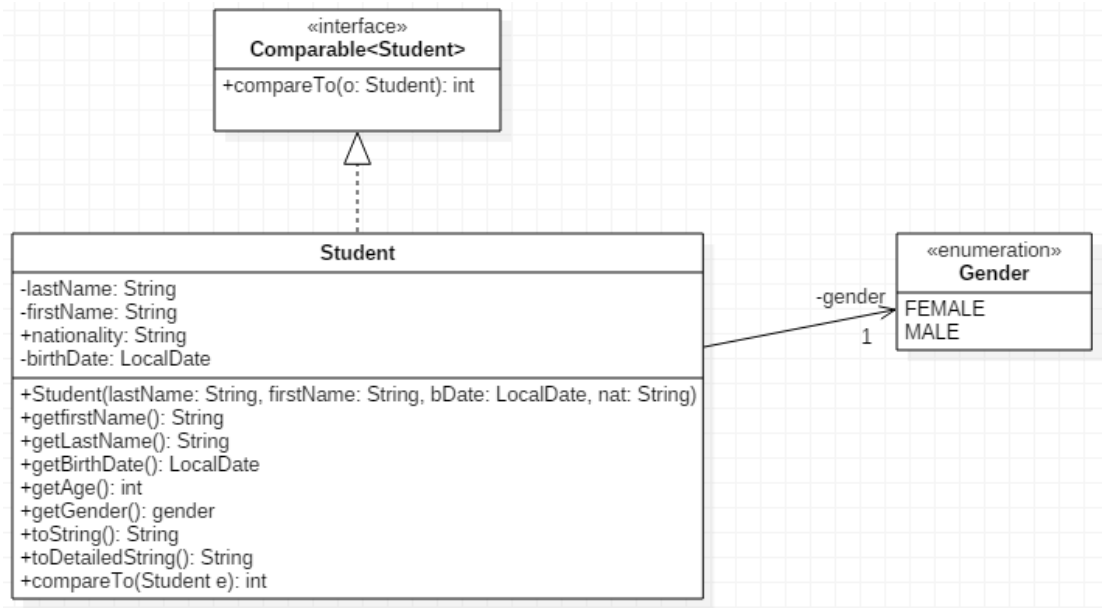
Interfaces fonctionnelles et Lambdas expressions en Java

Philippe Genoud
dernière mise à jour :26/02/2026 04:15

Lambdas

- Ecrire un programme qui étant donnée une liste d'étudiants affiche:
  - *les étudiants étrangers*
  - *les étudiantes*




```java
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    // TODO

    System.out.println("Les étudiantes");
    // TODO
}
```

Lambdas

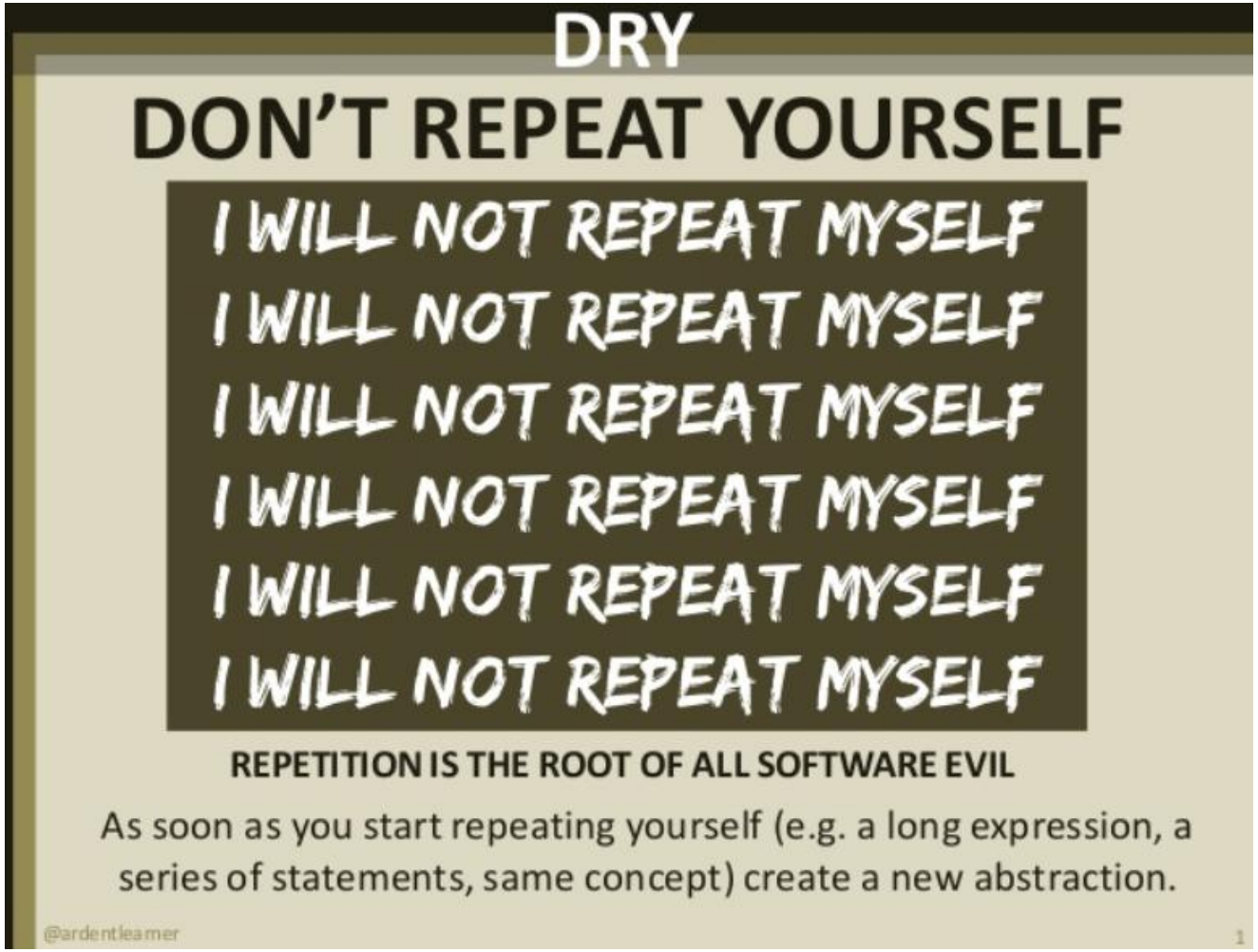
- Créer une méthode spécifique pour chaque critère de recherche

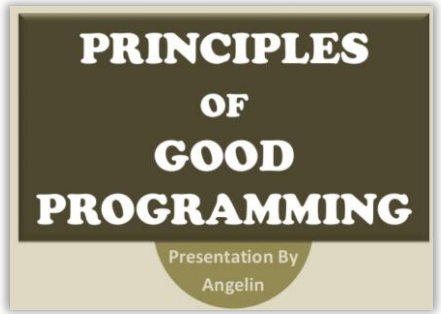
```java
private static void displayStudents(List<Student> studs, Gender gender) {
    for (Student s : studs) {
        if (gender == s.getGender()) {
            System.out.println(s);
        }
    }
}
```

```java
private static void displayForeignStudents(List<Student> studs) {
    for (Student s : studs) {
        if (!"FR".equals(s.getNationality())) {
            System.out.println(s);
        }
    }
}
```

```java
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayForeignStudents(studs);

    System.out.println("Les étudiantes");
    displayStudents(studs, Gender.FEMALE);
}
```

code très répétitif
seul le code correspondant au critère de recherche change





http://www.slideshare.net/ardentlearner/the-principles-of-good-programming

Lambdas

- Créer une méthode spécifique pour chaque critère de recherche

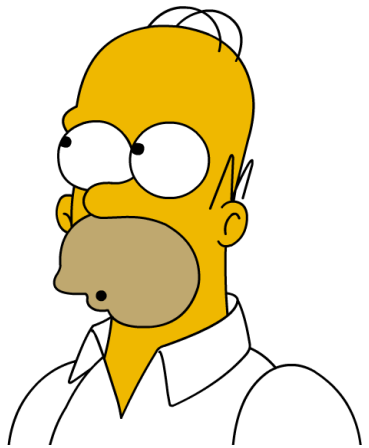
```java
private static void displayStudents(List<Student> studs, Gender gender) {
    for (Student s : studs) {
        if (gender == s.getGender()) {
            System.out.println(s);
        }
    }
}
```

```java
private static void displayForeignStudents(List<Student> studs) {
    for (Student s : studs) {
        if (!"FR".equals(s.getNationality())) {
            System.out.println(s);
        }
    }
}
```

```java
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayForeignStudents(studs);

    System.out.println("Les étudiantes");
    displayStudents(studs, Gender.FEMALE);
}
```

code très répétitif
seul le code correspondant au critère de recherche change

Comment séparer le code définissant le critère de recherche du code de parcours de la liste ?



- Créer une méthode spécifique pour chaque critère de recherche

```java
private static void displayStudents(List<Student> studs, Gender gender) {
    for (Student s : studs) {
        if (gender == s.getGender()) {
            System.out.println(s);
        }
    }
}
```

```java
private static void displayForeignStudents(List<Student> studs) {
    for (Student s : studs) {
        if (!"FR".equals(s.getNationality())) {
            System.out.println(s);
        }
    }
}
```

```java
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayForeignStudents(studs);

    System.out.println("Les étudiantes");
    displayStudents(studs, Gender.FEMALE);
}
```

code très répétitif
seul le code correspondant au critère de recherche change

sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

```
private static void displayStudents(List<Student> studs, Gender gender) {
    for (Student s : studs) {
        if (gender == s.getGender()) {
            System.out.println(s);
        }
    }
}
```

```
private static void displayForeignStudents(List<Student> studs) {
    for (Student s : studs) {
        if (!"FR".equals(s.getNationality())) {
            System.out.println(s);
        }
    }
}
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayForeignStudents(studs);

    System.out.println("Les étudiantes");
    displayStudents(studs, Gender.FEMALE);
}
```

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

① *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

② *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

```
public class ForeignStudFilter
        implements StudentFilter {
    @Override
    public boolean test(Student s) {
        return !"FR".equals(s.getNationality());
    }
}
```

```
public class FemaleStudFilter
        implements StudentFilter {
    @Override
    public boolean test(Student s) {
        return s.getGender() == Gender.FEMALE;
    }
}
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

③ *A chaque filtre correspond une classe réalisant l'interface*

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

1 *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

*Certes on isole le code correspondant au critère de recherche **mais une interface, deux classes... Est-ce qu'on y gagne vraiment ?***

```
...(String[] args) {

    Li...Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    ...out.println("Les étudiants étrangers");
    ...layStudents(studs, new ForeignStudFilter());

    ...em.out.println("Les étudiantes");
    ...layStudents(studs, new FemaleStudFilter());
```

3 *A chaque filtre correspond une classe réalisant l'interface*

2 *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

*Les classes **ForeignStudFilter** et **FemaleStudFilter** sont inutiles. On peut utiliser directement des **classes anonymes***

```
public class ForeignStudFilte...
        implements StudentFilte...
    @Override
    public boolean test(Student s) {
        return !"FR".equals(s.getNationali...y());
    }
}
```

```
public class FemaleStud...
        implements Stu...
    @Override
    public boolean tes...                    {
        return s.getGe...                    ender.FEMALE;
    }
}
```

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

1 *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

2 *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

Les classes **ForeignStudFilter** et **FemaleStudFilter** sont inutiles. On peut utiliser directement des **classes anonymes**

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

3 *A chaque filtre correspond une classe réalisant l'interface*

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

1 *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

2 *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return !"FR".equals(s.getNationality());
    }
}
```

Classes anonymes

```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return s.getGender() == Gender.FEMALE;
    }
}
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

3 *A chaque filtre correspond une classe réalisant l'interface*

*Les classes **ForeignStudFilter** et **FemaleStudFilter** sont inutiles. On peut utiliser directement des **classes anonymes***

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

① *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

② *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return !"FR".equals(s.getNationality());
    }
}
```

Classes anonymes

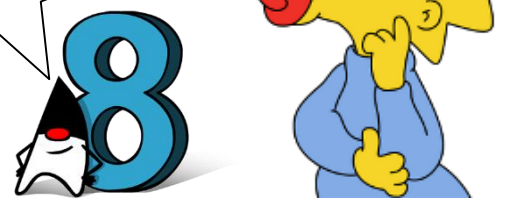
```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return s.getGender() == Gender.FEMALE;
    }
}
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

③ *A chaque filtre correspond une classe réalisant l'interface*

*Certes… on diminue le code à écrire, pas besoin de définir de nouvelles classes, mais syntaxe encore verbeuse et pas toujours très lisible …**Ne pourrait-on pas faire encore mieux ?***

*J'apporte la solution : **les expressions lambda***



Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

① *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

② *le type du filtre est une interface*

```
public interface StudentFilter {
    boolean test(Student s);
}
```

```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return !"FR".equals(s.getNationality());
    }
}
```

Classes anonymes

```
new StudentFilter() {
    @Override
    public boolean test(Student s) {
        return s.getGender() == Gender.FEMALE;
    }
}
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

③ *A chaque filtre correspond une classe réalisant l'interface*

Lambdas

- sous-traiter le critère de recherche à un objet chargé de filtrer les éléments de la liste

① *une seule méthode avec un objet filtre passé en paramètre*

```
private static void displayStudents(List<Student> studs, StudentFilter filter) {
    for (Student s : studs) {
        if (filter.test(s)) {
            System.out.println(s);
        }
    }
}
```

② *le type du filtre est une interface*

```
@FunctionalInterface
public interface StudentFilter {
    boolean test(Student s);
}
```

interface fonctionnelle ne contient qu'une seule méthode abstraite

expressions lambda peuvent être utilisées pour implémenter directement une interface fonctionnelle (pas besoin de rappeler nom de la méthode, plus concis que les classes anonymes)

```
(Student s) -> !"FR".equals(s.getNationality())
```

```
(Student s) -> s.getGender() == Gender.FEMALE
```

```
public static void main(String[] args) {

    List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
            "FR", "USA", "CH", "GB", "IT");

    System.out.println("Les étudiants étrangers");
    displayStudents(studs, new ForeignStudFilter());

    System.out.println("Les étudiantes");
    displayStudents(studs, new FemaleStudFilter());
}
```

③ *A chaque filtre correspond une classe réalisant l'interface*

(λ-calcul

Lambdas

- Lambda-calcul (ou λ-calcul):
  - *un système formel inventé par Alonzo Church dans les années 1930*

Alonzo Church 1903 - 1995

  - *idée de base: tout est fonction*
    - *concept central, expression générée par grammaire suivante*

```
expr -> λ var . expr   |   expr expr   |   var   |   (expr)
```

définition d'une fonction | application d'une fonction | variable | expression parenthésée

    - *expression lambda: programme qui quand il est évalué par beta-réductions retourne un résultat qui consiste en une autre expression lambda*

Lambdas

- Lambda-calcul (ou λ-calcul):
  - *modèle de calcul (computational model) à la base des langages de programmation fonctionnels : Lisp, Haskell, Ocaml …JavaScript*
  - *programmation fonctionnelle*
    - *style de programmation qui traite le calcul (computation) comme l'évaluation de fonctions mathématiques*
    - *élimine les effets de bord*
    - *traite les données comme étant immuables*
    - *expressions ont une transparence référentielle*
      - *une expression peut être remplacée par sa valeur sans changer le comportement du programme pour en savoir plus: http://stackoverflow.com/questions/210835/what-is-referential-transparency*
    - *les fonctions peuvent prendre d'autres fonctions comme arguments et retourner des fonctions comme résultat*
    - *préfère récursion plutôt que des les itérations explicites (boucle for)*

Lambdas

- programmation fonctionnelle
  - *permet d'écrire du code plus déclaratif, plus concis et plus lisible*
  - *permet de se focaliser sur le problème plutôt que sur le code*
  - *facilite le parallélisme*

- En introduisant les expressions λ Java 8 a joué le rattrapage
  - *de nombreux autres langages proposaient déjà un support (Scala, C# …)*

- L'une des principales difficultés a été d'introduire cette fonctionnalité en gardant une compatibilité ascendante
  - *éviter de recompiler les binaires existants*

λ-calcul)

Lambdas

- Les interface fonctionnelles et les lambda expressions permettent en java de manipuler des fonctions en tant **qu'objet standard**
- Une fonction définie par une lambda expression peut être utilisée comme
  - *valeur d'une variable*
  - *paramètre d'un fonction*
  - *comme type de retour d'une fonction*
- Une fonction qui a comme paramètre une autre fonction ou comme type de retour une fonction est appelée une fonction **d'ordre supérieur**

- Lambda expression composée de 3 parties

paramètres **->** **corps**

- paramètres:
  - *liste de paramètres formels entre parenthèses, séparés par des virgules*
    - *le type des paramètres peut être omis, mais si il est spécifié il doit l'être pour tous*
    - *les parenthèses peuvent être omises si un seul paramètre et que le type n'est pas précisé*
- corps:
  - *une simple expression*
  - *un bloc d'instructions*

```
(int x, int y) -> x + y

() -> 42

s -> s.length()

(x, y) -> System.out.println(x+y)
```

`return` implicite, peut être omis

```
(x, y, z) -> {
        System.out.println(x);
        if (y > z) {
           return y;
        } else {
           return x;
        }
}
```

Lambdas

- Le type d'une variable contenant une fonction est une **interface fonctionnelle**
  - *Une interface qui ne définit qu'une et une seule méthode*

```
public interface MonInterface {
    double methode1(double a, double b);
}
```

Interface fonctionnelle

```
public interface MonAutreInterface {
    double methode1(double x, double y);
    String methode2(String s);
}
```

Interface non fonctionnelle

- Pour pouvoir être affectée à une variable **f** de type interface fonctionnelle une lambda expression doit avoir une signature (type de retour, nombre et type des paramètres) compatible avec la signature définie dans la méthode de l'interface fonctionnelle

```
MonInterface f = (x, y) -> Math.pow(x, y) ;
```

```
MonInterface f = x -> Math.pow(x, x) ;
```

```
Lambda expression's signature does not match the signature of the
functional interface method methode1(double, double)
```

- Une lambda expression ne peut être affectée qu'à une variable de type interface fonctionnelle

```
MonAutreInterface f = (x, y) -> Math.pow(x, y) ;
```

```
The target type of this expression must be a functional interface
```

- Lorsque l'on veut évaluer (exécuter) une fonction définie dans une variable **f**, il suffit d'appeler la méthode définie par l'interface fonctionnelle ayant servi à typer **f**.

```
double res = f.methode1(2, 3);    // res vaut 8.0
```

- Rien ne distingue un interface fonctionnelle d'une interface *'normale'* si ce n'est le nombre de méthodes qu'elle définit

```
public interface MonInterface {
    double methode1(double a, double b);
}
```

Interface fonctionnelle

```
public interface MonAutreInterface {
    double methode1(double x, double y);
    String methode2(String s);
}
```

Interface non fonctionnelle

- Java8 fourni une annotation `@FunctionalInterface` destinée à la définition d'interfaces fonctionnelles
  - *Force le compilateur à vérifier que l'interface annotée est bien un interface fonctionnelle*

```
@FunctionalInterface
public interface MonInterface {
    double methode1(double a, double b);
}
```

```
@FunctionalInterface
public interface MonAutreInterface {
    int methode1();
    String methode2(String s);
}
```

```
Invalid '@FunctionalInterface' annotation; MonAutreInterface
is not a functional interface
```

  - *Pas obligatoire d'utiliser l'annotation mais c'est un outil*
    - *qui rend explicite aux développeurs l'intention de l'interface*
    - *qui empêchera toute modification de l'interface la rendant non fonctionnelle*

Références de méthodes

- Parfois, on veut manipuler une méthode d'un objet ou une méthode statique de classe en tant que valeur d'une variable de type fonctionnelle.
- Utilisation de la syntaxe d'une **référence de méthode**, qui utilise l'opérateur `::`
- 4 contextes différents dans lesquels on peut utiliser ne référence de méthode.

Contexte	Syntaxe	Exemple
Une méthode statique `m` de la classe `C`	`C::m`	`Math::random`
Une méthode `m` de l'instance `o`	`o::m`	`s::indexOf`
Un méthode `m` d'une instance quelconque de la classe `C`	`C::m`	`String::indexOf`
Un constructeur de la classe `C`	`C::new`	`String::new`

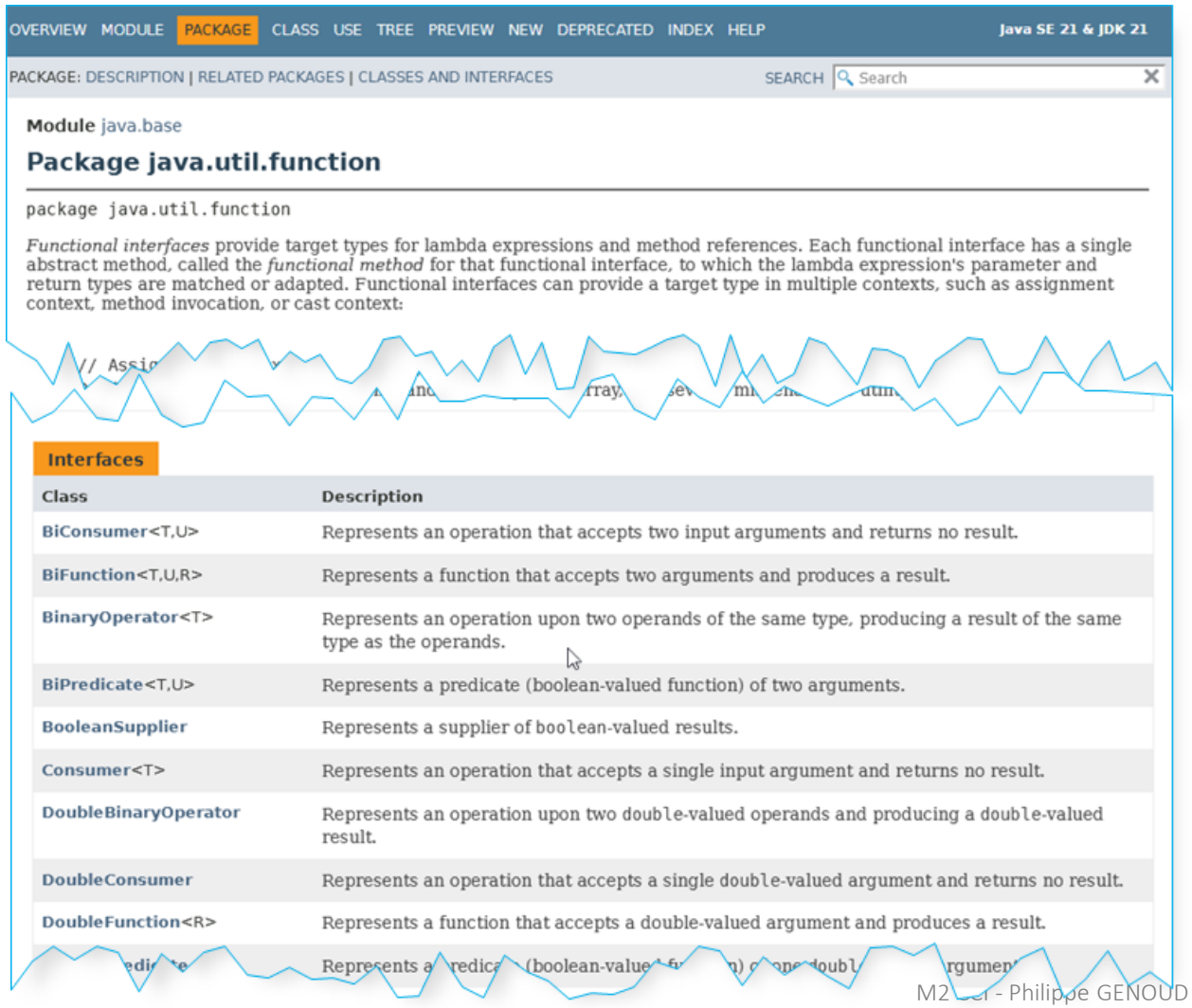
- quelques exemples commentés, avec leur équivalent sous forme de lambda expression

Exemple	lambda équivalente	Remarques
`Math::random`	`() -> Math.random()`	La méthode **`random()`** de l'objet **`Math`** est une fonction statique sans arguments.
`s::indexOf`	`x -> s.indexOf(x)`	En supposant que **`s`** est une instance de **`String`**, nous faisons référence ici à la méthode **`indexOf(String x)`** appliquée à l'objet **`s`**.
`String::indexOf`	`(s, x) -> s.indexOf(x)`	Vous remarquerez que dans ce cas, on fait référence à une méthode non statique sans préciser l'instance sur laquelle elle doit s'appliquer. Dans ce cas, l'instance est le premier paramètre de la lambda expression et les paramètres suivant représentent les paramètres de la méthode.

`String::new`	`() -> new String()`
	`(x) -> new String(x)`
	...
		Dans le cas d'un constructeur ou d'une méthode surchargés[a] le compilateur devra déterminer de quelle méthode il s'agit grâce au contexte de type dans lequel la lambda expression est définie. Les constructeurs sont des méthodes qui sont souvent surchargés.
		[a]Rappel: une méthode est dite surchargée lorsqu'il existe plusieurs signatures différentes (différents paramètres) pour un même nom de méthode. Par exemple, dans la classe **`String`**, la méthode **`substring`** est surchargée (**`substring(int from)`** et **`substring(int form, int to)`**).

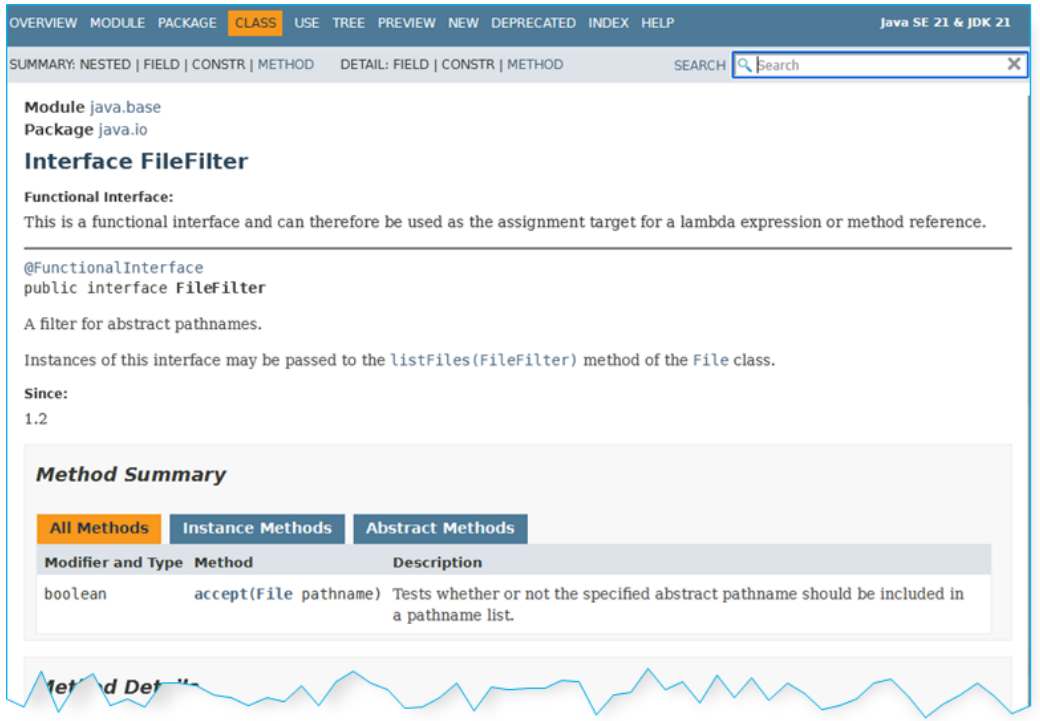
Lambdas

- Possibilité de définir ses propres interfaces fonctionnelle, mais cela n'est pas nécessaire dans la grande partie des cas.
- Java 8+ définit des interfaces fonctionnelles courantes utilisées dans de nombreuses API standard et que l'on peut réutiliser dans ses propres projets



des fonctions très générales dans le package [java.util.function]

des fonctions plus spécifiques dans d'autres packages (par ex. java.io)



- Quelques unes des interfaces fonctionnelles définies dans Java8 +

Fonctions à 1 paramètre

Interface	Exemple	Description
`Function<T,R>`	`x -> x.toString()`	Une fonction prenant un paramètre de type T et renvoyant un résultat de type R.
`UnaryOperator<T>`	`x -> x + x`	Une opération sur un paramètre de type T et renvoyant un résultat de même type T. Cette interface hérite de `Function<T,T>`.
`Predicate<T>`	`x -> "".equals(x)`	Une prédicat prenant un paramètre de type T et renvoyant un booléen.
`Consumer<T>`	`x -> System.out.println(x)`	Une fonction prenant un paramètre de type T et ne renvoyant aucun résultat.

- Quelques unes des interfaces fonctionnelles définies dans Java8 +

Toutes les fonctions précédentes existent aussi avec 2 paramètres

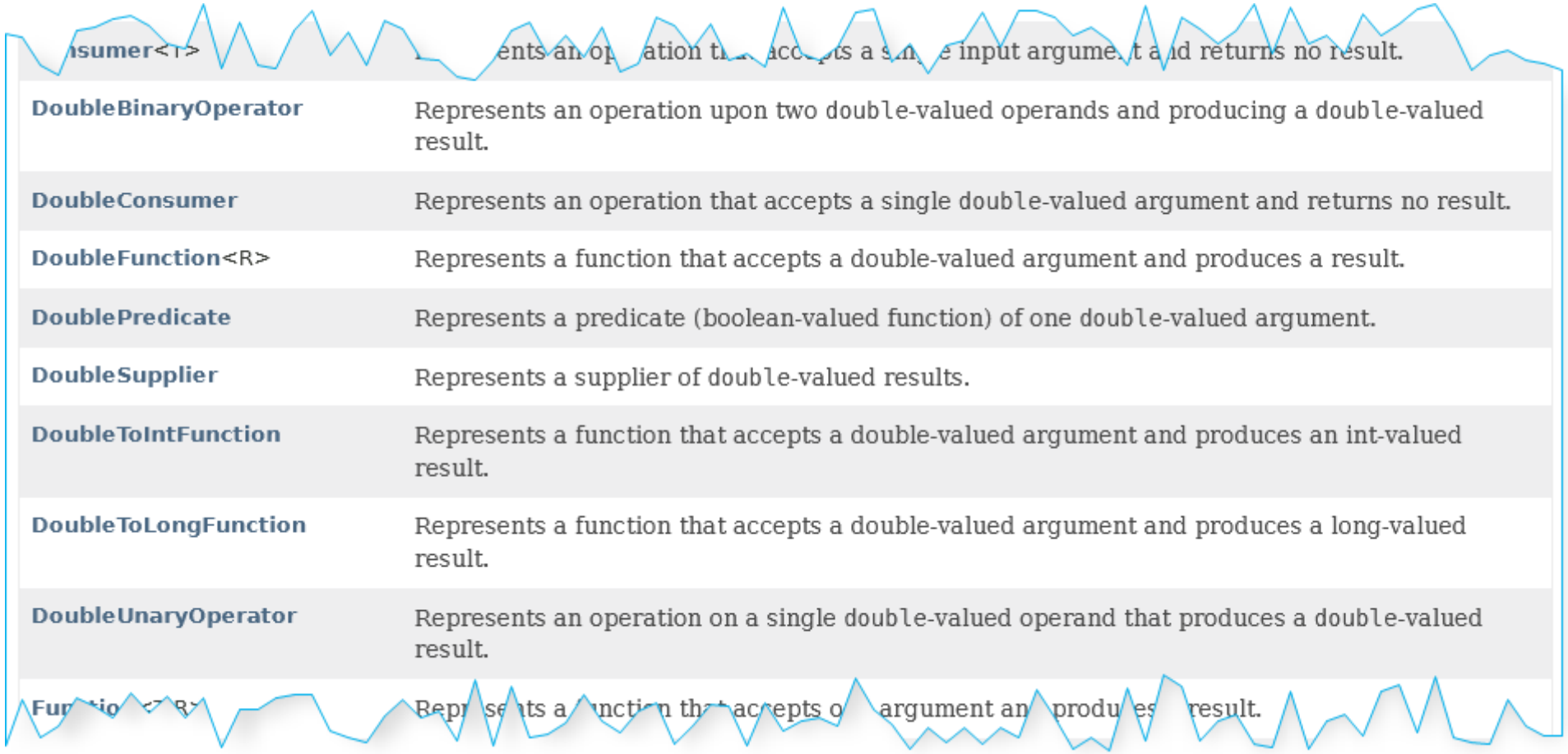
`BiFunction<T,U,R>`	`(x,y) -> x.toString() + y.toString()`	Une fonction prenant deux paramètres de type respectifs T et U et renvoyant un résultat de type R.
`BinaryOperator<T>`	`(x,y) -> x + y`	Une opération sur deux paramètres de type T et renvoyant un résultat de même type T. Cette interface hérite de `BiFunction<T,T,T>`.
`BiPredicate<T,U>`	`(x,y) -> x.equals(y)`	Une prédicat prenant deux paramètres de type respectifs T et U et renvoyant un booléen.
`BiConsumer<T,U>`	`(x,y) -> System.out.println(x + y)`	Une fonction prenant deux paramètres de type respectifs T et U et ne renvoyant aucun résultat.

Il existe aussi des fonctions sans paramètre

`Supplier<T>`	`() -> new T()`	Une fonction sans paramètre renvoyant un résultat de type T.

Lambdas

- Quelques unes des interfaces fonctionnelles définies dans Java8 +
  - *Java 8+ définit aussi des interfaces équivalentes qui prennent ou retournent des paramètres de types primitifs (en effet la généricité ne peut pas être utilisée avec les types primitifs)*
  - *exemples*



nsumer<T>	ents an op ation th acc pts a s n e input argume t a id returns no result.
DoubleBinaryOperator	Represents an operation upon two double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single double-valued operand that produces a double-valued result.
Fun tio <T,R>	Repr se ts a unct n th t ac epts o argument an produ es result.

- Exemple utilisation des interface fonctionnelles standards

Par quoi remplacer l'interface `StudentFilter` ?

```
@FunctionalInterface
public interface StudentFilter {
    boolean test(Student s);
}
```

```
import java.util.List;


public class Studs4 {
    public static void main(String[] args) {

        List<Student> studs = SudentsGenerator.generate(100, 1986, 1999,
                "FR", "USA", "CH", "GB", "IT");

        System.out.println("Les étudiants étrangers");
        displayStudents(studs, (Student s) -> !"FR".equals(s.getNationality()));

        System.out.println("Les étudiantes");
        displayStudents(studs, (Student s) -> s.getGender() == Gender.FEMALE);

    }

    private static void displayStudents(List<Student> studs, StudentFilter filter) {
        for (Student s : studs) {
            if (filter.test(s)) {
                System.out.println(s);
            }
        }
    }
}
```