Analyses automatiques (draft)
(mais pas le Test unitaire, autre cours)

Quoi et pourquoi ?

- Nous allons étudier 3 outils/méthodes
  - **Analyse de style** ⇒ CheckStyle
  - **Analyse d'erreurs courantes** ⇒ Spotbugs
  - **Model-checking** (de threads) ⇒ JPF
- Quelles différences (surtout avec du Test Unitaire)
  - Le style est juste une analyse syntaxique (la forme) et pas la sémantique (le fond) ; Même pas une analyse de type
  - Avec SpotBugs, on ne définit pas l'attendu, juste çà recherche des erreurs typiques des humains genre for(int i=0;i<=tab.length;i++){…} ⇒ Array.OutOfBound
  - Le model-checking a pour rôle de tester TOUTES (!!!) les combinaisons possibles d'exécutions et non pas quelques cas comme dans le test. C'est indécidable en général, pire que NP-Complet dans bcp de cas pratique. Mais très utilisé pour rechercher des erreurs si on limite certaines entrées ; On en rediscute après c'est méthode formelle (preuve de programme)

CheckStyle

- https://checkstyle.sourceforge.io/
- Installation et exécution simple (jar) :
  - https://github.com/checkstyle/checkstyle/releases/download/checkstyle-8.31/checkstyle-8.31-all.jar
  - java -jar checkstyle-8.31-all.jar -c config.xml MyClass.java
  - Il existe aussi des mode pour Maven, Ant, etc. ou compiler les sources
- CheckStyle propose aussi d'extraire l'AST (Abstract Syntax Tree) du programme

  java -cp checkstyle-8.31-all.jar com.puppycrawl.tools.checkstyle.gui.Main MyClass.java
- C'est dans « config.xml » qu'on spécifie quels types d'analyses on souhaite effectuer ; 2 exemples sont proposés ici :
  - https://raw.githubusercontent.com/checkstyle/checkstyle/master/src/main/resources/google_checks.xml
  - https://raw.githubusercontent.com/checkstyle/checkstyle/master/src/main/resources/sun_checks.xml

Quelle configurations ?

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
        "-//Checkstyle//DTD Check Conf 1.3//EN"
        "https://checkstyle.org/dtds/configuration_1_3.dtd">
<module name="Checker">
  <module name="JavadocPackage"/>
  <module name="TreeWalker">
    <module name="AvoidStarImport"/>
    <module name="EmptyBlock"/>
    <module name="MethodLength">
      <property name="max" value="10"/>
    </module>
  </module>
</module>
```

Obligatoire pour XML

Début configuration

Load module pour JavaDoc

Parcours de l'AST

Des modules

Un module paramétré

On trouve aussi des trucs tout fait comme « return null » à la place d'un boolean etc.

D’autres modules

- Pleins …
  - Doc ≡ https://checkstyle.sourceforge.io/config.html
  - On y trouve quelques exemples
- Quelques trucs :
  - **Severity**

```
<module name="Translation">
  <property name="severity" value="warning"/>
</module>
```

  - **Id**, pour avoir différencier certains paramétrages

```
<module name="DescendantToken">
  <property name="id" value="stringEqual"/>
  <property name="tokens" value="EQUAL,NOT_EQUAL"/>
    ...
</module>
<module name="DescendantToken">
  <property name="id" value="switchNoDefault"/>
  <property name="tokens" value="LITERAL_SWITCH"/>
    ...
</module>
```

  - Etc.
- **Voyons en pratique...**

SpotBugs

- Find bugs in Java Programs (bytes-codes)
- https://spotbugs.github.io/
- Outil d'analyse **statique** du byte-code pour trouver des bugs courants. *Ne garantie pas l'absence de bugs !*
- Fonctionne par « patterns » (idiomes en français)
- Exemples bugs :
  - Method with Boolean return type returns explicit null
  - RANGE: Array index is out of bounds (Que p*our des patterns simples qu'ils ont implantés donc pas de garantie ...*)
  - Suspicious reference comparison (This method compares two reference values using the == or != operator)
  - Etc. etc. Des dizaines et des dizaines...

Execution

- https://github.com/spotbugs/spotbugs/releases/download/4.0.2/spotbugs-4.0.2.zip

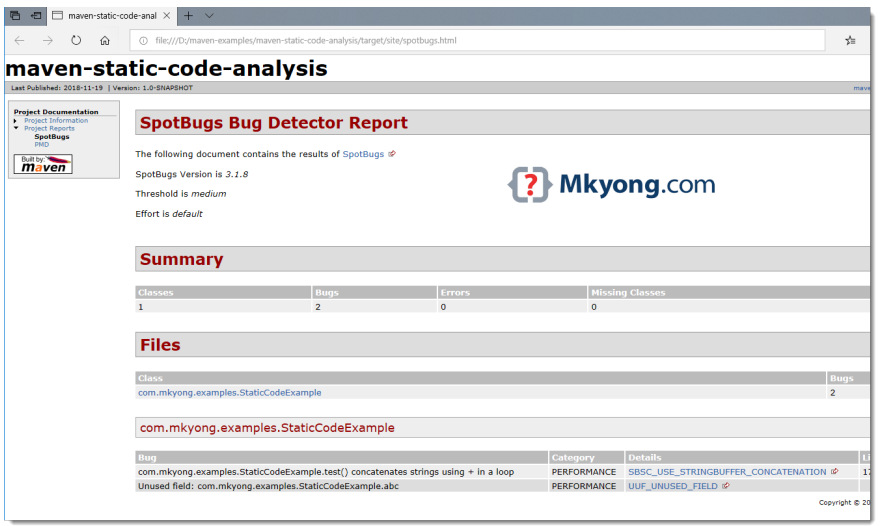
- Unzip puis
  - java [JVM arguments] -jar $SPOTBUGS_HOME/lib/spotbugs.jar
  - Ou $SPOTBUGS_HOME/bin/spotbugs
  - https://spotbugs.readthedocs.io/en/latest/running.html
- Installation Ant, Maven, Grable etc.
  - https://mkyong.com/maven/maven-spotbugs-example/
  - On peut aussi générer des rapports HTML
- Filtre XML
  - https://spotbugs.readthedocs.io/en/stable/filter.html
  - spotbugs -include (ou exclude) myIncludeFilter.xml myApp.jar
  - <Match>

    <Class name="com.foobar.MyClass" />

    <Method name="writeDataToFile" />

    <Bug pattern="OS_OPEN_STREAM" />

    </Match>

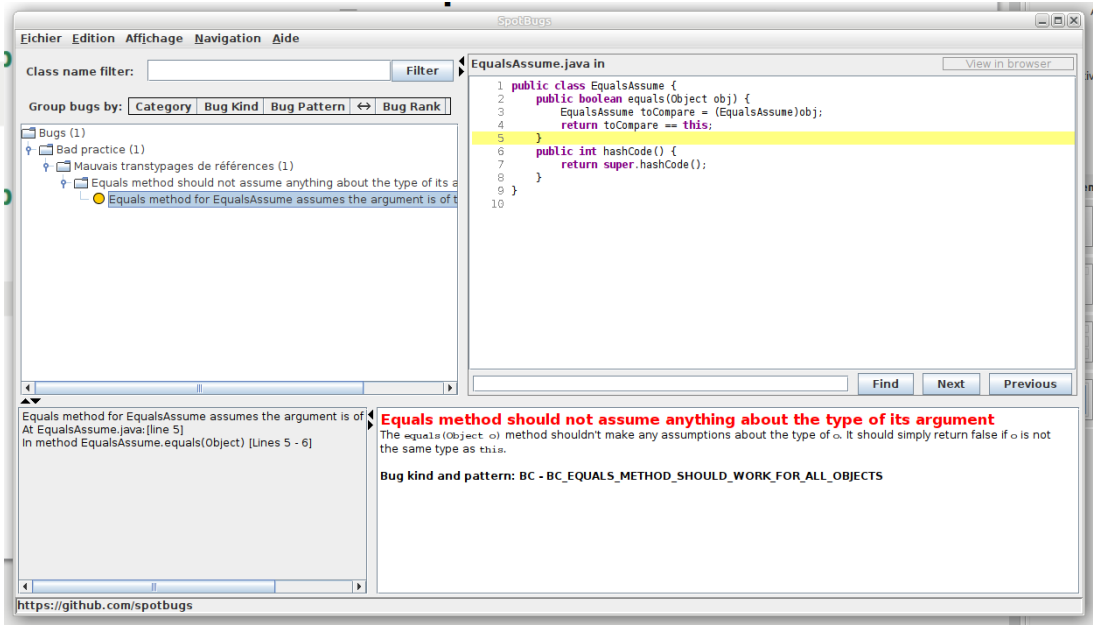


Exemple 1

```
1  public class Test2 {
2      //Unused field
3      private int abc;
4      private String ip = "127.0.0.1";
5      public void test() {
6          String[] field = {"a", "b", "c", "s", "e"};
7          //concatenates strings using + in a loop
8          String s = "";
9          for (int i = 0; i < field.length; ++i) {
10             s = s + field[i];
11         }
12         System.out.println(ip);
13     }
14 }
```

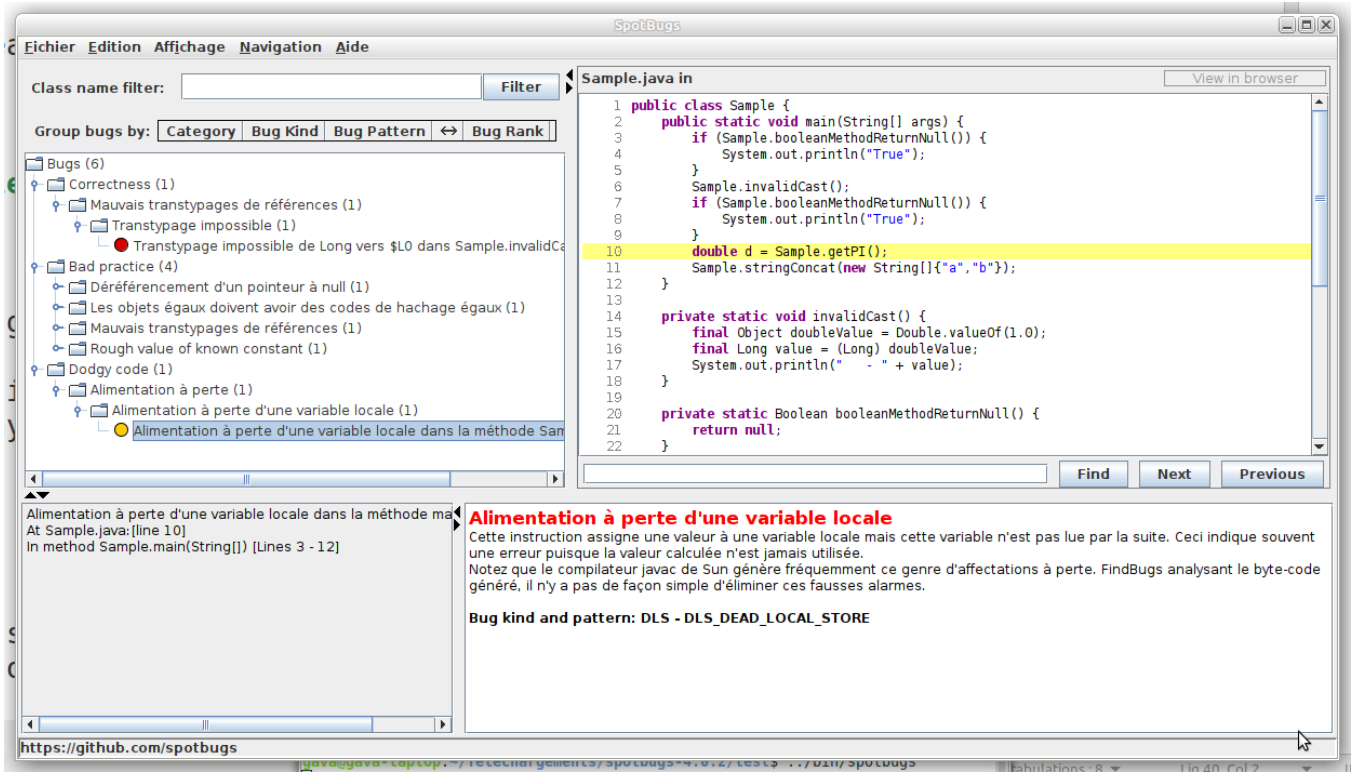
Exemple 2

```java
public class EqualsAssume {
    public boolean equals(Object obj) {
        EqualsAssume toCompare = (EqualsAssume)obj;
        return toCompare == this;
    }
    public int hashCode() {
        return super.hashCode();
    }
}
```



Exemple 3

- Sample.java (sur mon PC)
- https://mip-cloud.gitlab.io/post/2018-12-19-spotbugs/



Exercice

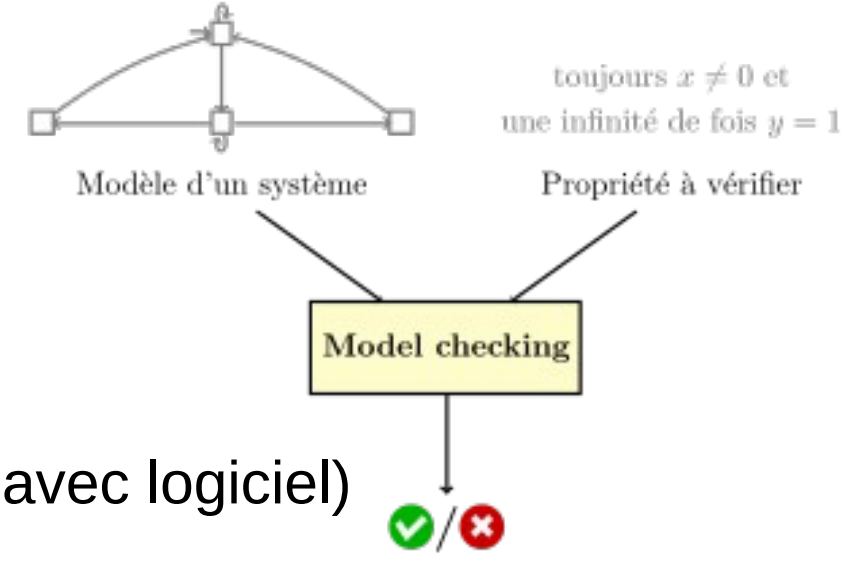
- Installez SpotBug
- Exécutez puis Analysez le fichier « Exo.java »
  - Une grosse erreur
  - Des warnings
- Corrigez !

Limitations et autres outils

- Modifions nos programmes de tris… et voyons un programme avec accès concurrents
  - ⇒ Tout va bien pour SpotBug… hum...
- https://developer.okta.com/blog/2019/12/20/five-tools-improve-java
- Model-Checking

Principe Model-Checking

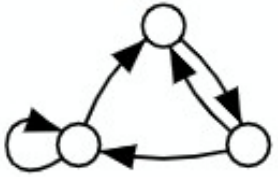
- Initié par Amir Pnueli (prix Turing en 1996) puis vraiment « inventé » par Edmund M. Clarke, E. Allen Emerson, Jean-Pierre Queille et Joseph Sifakis (Prix Turing 2007)
- Vérifier que le modèle d'un système satisfait une propriété (logique temporelle)
  - Système=des programmes, machines
  - Modèle=automates, Petri-Net, etc.
  - Logique ≡ LTL, CTL, etc.
- Modélisation
  - Faite « main » (ex,écriture d'un automate avec logiciel)
  - Depuis un programme (C, Java, etc.)



Résumons

Est-ce

système



modèle

satisfait

?

spécification

φ

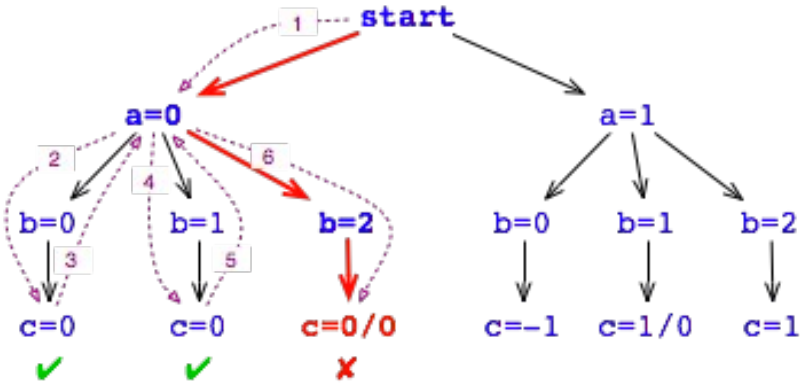
?

formule

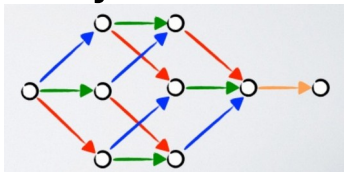
$\models$

algorithme de
Model Checking

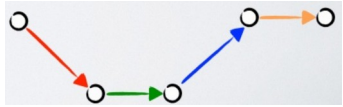
Fonctionnement

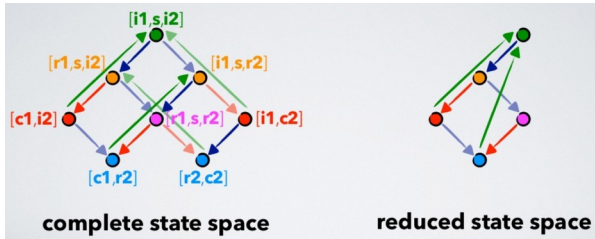


- (très grossièrement) Dans le cadre de la concurrence (threading), exécuter tout les chemins possibles (**state-space**) puis vérifier la propriété pour chacun des **chemins** (en cas d'erreur, on obtient un **trace** du problème)
- Inconvénient, taille de l'espace des états souvent exponentiel !
- Description formelle (mais accessible) : https://docplayer.fr/17937329-Logique-temporelle-et-model-checking-nathalie-sznajder-universite-pierre-et-marie-curie-lip6.html
- Des techniques de réduction existent :
  - Ordres partiels
  - Symétrie
  - Etc.



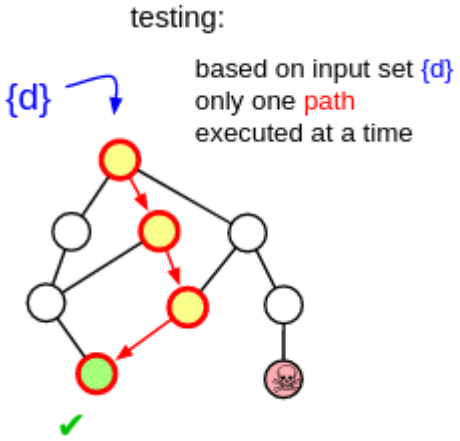
Partial Order

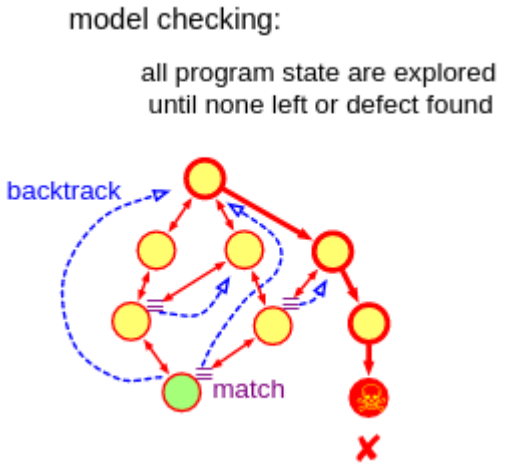




Tester <> Model Checker

- On test certains chemins ; On model-check **tout** les chemins… cela rend l'analyse bien plus compliquée





Pour commencer

- Vérifier (in)accessibilité à certains états dans le système (logique temporelle non nécessaire, formules de base de l'outil)
  - Accès concurrent à des données (data-race)
  - Inter-blocage (Deadlock)
- Système=Programme Java avec threads
- Outils JavaPathFinder (JPF) :
  - https://github.com/javapathfinder/
  - « runtime configured combination of different components » ⇒ Une VM qui exécutent tout les chemins (si possible) et qui vérifie si une formule logique est toujours satisfaite ou non

Installer JPF

- Existe aussi avec Ant
- Via Gradle (https://gradle.org/install)
  - Télécharger Gradle
    - Unzip (où vous voulez)
    - export PATH=$PATH: « chez vous »/gradle/gradle-6.3/bin
    - (ou dans le .bashrc/zsc)
- https://github.com/javapathfinder/jpf-core/tree/java-10-gradle
  - Download, unzip dans un dossier (projet)
  - Et commande « ./gradlew buildJars » (dans le dossier)

Accès concurrents

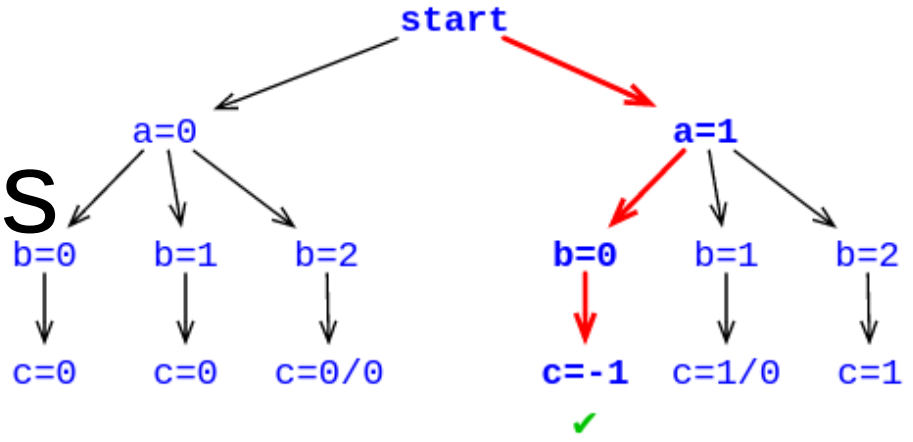
- javac -cp jpf-core/build/main/ Racer.java
- ./jpf-core/bin/jpf Racer.jpf
  - Le fichier jpf est :

    target = Racer

    listener=gov.nasa.jpf.listener.PreciseRaceDetector

    report.console.property_violation=error,trace
  - error #1: gov.nasa.jpf.listener.PreciseRaceDetector "race for field Racer@199.d main at Racer.main(Ra..."

Valeurs aléatoires




```
① Random random = new Random()
② int a = random.nextInt(2)
③ int b = random.nextInt(3)
④ int c = a/(b+a -2)
```

- Si on test, on peut rater une division par 0
- bin/jpf Rand ⇒ no errors detected
- bin/jpf +cg.enumerate_random=true Rand

Actuellement plante chez moi…

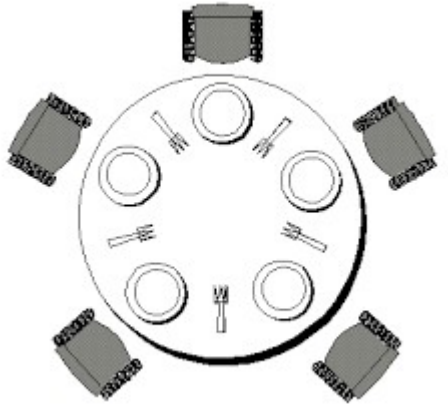
(java -jar RunJPF.jar ../Projet.jpf)

Différentes valeur

- Il est possible de faire de vérifier sur des ensembles de valeurs
- Verify.getInt(0,4);
- javac -cp . -cp jpf-core/build/main/ Test.java
- Plus de détails dans :
  - https://github.com/javapathfinder/jpf-core/wiki/ChoiceGenerators
  - Surtout gérer les cas comme double b=Verify.getDouble("heuristique") ;

    car il y a une « infinité » de doubles pour n'importe quelle borne
- On peut aussi rajouter des vérifications dans les programmes eux-même
  - Méthode du pauvre ⇒ assert
  - https://github.com/javapathfinder/jpf-core/wiki/Writing-JPF-tests

Exo, le dîner des philosophes



- N philosophes (threads) autour d'une table veulent, une unique fois, penser puis manger. Pour manger ils ont besoin de 2 fourchettes (gauche et droite). Mais il n'y a que N fourchettes. Attention :
  - Si un philosophe ne mange pas, il meurt.
  - La fourchette de gauche d'un philosophe peut être la fourchette de droite d'un autre philosophe ; On ne veut donc pas de deadlock
- Dans le vrai problème, les philosophes dînent sans arrêt (while (true)), cela rendrait le model-checking encore plus difficile...
- Regardons le fichier « Philo.java »
  - Exécutez pour plusieurs valeurs de N ; Corriger en mettant les « synchronized »
  - Vérifiez avec JPF
  - Trouvez une correction !
  - Vérifiez votre programme avec JPF pour 2,3,4,5,10, philosophes… (pour les courageux, avec CheckStyle et spotbugs aussi !)