CSC4102 : Qualité du code JAVA et introduction aux idiomes JAVA

Denis Conan

Janvier 2025

1 Motivations et objectifs

1.1 Qui demande à améliorer la qualité du code ?

« *The Google FindBugs Fixit* » [Ayewah and Pugh, 2010]

In May 2009, Google conducted a company wide FindBugs *"fixit"*[1].

Hundreds of engineers reviewed thousands of FindBugs *warnings.*

- *Static analysis tools [such as* FindBugs*] scan software looking for issues that might cause defective behavior.*
- *Most interesting software quality problems were eventually found and fixed without* FindBugs*, but* FindBugs *could have found these problems early, when they are cheap to remediate.*
  - *These observations [are attributed] to the success of Google's testing and monitoring practices.*
- *Engineers viewed* FindBugs *warnings as worth fixing.*
  - *Over 77% of the reviews identified the warnings as real defects and recommended fixing the underlying problems.*

1. C'est ce que l'on appelle une revue de code, ici avec FindBugs, qui a été repris dans SpotBugs [Ayewah et al., 2008]

1.2 Pourquoi les idiomes ?

G.L. Steele, ACM Grace Murray Hopper Award, 1988

Foreword of « *Effective JAVA* » [Bloch, 2008]

- *If you ever studied a second language yourself and then tried to use it outside the classroom, you know that there are three things you must master :*
  1. *How the language is structured (grammar)*
  2. *How to manage things you want to talk about (vocabulary)*
  3. *The customary and effective ways to say everyday things (usage)*
- *It is much the same with a programming language.*
  1. *You need to understand the core language [UML,JAVA = object-orientation]*
  2. *You need to know the vocabulary [e.g. JAVA standard library for collections]*
  3. *You need to be familiar with the customary and effective ways to structure your code*

  — C'est ce qui s'appelle les patrons d'implémentation ou idiomes

1.3 Quels sont les objectifs de la séance ?

- Étudier la qualité du code déja produit
  - On paie de la dette technique
  - Avec les outils SpotBugs et CheckStyle
    - On atténue les efforts nécessaires à l'aide d'outil d'analyse statique du code
- Étudier l'application d'idiomes JAVA
  - On améliore la qualité du code en améliorant nos connaissances sur le langage
  - À titre d'exemple, sur les méthodes communes à toutes les classes : `equals`, `hashCode`, `toString`, etc.
  - Autres exemples avec les expressions *lambda*, les *Streams*, et `Optional`
- Continuer le développement de l'application de l'étude de cas
  - On annonce dès maintenant les nouveaux cas d'utilisation pour les dernières séances

1.4 Après la séance — À quoi ça sert ? où ? quand ?

- Périodiquement, utiliser les outils d'analyse statique de code dans ECLIPSE
- Maintenir la qualité (passage des tests) avec l'intégration continue

[optionn.] Pour ceux qui veulent aller plus loin, possibilité de générer un site Web MAVEN, qui présente les rapports de qualité

[optionn.] Pour les « professionnels », possibilité de faire échouer la construction du logiciel en cas de problèmes détectés avec les outils CHECKSTYLE et SPOTBUGS

— Dans ce module, nous n'irons pas jusque là

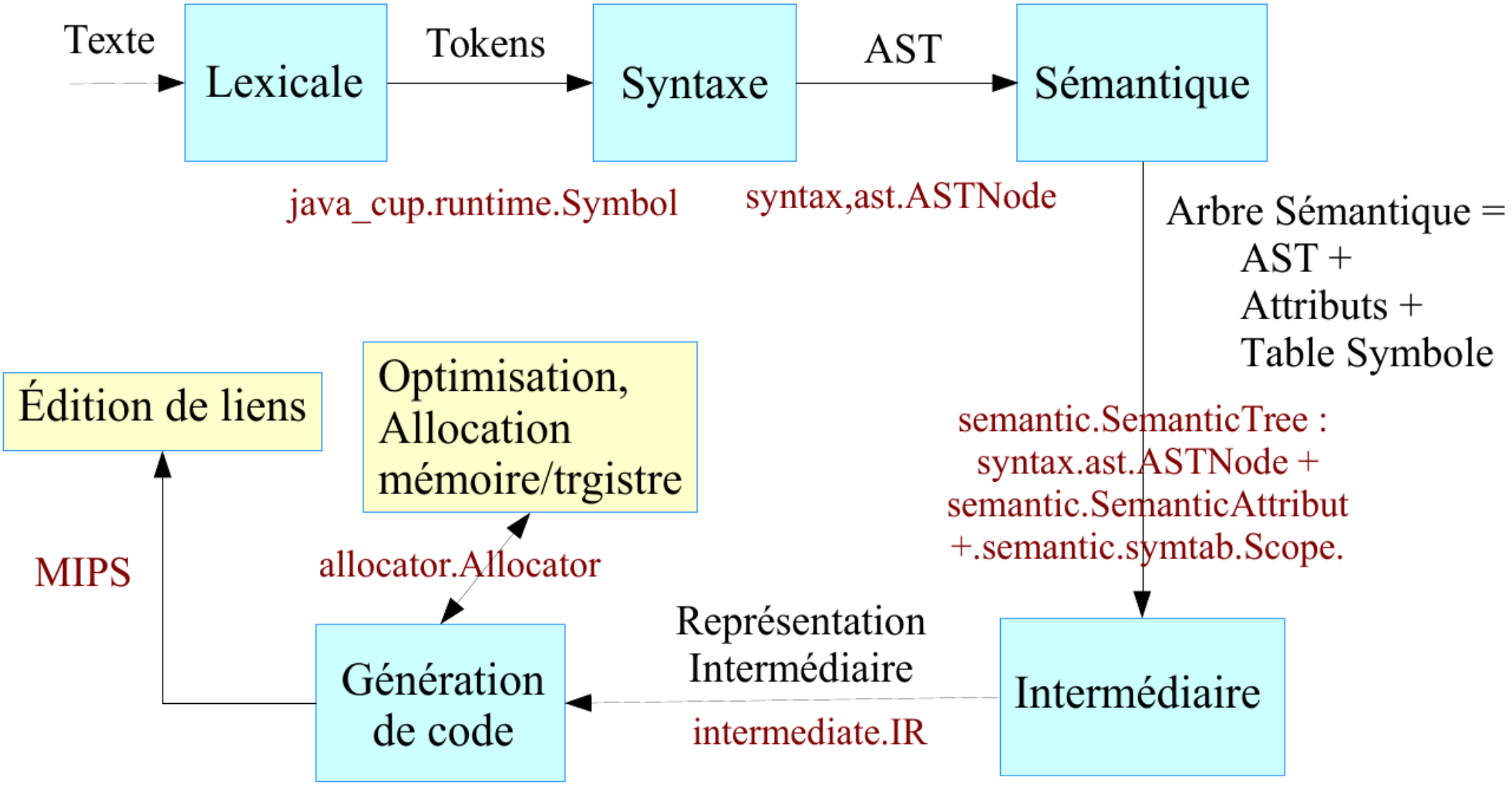
- Appliquer des idiomes JAVA, y compris utilisant les *lambda expressions*
  - Dans ce module, nous n'étudions pas la programmation orientée fonction à proprement parler

— Pour découvrir ce sujet, p.ex. :
P.-Y. Saumont, *Functional Programming in Java—How functional techniques improve your Java programs*, [Saumont, 2017]

Sommaire

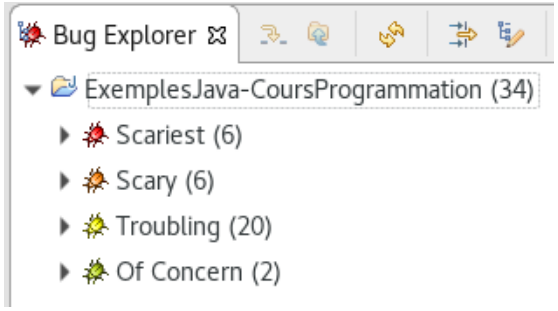
2 Outils d'analyse statique de code = Pour aider !

2.1 Analyse statique de code



1. Image extraite du cours CSC4251–52 (ex CSC4536)

2.2 SpotBugs



- Catégories (https://spotbugs.github.io/)
  - *Correctness bug* : *Probable bug, an apparent coding mistake resulting in code that was probably not what the developer intended*
    - *We strive for a low false positive rate*
  - *Bad Practice* : *Violations of recommended and essential coding practice*
    - *Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize*
    - *We strive to make this analysis accurate, although some groups may not care about some of the bad practices*
  - *Dodgy (douteux, risqué) : Code that is confusing, anomalous, or written in a way that leads itself to errors*
    - *Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null*
    - *More false positives accepted*

1. SpotBugs *doesn't try to identify all defects in a particular category*

[Ayewah et al., 2008]

2.2.1 Exemples, b/e

```
1 pi = null;
2 for(it= vPersonnes.iterator();it.hasNext();pi=it.next()) {
3   if(pi.equals(new Personne("Dupont"+5, "Jules","19102271271"+5))) {
4     it.remove();
5   }
6 }
```

- « Déréférencement d'un pointeur null dans la méthode main(String[]) »
  - « Un pointeur à null est déréférencé ici. Ceci va mener à une NullPointerException quand le code sera exécuté. »
    - Le problème est à la ligne 2 : l'affectation « pi = it.next() » n'a lieu qu'en fin de boucle ;
      donc, lors de l'entrée dans la boucle for, pi vaut sa valeur avant la boucle, c'est-à-dire ici null

2.3 CheckStyle

- Comme son nom le suggère, adhésion à un standard de codage
  - Deux fournis par défaut dans Eclipse
    - *Sun Checks* et *Google checks*
- Pour nous, l'objectif « numéro 1 » est la documentation Javadoc
  - Configuration particulière de CheckStyle fournie en TP
    - Feuille de style construite à partir de *Sun Checks*
      - Nous autorisons les tabulations, les caractères « _ » dans les noms de paquetages, les lignes à plus de 80 colonnes, etc.
      - Nous ne développons pas une bibliothèque
    - Uniquement pour quelques classes
  - Génération de la documentation dans Eclipse (menu *Project/Generate Javadoc*)
    - Possibilité pour ceux qui le souhaitent de générer avec Maven (`mvn site`)

1. https://checkstyle.sourceforge.io/

2.3.1 Documentation JavaDoc

Classe `eu.telecomsudparis.csc4102.util.Datutil`

```
1           /**
2            * compare la date avec aujourd'hui : vrai si aujourd'hui.
3            *
4            * @param date la date a comparer avec le jour courant.
5            * @return vrai si aujourd'hui
6            * @throws IllegalArgumentException si {@code date} est {@code null}
7            */
8           public static boolean dateEstAujourdhui(final LocalDate date) {
9                   if (date == null) {
10                          throw new IllegalArgumentException("date == null");
11                  }
12                  return date.equals(aujourdhui());
13          }
```

http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[Javadoc, JSE8]

Notez que les commentaires Javadoc commencent par la chaîne de caractères « /** »

2.3.2 Exemples, a/d

- Vérification de la présence et correction « du Javadoc » avec CheckStyle

```
1 /**
2  * Bla bla.
3  * @param e explication sur e.
4  */
5 public void ajouter(final int a, final int b) { };
```

- « Balise javadoc @param inutilisé pour ‘e’. »
- « Balise javadoc @param manquante pour ‘a’. »
- « Balise javadoc @param manquante pour ‘b’. »

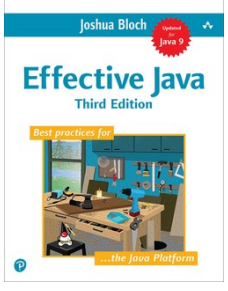
3 Définition idiome/patron d'implémentation

- « *By idioms, we mean some (often unusual) combination of abstractions specific to some programming language, which are used for implementing some other abstraction that is not part of this language.* » [Czarnecki and Eisenecker, 2000]

- Notre position dans ce module : les idiomes permettent d'approfondir des concepts de base importants

- Livre de référence pour JAVA : « *Effective Java, 3rd Edition* » [Bloch, 2018]

  - Dans CSC4102, à titre d'exemple, parcours de quelques idiomes
    - Méthodes de la classe `Object` communes à toutes les classes
    - Lambda expressions, *Streams*, et `Optional`
    - Plus dans le document « pour aller plus loin » de cette séance



4 Idiomes JAVA, premiers exemples, méthodes de la classe `Object` communes à tous les objets

4.1 Méthode `equals`

- Repris de l'*Item 8* de [Bloch, 2018]
- Contrat[2] [Class Object, JSE8]
  - *Reflexive : for any non-null reference value* `x`*,* `x.equals(x)` *should return* `true`
  - *Symmetric : for any non-null reference values* `x` *and* `y`*,* `x.equals(y)` *should return* `true` *if and only if* `y.equals(x)` *returns* `true`
  - *Transitive : for any non-null reference values* `x`*,* `y`*, and* `z`*, if* `x.equals(y)` *returns true and* `y.equals(z)` *returns* `true`*, then* `x.equals(z)` *should return* `true`
  - *Consistent : for any non-null reference values* `x` *and* `y`*, multiple invocations of* `x.equals(y)` *consistently return* `true` *or consistently return* `false`*, provided no information used in equals comparisons on the objects is modified*
  - *For any non-null reference value* `x`*,* `x.equals(null)` *should return* `false`
- Cf. « pour aller plus loin » avec des exemples de non-respect des propriétés
  - Conclusion : Utilisez l'assistant Eclipse pour générer les méthodes `equals` + Choisir les attributs repérant de manière unique et constante un objet de la classe

2. Prend en argument un `Object` : utiliser `@Override` pour éviter une erreur

4.2 Méthode hashCode

- Repris de l'*Item 9* de [Bloch, 2018]
- Redéfinir equals impose de redéfinir aussi hashCode :
  $\forall o_1, o_2 \in Classe : o_1.\texttt{equals}(o_2) \implies o_1.\texttt{hashCode}() = o_2.\texttt{hashCode}()$

Classe seance7.methodescommunesatouslesobjets.CleErronee

```
1 public final class CleErronee {
2   private String nom; private String prenom;
3   public CleErronee(final String n, final String p) {nom=n;prenom=p;}
4   @Override public boolean equals(final Object obj){//...avec nom,prenom
5 }
```

Classe seance7.methodescommunesatouslesobjets.ExempleCleSansHashCode

```
1   Client c = new Client("nom", "prenom", "adresse");
2   CleErronee c1=new CleErronee("nom","prenom"),c2=new CleErronee("nom","
        prenom");
3   Map<CleErronee,Client> d=new HashMap<>();d.put(c1, c);
4   System.out.print(c1.hashCode()+"!="+c2.hashCode()+",␣"+d.get(c2));
```

- Résultat de l'exécution : `366712642!=1829164700, null`
  - *As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer.)* [Class Object, JSE8]

4.3 Méthode `toString`

- Repris de l'*Item 10* de [Bloch, 2018]
- Toujours redéfinir `toString`
  - Par défaut, nom de la classe + `@` + représentation hexadécimale non signée de la valeur donnée par `hashCode()`

Classe seance7.methodescommunesatouslesobjets.EnfantAvecToString

```
1 public class EnfantAvecToString extends ParentAvecToString {
2   private int b;
3   public EnfantAvecToString(final int a,final int b) {super(a);this.b = b;}
4   @Override public String toString() {
5     return "EnfantAvecToString␣[b="+b+",␣toString()="+super.toString()+"]";}}
```

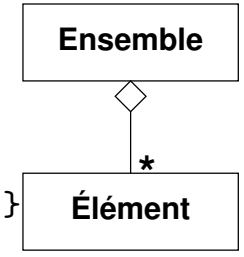
Classe seance7.methodescommunesatouslesobjets.ExemplesToString

```
1   EnfantSansToString e = new EnfantSansToString(1,2);
2   System.out.println(e + ",␣" + Long.toHexString(e.hashCode()));
3   EnfantAvecToString e2 = new EnfantAvecToString(1,2);
4   System.out.println(e2);
```

- Résultat de l'exécution :

```
seance7.methodescommunesatouslesobjets.EnfantSansToString@15db9742, 15db9742
EnfantAvecToString [b=2, toString()=ParentAvecToString [a=1]]
```

4.3.1 Assoc. bidirectionnelles et `toString`



Classe seance7.methodescommunesatouslesobjets.associationbidirectionnelle.Ensemble

```
1   private List<Element> elems;
2   public Ensemble(final int v) { elems = new ArrayList<>(); }
3   public void ajouterUnA(final int v) { elems.add(new Element(this, v)); }
4   @Override // redefinition : contenu toString() de elems
5   public String toString() {return "Ensemble [elems="+elems+"]";}}
```

Classe seance7.methodescommunesatouslesobjets.associationbidirectionnelle.Element

```
1   private int v;
2   private Ensemble ens;
3   public Element(final Ensemble ens, final int v) {this.v=v;this.ens=ens;}
4   @Override // redefinition : contenu de v et toString() de ens
5   public String toString() {return "Element [v="+v+", ens="+ens+"]";}}
```

Classe seance7.methodescommunesatouslesobjets.associationbidirectionnelle.ExempleBoucleInfinie

```
1   Ensemble b = new Ensemble(1); b.ajouterUnA(2); System.out.println(b);
```

- Résultat de l'exécution :

  `Exception in thread "main"` `java.lang.StackOverflowError`

- Il faut éviter la formation du cycle d'appels
  - `Ensemble::toString` doit utiliser des *getters* de `Element` au lieu d'utiliser `Element::toString`

4.3.2 Énumérations avec toString *

Classe seance7.methodescommunesatouslesobjets.EnumerationSansRedefinitionDeToString

```
1 public enum EnumerationSansRedefinitionDeToString {PREMIER, DEUXIEME;}
```

Classe seance7.methodescommunesatouslesobjets.EnumerationAvecRedefinitionDeToString

```
1 public enum EnumerationAvecRedefinitionDeToString {
2  PREMIER("Mon premier"),DEUXIEME("Mon second");
3    private String libelle;
4    // notez que le constructeur est prive
5    private EnumerationAvecRedefinitionDeToString(final String libelle) {
6      this.libelle = libelle; }
7    @Override // redefinition : le libelle, plutot que le nom de l'objet
8    public String toString() { return libelle; } }
```

Classe seance7.methodescommunesatouslesobjets.ExemplesToStringEnumeration

```
1    System.out.println(EnumerationSansRedefinitionDeToString.DEUXIEME);
2    System.out.println(EnumerationAvecRedefinitionDeToString.DEUXIEME);
```

- Résultat de l'exécution :

```
DEUXIÈME
Mon second
```

2. Le constructeur de EnumerationAvecRedefinitionDeToString est private car il n'existe aucune raison de le mettre public puisqu'aucun nouvel objet n'est instanciable depuis l'extérieur

4.4 Les autres méthodes de la classe `Object`

[Class Object, JSE8]

- `getClass` : *Returns the runtime class of this object*
- Méthodes pour la concurrence (hors programme CSC4102) :
  - S'utilise avec la directive `Synchronized`
  - `notify` : *wakes up a single thread that is waiting on this object's monitor*
    - *If any threads are waiting on this object, one of them is chosen to be awakened*
    - *The choice is arbitrary and occurs at the discretion of the implementation*
  - `notifyAll` : *wakes up all threads that are waiting on this object's monitor*
  - `wait` : *causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed*

5 Retours sur les *Lambda expressions* JAVA

. Les éléments de cette section sont extraits des références [Naftalin, M., 2012], [Naftalin, 2015], et [Bloch, 2018]

5.1 Préliminaire : interface fonctionnelle

- Une interface fonctionnelle
  - Est une interface
  - Possède une seule méthode abstraite
  - Est annotée `@FunctionalInterface`
- Un exemple :

```
@FunctionalInterface public interface Consumer<T> { void accept(T t); }
```

- Objectif : simplifier le code avec du sucre syntaxique
  - De

```
pointList.forEach(
new Consumer<Point>(){public void accept(Point p){p.translate(1,1);}}
);
```

  - À

```
pointList.forEach(p -> p.translate(1, 1));
```

5.2 Définition et syntaxe

- *Lambda expression* = conceptuellement une fonction
  - Arguments en entrée et valeur de retour
- Syntaxe
  - **Soit :** `(parameters) -> expression`
  - **Soit :** `(parameters) -> { statements; }`
- C'est un objet = une instance d'une interface fonctionnelle

```
Runnable r = () -> {}; // crée une lambda expression
                       // et affecte une réf. vers cette lambda expression
Object o = r; // transtypage vers le haut, comme pour une référence/un objet
```

5.3 Exemples de *lambda expressions*

```
1.  (int x, int y) -> x + y                    // retourne la somme des deux arguments
2.  (x, y) -> x - y                            // retourne la différence
3.  () -> 42                                   // pas d'argument, retourne 42
4.  (String s) -> System.out.println(s)        // affiche l'argument et ne retourne rien
5.  x -> 2 * x                                 // un argument (sans parenthèse)
6.  c -> { int s = c.size(); c.clear(); return s; }   // type de l'argument possédant les méthodes
                                                      // size() et clear(), p.ex. une collection
```

- Types des arguments explicites (1, 4) ou inférés (2, 5, 6)
  - Pas de mélange entre « explicite » et « inféré »
- Le corps peut être un bloc (6) ou une expression (1–5)
  - **Bloc retournant une valeur (dit *value-compatible*) ou rien (dit *void-compatible*)**
  - Idem pour l'expression : une valeur (1, 2, 3, 5) ou rien (4)
- Un seul argument $\Longrightarrow$ possible d'omettre les parenthèses autour de l'argument

5.4 Référence de méthode*

- Cas particulier d'une *lambda expression* avec un seul argument

  + l'expression est un unique appel à une méthode avec un seul argument

  - Par exemple : `str -> Integer.parseInt(str)`
  - Simplification de l'écriture avec la forme « référence de méthode »

  — Pour le même exemple : `Integer::parseInt`

- Plus généralement, une *lambda expression* peut être représentée par une méthode concrète d'une classe

  $\Longrightarrow$ Une référence de méthode est un raccourci d'écriture d'une *lambda expression* avec un argument et l'expression formée de l'appel unique de la méthode référencée

- Syntaxe (plus d'informations dans le document « pour aller plus loin »)

  - **`ReferenceType::identifier`** pour méthode de classe,
    p.ex. « `Integer::parseInt` »
  - **`ObjectReference::identifier`** pour méthode d'instance :
    p.ex. « `System.out::println` »
  - **`ReferenceType::new`** pour constructeur, p.ex. « `ArrayList::new` »

5.5 Interfaces fonctionnelles standard*

- `java.util.function` [PackageJavaUtilFunction, JSE8]
  - Plus de 40 fonctions
    - Dont **5 de base : `Operator`, `Predicate`, `Function`, `Supplier`, `Consumer`**
    - Avec des variantes pour les types primitifs

	Interface	**Signature de la fonction**	**Exemple**
1	`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
2	`BinaryOperator<T>`	`T apply(T t1, T, t2)`	`BigInteger::add`
3	`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
4	`Function<T, R>`	`R apply(T t)`	`Arrays::asList`
5	`BiFunction<T, U, R>`	`R apply(T t, U u)`	`(x, y) -> x - y`
6	`Supplier<T>`	`T get()`	`Instant::now`
7	`Consumer<T>`	`void accept(T t)`	`System.out::println`
8	`ToIntFunction<T>`	`int applyAsInt(T t)`	`Integer::parseInt`
9	`IntFunction<R>`	`R apply(int i)`	`int[]::new`

- Pour chaque ligne du tableau, on peut écrire « colonne Interface » `ref =` « colonne Exemple »
  - P.ex. `Function<ArrayList, List> obj = Arrays::asList;`

5.5.1 Exemples d'utilisation (de base)

Classe seance7/lambdaexpressions/InterfaceFonctionnelleStandard

```
1  // interfaces fonctionnelles de base
2  UnaryOperator<String> f1 = String::toLowerCase;
3  System.out.println(f1.apply("ABC")); //
4  BinaryOperator<BigInteger> f2 = BigInteger::add;
5  BigInteger a = BigInteger.valueOf(2), b = BigInteger.valueOf(3);
6  System.out.println(f2.apply(a, b)); //
7  Predicate<Collection<String>> f3 = Collection<String>::isEmpty;
8  System.out.println(f3.test(new ArrayList<>())); //
9  Function<String[], List<String>> f4 = Arrays::asList;
10 System.out.println(f4.apply(new String[] {"a", "b"})); //
11 Supplier<Instant> f5 = Instant::now;
12 System.out.println(f5.get()); //
13 Consumer<String> f6 = System.out::println;
14 // variante pour les types primitifs
15 ToIntFunction<String> f7 = Integer::parseInt;
16 IntFunction<int[]> f8 = int[]::new;
17 f6.accept("abc"); System.out.println(f7.applyAsInt("11")); //
18 int[] t = f8.apply(4); System.out.println(t.length); //
```

Affichage :

« abc », puis « 5 », « true », « [a, b] », « 2019-03-13T21:06:26.529Z », « abc », « 11 », et enfin « 4 »

5.6 Exemple d'utilisation dans JUnit

- Le second argument de la méthode de classe `Assert.assertThrows` est une expression lambda
  - Expression de type `ThrowingRunnable`, qui est une interface fonctionnelle
  - L'expression n'a pas d'argument en entrée : « `()` »
  - L'expression exécute une méthode qui ne retourne aucune valeur : « `void` »
  - Levée d'une exception `AssertionError` si l'expression lambda ne lève pas d'exception

Classe `seance6.mediathequesimplifie.testdevalidation.TestAjouterLocalisation`

```
1       @AfterEach
2       public void tearDown() {
3           mediatheque = null;
4       }
5   }
```

- `OperationImpossible assertThrows(Class<OperationImpossible>, Executable)`
  - Avec `public interface Executable { void execute() throws Throwable; }`

6 Idiomes JAVA, pipelines des *Streams*

- Philosophie UNIX : « *At its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.* » [Kernighan and Pike, 1984]
  - Les *lambda expressions* = traitements à granularité fine et composables
- Pipeline = « début » + « traitement intermédiaire » + « terminaison »
  - Description détaillée dans le document « pour aller plus loin »
- *Stream* = Séquence de valeurs, ordonnées ou non, et non stockées
- Pipeline de *streams* + évaluation tardive
  - C'est l'opération terminale qui « tire » les valeurs
    - Aucune valeur n'est calculée tant qu'elle n'est pas demandée

```
IntStream.iterate(1, i -> i * 2) // génération stream de taille infinie
         .limit(10) // limitation du nombre de générations
         .forEach(System.out::println) // opération qui « tire » les valeurs
```

. Les éléments de cette section sont extraits de [Naftalin, M., 2012], [Naftalin, 2015], et [Bloch, 2018]

UNIX [illegible] suit

6.1 Exemples sur les pipelines des *Streams* a/i

- Début : génération d'une séquence infinie avec `IntStream.iterate`
  Intermédiaire : troncature avec `limit` + insertion déverminage avec `peek`
  Terminaison : test de présence avec `anyMatch`
- Démonstration de l'évaluation tardive

Classe `seance7/lambdaexpressions/ExampleNaftalin2015`

```
1 boolean trouve = IntStream.iterate(1, i -> i * 2)
2                           .limit(10)
3                           .peek(System.out::println)
4                           .anyMatch(v -> v == 16);
```

Affichage :

```
1
2
4
8
16
```

6.2 Exemples sur les pipelines des *Streams* b/i

- Début : `stream de Collection`
  Intermédiaire : filtrage avec `filter`
  Terminaison : recherche d'un élément avec `findFirst`

- Recherche dans une collection avec filtrage
  + obtention de la première valeur trouvée

Classe `etudesdecas/mediathequeaveclambdasoptionaletstreams/Mediatheque`

```
1 /**
2  * chercherGenre cherche un Genre dans la liste des genres.
3  *
4  * @param nomGenre du Genre a chercher
5  * @return le genre correspondant au nom dans la collection
6  */
7 private Optional<Genre> chercherGenre(final String nomGenre) {
8   return lesGenres.stream().filter(g -> g.getNom().equals(nomGenre)).
        findFirst();
9 }
```

La classe `Optional` est étudiée dans la section qui suit celle-ci.

6.3 Exemples sur les pipelines des *Streams* d/i

- Début : génération d'une séquence infinie avec `IntStream.iterate`
  Intermédiaire : troncature avec `limit` + transformation avec `mapToObj`
  Terminaison : réduction avec `collect` en une liste avec `toList de Collectors`

- Collecte dans une liste

Classe `seance7/lambdaexpressions/ExampleNaftalin2015`

```
1 List<String> l = IntStream.iterate(1, i -> i * 2)
2                           .limit(10)
3                           .mapToObj(String::valueOf)
4                           .toList(); // idem .collect(Collectors.toList())
5 System.out.println(l);
```

Affichage :

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

- Cet exemple démontre implicitement une construction d'objet
  - On pourrait écrire `mapToObj(Integer::new)` pour récupérer une `List<Integer>`

Depuis JAVA 16, `.collect(Collectors.toList())` est écrit plus simplement `.toList()`

6.4 Exemples sur les pipelines des *Streams* f/i

- Début : génération d'une séquence infinie avec `IntStream.iterate`
  Intermédiaire : troncature avec `limit`
  Terminaison : réduction avec `reduce`
  à partir d'une valeur de départ dite « identité » et avec une fonction d'accumulation telle que accumulateur.apply(identité, t).equals(t)

Classe `seance7/streams/ExemplesStreamsReduce`

```
1 IntStream stream = IntStream.iterate(1, i -> i * 2).limit(10);
2 int result = stream.reduce(0, Integer::sum); // identite, accumulateur
3 //equivalent a :
4 stream = IntStream.iterate(1, i -> i * 2).limit(10);
5 int resultBis = 0; // 0 : valeur dite identite
6 for (Integer element : stream.toArray()) {
7     resultBis = Integer.sum(resultBis, element);
8 }
9 System.out.println(result + "␣=␣" + resultBis);
```

Affichage :

```
1023 = 1023
```

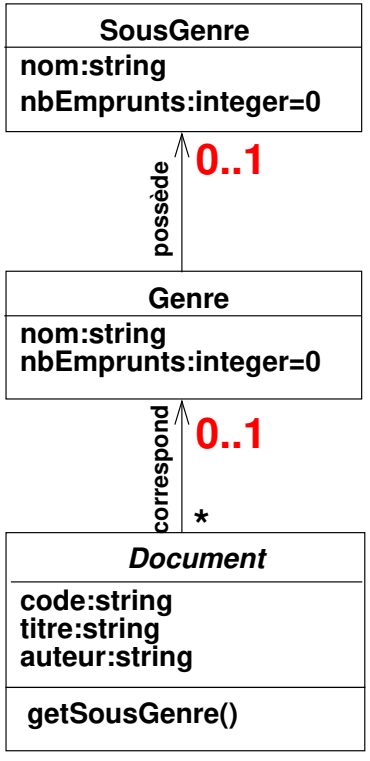
7 Idiomes JAVA, gestion des références `null` avec `Optional`

7.1 Problème des références `null`

- Situations dans lesquelles « pointeur `null` » signifie « absence de valeur »
  - P.ex. valeur de retour pouvant être `null`

Classe `seance7/lambdaoptional/DocumentSansOptional`

```
1  public String getNomSousGenre() {
2    String result = "inconnu";
3    if (genre != null) { // attribut pouvant etre null
4      SousGenre sg = genre.getSousGenre();
5      if (sg != null) { // variable pouvant etre null
6        String n = sg.getNom();
7        if (n != null) { // inutile si l'invariant le verifie
8          result = n;
9        }
10     }
11   }
12   return result;
13 }
```



7.2 Solution non satisfaisante : documentation

- Besoin d'avertir le programmeur de la méthode appelante
  - Premier essai : utiliser la documentation Javadoc

Classe `seance7/lambdaoptional/GenreSansOptional`

```
1 /**
2  * obtient le sous-genre.
3  *
4  * @return le sous-genre, qui peut etre {@code null}.
5  */
6 public SousGenre getSousGenre() {
7   return sousGenre;
8 }
```

7.3 Solution/idiome JAVA : classe `Optional`

- La solution/l'idiome JAVA :
  - Similairement à d'autres langages comme Groovy ou Scala,
    JAVA propose depuis la version 8 un idiome pour rendre explicite la possibilité qu'une référence puisse être `null`
    et ainsi alerter le programmeur pour qu'il évite le déréférencement dans ce cas
    - P.ex. : `Optional<SousGenre> getSousGenre()`

- Pour votre culture générale, conférence de C.A.R. Hoare,
  qui en 1965 a introduit le concept de « référence `null` »,
  ce qu'il considère maintenant comme une erreur :
  - Conférence en 2009 : « *Null References: The Billion Dollar Mistake* »

7.3.1 Classe `Optional`

- Classe conteneur et paramétrée par le type du contenu : `Optional<T>`
- Pas de constructeur, mais création par méthodes de classes dites « fabriques » (*factories*)
  - `Optional<Genre> g = Optional.of(genre)` : valeur `null` non autorisée (exception `NullPointerException`)
  - `Optional<Genre> g = Optional.ofNullable(genre)` : valeur `null` autorisée
  - `Optional<Genre> g = Optional.empty()` : contient `null`
- Des méthodes pour récupérer la valeur ou tester si `null`
  - `g.isPresent()` : retourne `false` si contenu `null`
  - `g.get()` : lève l'exception `NoSuchElementException` si contenu `null`
  - `g.orElse("inconnu")` : retourne contenu non `null`, sinon `"inconnu"`
- Des méthodes pour manipuler la valeur si non `null`
  - `sg.map(SousGenre::getNom)` : applique la *lambda expression* si non `null` et retourne la valeur dans un `Optional` (p.ex. ici, un `Optional<String>`)
  - `sg.flatMap(SousGenre::getNom)` : idem `map`, mais sans mettre le résultat dans un `Optional` (p.ex. ici retourne un `String`)

7.3.2 Où mettre `Optional` ?

- « *`Optional` is primarily intended for use as a method return type where there is a clear need to represent "no result," and where using `null` is likely to cause errors.* » [Class Optional, JSE8]
- « *The intention of the `Optional` class is not to replace every single `null` reference* » [Urma, 2014]
- Dans le module CSC4102, `Optional` en guise de valeur de retour lorsque `null` est fonctionnellement possible (pas un cas d'erreur)

Classe `seance7/lambdaoptional/GenreAvecOptional`

```
1  /**
2   * obtient le sous-genre.
3   *
4   * Le diagramme de classes specifie qu'un sous-genre est optionnel.
5   *
6   * @return le sous-genre, qui peut etre {@code null}.
7   */
8  public Optional<SousGenre> getSousGenre() {
9    return Optional.ofNullable(sousGenre);
10 }
```

7.3.3 Manipulation d'un objet `Optional` (1/2)

- Même exemple qu'au début de la section avec imbrication de `if`
  - `isPresent` : retourne `true` si un objet est présent
  - `get` : retourne la valeur présente, sinon lève l'exception `NoSuchElementException`

Classe `seance7/lambdaoptional/DocumentAvecOptional1`

```
1  public String getNomSousGenre() {
2    String result = "inconnu";
3    if (getGenre().isPresent()) {
4      Optional<SousGenre> sg = genre.getSousGenre();
5      if (sg.isPresent()) {
6        String n = sg.get().getNom();
7        if (n != null) { // inutile si l'invariant le verifie
8          result = n;
9        }
10     }
11   }
12   return result;
13 }
```

7.3.4 Manipulation d'un objet de `Optional` (2/2)

- Meilleur code

  Classe `seance7/lambdaoptional/DocumentAvecOptional2`

```
1 public String getNomSousGenre() {
2   return getGenre()
3           .flatMap(GenreAvecOptional::getSousGenre)
4           .map(SousGenre::getNom)
5           .orElse("inconnu");
6 }
```

  - Utilisation de `flatMap`, `map`, et `orElse`
    - `GenreAvecOptional::getSousGenre` retourne un `Optional<SousGenre>`
    - D'où, `getGenre().map(GenreAvecOptional::getSousGenre)` retournerait un objet de type `Optional<Optional<SousGenre>>`
    - Donc, `flatMap` pour obtenir un objet de type `Optional<SousGenre>` au lieu d'un objet de type `Optional<Optional<SousGenre>>`

      `flatMap` : *This method is similar to `map(Function)`, but the mapping function is one whose result is already an `Optional`, and if invoked, `flatMap` does not wrap it within an additional `Optional`*

      [Class Optional, JSE8]
    - orElse : retourne la valeur si présente ($\equiv$ get), sinon une valeur par défaut

8 Mise en pratique en TP (2h) + HP (3h)

- Qualité du code avec SpotBugs puis CheckStyle
- Idiomes : programmation des méthodes communes à tous les objets
- Utilisation des *Streams* et de `Optional` dans certaines parties du code
- Continuation du développement de l'application de l'étude de cas
- Rendu de la séance en HP : diagrammes PlantUML + `readme.md` + code

- Compléments « Pour aller plus loin » sur les idiomes JAVA

Références I

Ayewah, N. and Pugh, W. (2010).
The Google FindBugs Fixit.
In *Proc. of the 19th International Symposium on Software Testing and Analysis*, pages 241–252, Trento, Italy.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, D., and Penix, J. (2008).
Using Static Analysis to Find Bugs.
IEEE Software, 25(5) :22–29.

Bloch, J. (2008).
Effective Java, 2nd Edition.
Addison-Wesley.

Bloch, J. (2018).
Effective Java, 3rd Edition.
Addison-Wesley.

Références II

Class Object (JSE8).
Javadoc of the class `Object` of JAVA SE 8.
https://docs.oracle.com/javase/9/docs/api/index.html?java/lang/Object.html.

Class Optional (JSE8).
Javadoc of the class `java.util.Optional` of JAVA 9.
https://docs.oracle.com/javase/9/docs/api/index.html?java/util/Optional.html.

Czarnecki, K. and Eisenecker, U. (2000).
Generative Programming : Methods, Tools, and Applications.
Addison-Wesley.

Javadoc (JSE8).
Javadoc Tool.
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

Références III

Kernighan, B. and Pike, R., editors (1984).
The UNIX Programming Environment.
Prentice-Hall.

Naftalin, M. (2015).
Mastering Lambdas : Java Programming in a Multicore World.
Mc Graw Hill, Oracle Press.

Naftalin, M. (2012).
The Lambda FAQ.
http://www.lambdafaq.org/.

PackageJavaUtilFunction (JSE8).
Javadoc of the package `java.util.function` of JAVA SE 8.
https://docs.oracle.com/javase/9/docs/api/java/util/function/package-summary.html.

Références IV

Saumont, P.-Y. (2017).

Functional Programming in Java—How functional techniques improve your Java programs.

Manning.

Urma, R.-G. (2014).

Tired of Null Pointer Exceptions ? Consider Using Java SE 8's `Optional` !

`https://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html`.