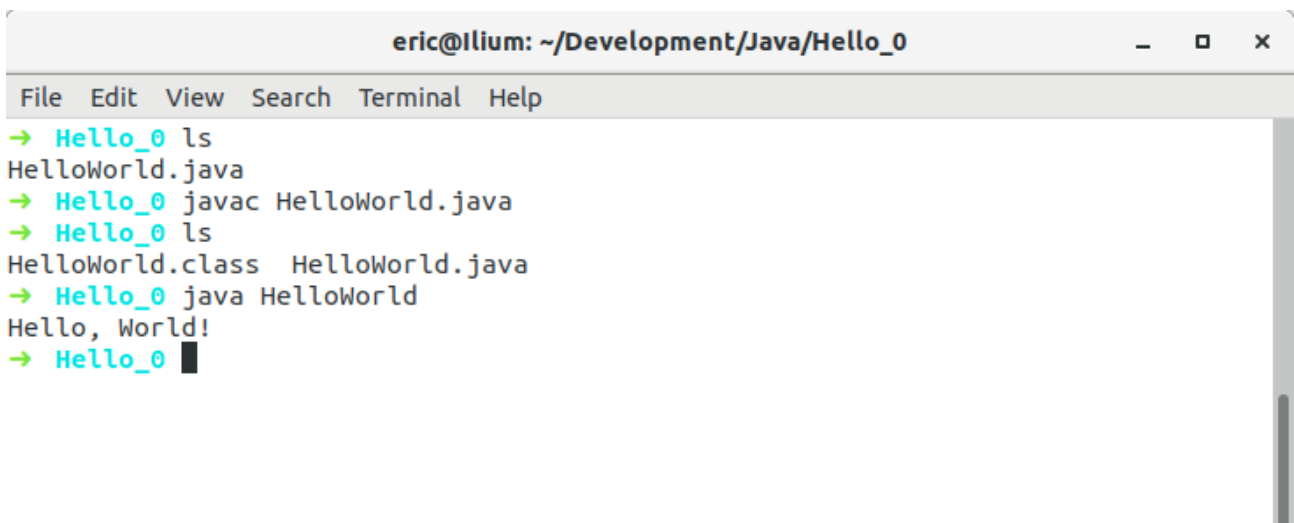
Et oui, Java est comme la plupart des langages de programmation “zero-based” (https://en.wikipedia.org/wiki/Zero-based_numbering).

Créer un répertoire Hello_0, avec un fichier HelloWorld.java qui contient :

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Ce fichier est la définition d’une classe HelloWorld, qui a une méthode statique (attachée à la classe et non aux instances de cette classe) main (nous passons sur les arguments) et qui utilise la méthode statique println de la classe System.out pour faire l’affichage tant attendu. Les principes ne sont pas vraiment différent de la POO en PHP.

Pour exécuter ce programme, il faut d’abord le compiler. Ouvrez un shell (cmd, bash, ...) et positionnez-vous dans le répertoire du fichier. Compilez et exécutez le programme comme ceci :



```
eric@llium: ~/Development/Java/Hello_0
File Edit View Search Terminal Help
→ Hello_0 ls
HelloWorld.java
→ Hello_0 javac HelloWorld.java
→ Hello_0 ls
HelloWorld.class HelloWorld.java
→ Hello_0 java HelloWorld
Hello, World!
→ Hello_0
```

La commande javac (Java Compiler) produit le fichier compilé HelloWorld.class. La commande java HelloWorld exécute ces étapes :

- cherche la classe HelloWorld dans l’environnement (par défaut dans le répertoire courant => définie dans HelloWorld.class
- charge cette classe

- cherche la méthode par défaut, main et l'exécute

Hello ++

L'opérateur de post-incrément est disponible en Java, comme en PHP ou JavaScript !

Créez l'arborescence de répertoires Hello_1/src/main/java/hello. En utilisant Bash, vous pouvez le faire avec la seule commande

```
mkdir -p Hello_1/src/main/java/hello
```

Créez dans le répertoire hello les deux fichiers Greeter.java et HelloWorld.java qui contiennent respectivement :

```
package hello;

public class Greeter {
    public String sayHello() {
        return "Hello world!";
    }
}
```

```
package hello;

public class HelloWorld {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

Le programme utilise maintenant deux classes et peut être compilé et exécuté comme suit :

```
eric@ilium: ~/Development/Java/Hello_1/src/main/java
File Edit View Search Terminal Help
→ hello ls
Greeter.java HelloWorld.java
→ hello javac *.java
→ hello ls
Greeter.class Greeter.java HelloWorld.class HelloWorld.java
→ hello java hello.HelloWorld
Error: Could not find or load main class hello.HelloWorld
Caused by: java.lang.ClassNotFoundException: hello.HelloWorld
→ hello cd ..
→ java ls
hello
→ java java hello.HelloWorld
Hello world!
→ java
```

Vous remarquerez dans ce cas que les classes indiquent qu'elles appartiennent au "package" hello - et cela correspond au répertoire hello. Lors de l'exécution, la classe ne peut être trouvée que depuis le répertoire java car la classe cherchée hello.HelloWorld (autrement dit la classe HelloWorld du package hello ne peut être trouvée que dans un répertoire hello).

Cette notion de package est très utile pour organiser son code.

Hello 2 - Je vous l'emballage ?

Dans cette nouvelle version, nous introduisons un outil standard (de fait, il en existe d'autres comme Ant ou Gradle) de "build", Maven. Ces outils permettent en général de gérer des dépendances (modules externes) et le cycle de compilation. Les frontières d'un outil à un autre et surtout d'un langage à un autre sont assez souples. Maven recouvre en quelque sorte Composer (PHP) et Artisan (Laravel) que vous avez rencontrés.

Installez Maven en suivant les instructions du site <http://maven.apache.org/>

Dans le répertoire Hello_2, créez la même arborescence src que dans l'exemple précédent (incluant les fichiers .java). L'utilisation de Maven repose sur un Project Object Model (POM), ici un seul fichier pom.xml, à côté sur répertoire src :

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>fr.campus-numerique-in-the-alps</groupId>
<artifactId>hello</artifactId>
<version>1</version>
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
</project>
```

Ce fichier XML est à peu de choses près un exemple minimal. Les options de source et cible pour le code Java ne sont pas nécessaires, mais règlent bien des problèmes. En effet, un problème récurrent avec les outils de build réside dans les choix implicites. Par exemple, il n'est dit nulle part que le code source est dans le répertoire `src` ! Il est donc préférable de fixer certaines options. Ici, l'encodage des caractères est UTF-8 : vous devez avoir ce même réglage dans votre éditeur. Cela ne pose en général pas de problème pour le code lui-même, mais le contenu des chaînes de caractères risque d'être erroné si l'encodage est mal choisi, en particulier pour les caractères accentués ou spéciaux (voir <https://docs.oracle.com/javase/tutorial/i18n/text/string.html> et <http://utf8everywhere.org/>).

Bref, nous pouvons maintenant utiliser la commande `mvn` pour compiler les différents fichiers `.java` qui pourraient être répartis dans différents répertoires (= packages), avec les bonnes options (encodage, etc) :


```
eric@ilium: ~/Development/Java/Hello_2
File Edit View Search Terminal Help
→ Hello_2 mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building hello 1
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/eric/Development/Java/Hello_2/src/main/resources
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /home/eric/Development/Java/Hello_2/target/classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.904 s
[INFO] Finished at: 2018-03-03T14:33:25+01:00
[INFO] Final Memory: 12M/201M
[INFO] -----
→ Hello_2 ls -R target/classes
target/classes:
hello

target/classes/hello:
Greeter.class HelloWorld.class
→ Hello_2 java -cp target/classes hello.HelloWorld
Hello world!
→ Hello_2
```

mvn compile déclenche la compilation du projet hello, en version 1. Deux classes sont trouvées et compilées, les fichiers .class allant dans le répertoire target/classes. Il faut alors indiquer à Java le (ou les) chemin pour trouver les classes pour l'exécution. On utilise ici l'option -cp (classpath) pour indiquer de chercher dans le répertoire target/classes.

Super, mais si je veux passer mon programme à quelqu'un d'autre ? L'écosystème Java a prévu de pouvoir "packager" ses applications dans une archive, un fichier .jar (aka jarfile, Java Archive). Il s'agit en fait d'une archive (avec quelques méta-données - [https://fr.wikipedia.org/wiki/JAR_\(format_de_fichier\)](https://fr.wikipedia.org/wiki/JAR_(format_de_fichier))). Il existe une commande jar dans le JDK, mais Maven peut s'occuper de tout cela :

```
eric@ilium: ~/Development/Java/Hello_2
File Edit View Search Terminal Help
→ Hello_2 mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building hello 1
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/eric/Development/Java/Hello_2/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/eric/Development/Java/Hello_2/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ hello ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello ---
[INFO] Building jar: /home/eric/Development/Java/Hello_2/target/hello-1.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello ---
[INFO] Installing /home/eric/Development/Java/Hello_2/target/hello-1.jar to /home/eric/.m2/repository/fr/campus-numerique-in-the-alps/hello/1/hello-1.jar
[INFO] Installing /home/eric/Development/Java/Hello_2/pom.xml to /home/eric/.m2/repository/fr/campus-numerique-in-the-alps/hello/1/hello-1.pom
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.886 s
[INFO] Finished at: 2018-03-03T14:41:49+01:00
[INFO] Final Memory: 11M/300M
[INFO] -----
→ Hello_2 java -cp target/hello-1.jar hello.HelloWorld
Hello world!
→ Hello_2 █
```

Maven ici “installe” le projet en créant le jarfile. Ce jarfile peut être passé comme un répertoire pour indiquer un endroit où trouver des classes, via l’option -cp (le “classpath”).

Hello 3 - Quelle heure est-il ?

Les jarfiles permettent donc de packager des classes, créant des packages ou des modules. Le but est évidemment de réutiliser ces packages.

Nous allons maintenant ajouter l’heure (comme dans l’exemple développé sur <https://spring.io/guides/gs/maven/>). Recopiez le projet Hello_2 dans Hello_3 et modifiez HelloWorld.java comme suit :

```
package hello;
```



```
public class HelloWorld {
    public static void main(String[] args) {
        LocalDateTime currentTime = new LocalDateTime();
        System.out.println("The current local time is: " + currentTime);
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

Si vous essayez de compiler avec `mvn compile` (ou directement avec `javac`), vous obtiendrez une erreur “[ERROR] symbol: class LocalDateTime”. En effet, cette classe n’est pas connue. Il faut d’abord l’importer dans notre classe, qui devient :

```
package hello;

import org.joda.time.LocalDateTime;

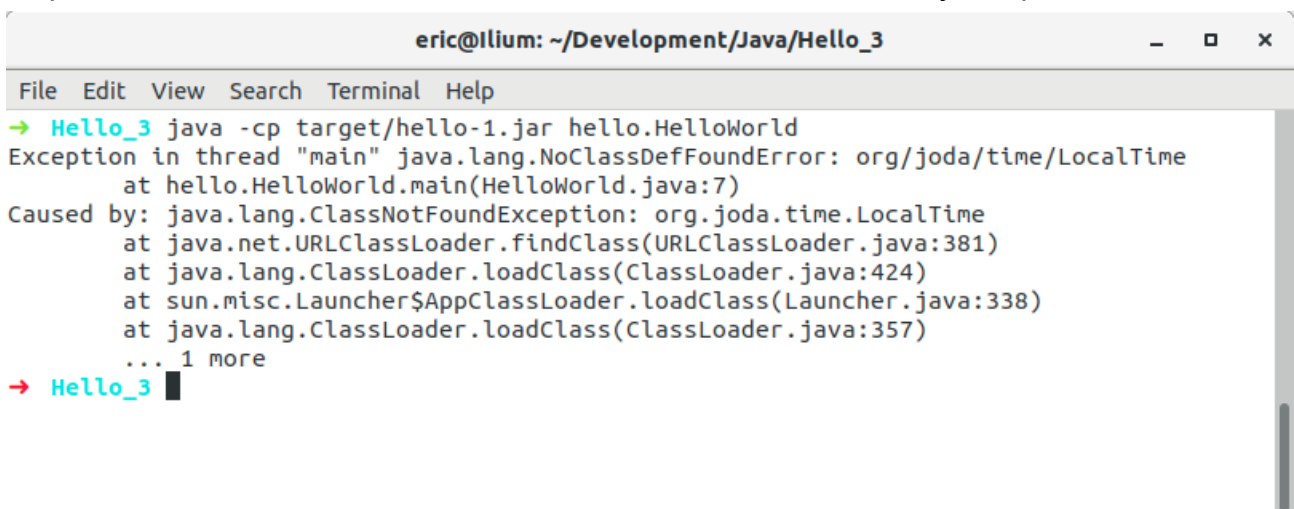
public class HelloWorld {
    public static void main(String[] args) {
        LocalDateTime currentTime = new LocalDateTime();
        System.out.println("The current local time is: " + currentTime);
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

Mais cela ne suffit pas : le package `org.joda.time` ne fait pas partie des packages standard du JDK ! Il nous faut donc résoudre la dépendance. Pour cela, modifions le fichier de définition du projet, `pom.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.campus-numerique-in-the-alps</groupId>
  <artifactId>hello</artifactId>
  <version>1</version>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.9.2</version>
  </dependency>
</dependencies>
</project>
```

La compilation fonctionne : comme Composer, Maven utilise un repository (Maven Central) sur lequel on peut trouver les packages pour la plupart des projets Open Source. On peut alors installer avec `mvn install`. Mais... cela ne suffit toujours pas :



```
eric@ilium: ~/Development/Java/Hello_3
File Edit View Search Terminal Help
→ Hello_3 java -cp target/hello-1.jar hello.HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: org/joda/time/LocalTime
    at hello.HelloWorld.main(HelloWorld.java:7)
Caused by: java.lang.ClassNotFoundException: org.joda.time.LocalTime
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:338)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
→ Hello_3
```

Le Classpath ne contient que le jarfile pour les classes définies dans notre projet et non les classes externes. Si vous avez été curieux, vous avez vu lors de l'étape précédente que les dépendances sont installées localement dans le répertoire `$HOME/.m2` :



```
eric@ilium: ~/Development/Java/Hello_3
File Edit View Search Terminal Help
→ Hello_3 java -cp target/hello-1.jar:/home/eric/.m2/repository/joda-time/joda-time/2.9.2/joda-time-2.9.2.jar hello.HelloWorld
The current local time is: 11:26:55.853
Hello world!
→ Hello_3
```

Nous pouvons facilement envoyer notre package pour un autre projet, mais l'exécution n'est pas très aisée (il faut connaître toutes les dépendances et leur répertoire d'installation). Une solution est de créer un script pour cela, mais les jarfiles peuvent être rendus "executables" en packagant toutes les dépendances dans un même jarfile et en indiquant la classe de démarrage.

Pour cela, nous allons utiliser un plugin Maven : "Shade". Modifiez le fichier de projet :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.campus-numerique-in-the-alps</groupId>
  <artifactId>hello</artifactId>
  <version>1</version>
  <packaging>jar</packaging>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.1</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransfor
mer">
                  <mainClass>hello.HelloWorld</mainClass>
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
```

```
</plugin>

</plugins>
</build>
<properties>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
</project>
```

Et maintenant :



```
eric@ilium: ~/Development/Java/Hello_3
File Edit View Search Terminal Help
→ Hello_3 java -jar target/hello-1.jar
The current local time is: 11:39:34.102
Hello world!
→ Hello_3
```

Tutoriel

Vous pouvez maintenant aborder le tutoriel pour le langage Java d'OpenClassRoom (vous auriez pu avant, mais vous pouvez vous concentrer désormais sur le langage en lui-même). Les exemples sont donnés pour Eclipse, mais vous pouvez suivre dans n'importe quel éditeur (ci-dessous, une annexe sur Visual Studio Code) et utiliser la ligne de commande autant que possible.

<https://openclassrooms.com/courses/apprenez-a-programmer-en-java?status=publishé>

- partie 1 (on peut sauter le point 1, sauf pour ceux qui veulent voir un nouvel environnement)
- partie 2, jusqu'au point 11

Livrable : Hello, again!

Vous allez mettre en œuvre ces récents acquis sur Java. L'objectif est de produire un programme Java, disponible sur Github, avec la documentation décrivant la préparation et l'exécution. Vous utiliserez Maven.

Le programme doit contenir une classe principal dont le seul but est l'exécution du point d'entrée (main). Vous définirez :

- une interface Bonjour avec une méthode sayHello qui renvoie un chaîne de caractères ;
- une classe abstraite Animal, qui implémente l'interface ci-dessus ;
- deux classes qui étendent Animal, Human et Dog.

Créez une instance pour chacune des classes concrètes et illustrez le fonctionnement dans votre main.

Annexe : Visual Studio Code

Vous pouvez installer un plugin pour Java dans Visual Studio Code qui selon nous apporte des aides sans trop de “magie” : <https://code.visualstudio.com/docs/languages/java>
Après installation, la situation précédente quand Joda Time n’était pas installé donne :

