PF
Programmation Fonctionnelle

Legond-Aubry Fabrice

fabrice.legond-aubry@parisnanterre.fr

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Streams

Compléments

Définitions (Hors Langage)

Exemple

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Streams

Compléments

Définitions (Hors Langage)

Exemple

Interfaces Fonctionnelles

Définition

- Les « fonctions » sont nécessaires pour implémenter le fonctionnel
  - ✓ On ne parle pas ici de méthode (au sens propre POO)
  - ✓ Un moyen additionnel pour implémenter du code (transverse)
  - ✓ On peut appeler cela des méthodes « anonymes » ou fonctionnelles
- Les « Interfaces fonctionnelles » sont nécessaires pour implémenter les fonctions
  - ✓ Attention, une interface fonctionnelle n'est PAS un élément de programmation fonctionnelle.
  - ✓ C'est juste un artifice Java pour implémenter des outils fonctionnels et ou manipuler les lambdas calculs

Interfaces Fonctionnelles

Définition

- Une interface Java :
  - ✓ Déclare une ou plusieurs méthodes qu'une classe doit implémenter
  - ✓ C'est un contrat
  - ✓ Java 8/9 : une interface peut avoir des méthodes « static » et/ou des méthodes « default »
  - ✓ Java 9 permet des méthodes « private » (utilisées en générale par les méthodes « default »)
  - ✓ Une méthode « default » est une méthode DONT L'IMPLEMENTATION SE TROUVE DANS L'INTERFACE
  - ✓ Une méthode « default » peut être sur-définie
- Une interface fonctionnelle java :
  - ✓ Une interface fonctionnelle à une seule méthode (« static » ou non)
  - ✓ Elle peut avoir en plus des méthodes « default »

Interfaces Fonctionnelles

Définition – Avant Java 8

✓ **Avant Java 8**, il n'y a pas d'interface fonctionnelle :

```
interface StringProcessor
{       String process(String x);  }
```

✓ Implémentation : Méthode 1

```
class StringProcessorNonAnonyme implements StringProcessor
{
   @Override
   public String process(String s) { return s; }
}
```

✓ Implémentation : Méthode 2 (classe anonyme)

```
StringProcessor SPAnonyme = new StringProcessor() {
   @Override
   public String process(String x)
        { return x.toUpperCase();  }
};
```

Interfaces Fonctionnelles

Définition

- Une interface fonctionnelle est annoté @FunctionalInterface
- Déclaration:

```
@FunctionalInterface
public interface StringComparator {
  int compare(String s1, String s2);
  default boolean isEmpty (String s)
         { return "".equals(s); }
}
…
// CLASSE ANONYME !!!! (On type avec une I.F.)
StringComparator mySC = new StringComparator {
   @Override
   int compare (String s1, String s2) {
         …
   }
}
```

Interfaces Fonctionnelles

Définition avec génériques

- Une interface fonctionnelle peut utiliser des génériques (avec limite)

```
@FunctionalInterface
interface TraiteJumeaux<X> {
    X process(X arg1, X arg2);
}
…
// CLASSE ANONYME !!!! (On type avec une I.F.)
TraiteJumeaux<Integer> multiplierEntiers
  = new TraiteJumeaux<>() {
      @Override
      public Integer process
           (Integer arg1, Integer arg2)
           { return arg1 * arg2; }
  };
```

Interfaces Fonctionnelles

Utilisations - Exemples

- Une fonction (ou méthode fonctionnelle) peut être utilisée directement à partir d'une référence de son interface
  - ✓ Cad : une interface fonctionnelle permet de faire une conversion automatique d'une référence à une méthode vers l'interface fonctionnelle
  - ✓ La **syntaxe ::** permet de référencer une méthode en indiquant le nom de la classe puis le nom de la méthode

- Vous devez créer une instance d'un type d'interface fonctionnelle accepté par la méthode
  - ✓ Conversion du type avec l'opérateur « :: »
  - ✓ Il y a inférence automatique des **TYPES DE FONCTIONS**

Interfaces Fonctionnelles

Utilisations - Exemples

- Des méthodes classiques peuvent ensuite utiliser des fonctions
  - ✓ On type une fonction comme on type une classe

- Ex dans les slides suivant
  - ✓ public static **<T>** void sort(**T[]** a, Comparator<? super **T**> c)
  - ✓ int compare(**T** o1, T o2)
  - ✓ https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-T:A-java.util.Comparator-
  - ✓ https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Interfaces Fonctionnelles

Utilisations - Exemples

```
Package FuncItf;

// IL EXISTE UN Comparator DANS java.util
@FunctionalInterface
public interface Comparator<T> {
  int compare(T t1, T t2);       // functional interface
}
public class Exemple {
  public static int lengthCmp(String s1, String s2) {
    ...
  }
  public static void main(String[] args) {
     ... Voir les slides suivant …
  }
}
```

Interfaces Fonctionnelles

Utilisations - Exemples

```
String[] strings = { "ab", "bvf", "jkl", "ooooo", "f", "", "sdfiusoiru", "nnvq", "ppppp" };

// OK. Inférence de type entre les "fonctions". Référence à la classe
java.util.Comparator<String> c1 = FuncItf.Exemple::lengthCmp;
Arrays.sort(strings, c1);

// OK. Non « static » nécessite une instance sur laquelle on applique « :: »
// Fonctionne avec ExempleAvecComparator implements java.util.Comparator
// ou toute autre I.F. implémentant une méthode compare compatible au typage !!!
// L'inférence de typage se fait à l'affectation dans un java.util.Comparator
ExempleAvecComparator eac = new ExempleAvecComparator();
java.util.Comparator<String> c2 = eac::compare;
Arrays.sort(strings, c2);
```

Interfaces Fonctionnelles

Utilisations - Exemples

```
//OK. UTILISATION D'UNE LAMBDA EXPRESSION. CF PLUS TARD …
// INFERENCE DE TYPE SUR LA FONCTION
java.util.Comparator<String> c3 = (String s1 ,String s2) -> { … };
Arrays.sort(strings, c3);

//OK. UTILISATION D'UNE CLASSE ANONYME
 java.util.Comparator<String> c4 = new java.util.Comparator<String>() {
        @Override
        public int compare(String o1, String o2) { … }
 };
 Arrays.sort(strings, c4);
```

Interfaces Fonctionnelles

Utilisations - Exemples

```
// KO.
FuncItf.Comparator<String> c5 = FuncItf.Exemple::lengthCmp;
// Arrays.sort attend un java.util.Comparator
// Typage impossible entre FuncITf.Comparator et java.util.Comparator
Arrays.sort(strings, c5);
```

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Streams

Compléments

Définitions (Hors Langage)

Exemple

Lambda Calculs

Définition des lambdas « expressions »

- Une lambda est une fonction anonyme (sans classe) – JLS 15.27
  - ✓ Permet par ex. d'implémenter une Interface Fonctionnelle
- La syntaxe des lambdas est une syntaxe raccourcie qui permet d'écrire une fonction anonyme qui va être convertit en objet dont la classe implante une interface fonctionnelle (qui est la fonction définie)
  - ✓ Forme: `LambdaParameters -> LambdaBody`
  - ✓ Flèche : '-' (moins) suivi de '>' (supérieur)

Lambda Calculs

Définition des lambdas « expressions »

- `LambdaParameters` sont les paramètres d'entrée
  - ✓ Il est possible d'éviter de spécifier le type des paramètres d'entrée

- Il existe deux types de `LambdaBody` :
  - ✓ Les expressions lambdas
  - ✓ Les blocs lambdas

Lambda Calculs

Définition des lambdas « expressions »

- LambdaBody de type expressions lambdas
  - ✓ le retour est typé par le résultat de l'expression ou void
  - ✓ Retourne le type de retour de la méthode java invoquée
  - ✓ Exemples:
    - // même type que x, le paramètre d'entrée
      x -> x + 1
    - // void
      list.forEach(e -> System.out.println(e));

Lambda Calculs

Définition des lambdas « expressions »

- LambdaBody de type bloc lambda
  - ✓ Toute variable utilisée, non déclarée dans le bloc ou en paramètre doit être « `final` » effectif (voir plus loin)
  - ✓ On peut déclarer des variables locales dans un bloc lambda
  - ✓ Le « `this` » d'un lambda référence l'élément du contexte appelant
  - ✓ Type de retour doit être soit **void-compatible** soit **value-compatible**
    - ➢ Erreur de compilation sinon
  - ✓ Exemple :

```
x -> { System.out.println("hello lambda"); }
```

Lambda Calculs

Définition des lambdas « expressions »

- Retour d'un LambdaBody de type bloc lambda
- Un bloc lambda est dit « void-compatible »
  - ✓ Si tous les retours sont des retours de types "return;" (sans rien)
  - ✓ Exemple
    - ➢ System.out.println
    - ➢ () -> { while (true); }
- Un bloc lambda est dit « value-compatible »
  - ✓ Si tous les retours sont des retours typés par un « return Expression;» ou du à une terminaison anormale (exception)
  - ✓ Le retour est typé par le(s) « return »(s) du bloc
  - ✓ Attention: En cas de retour de types différents
  - ✓ Exemple:
    - ➢ x **->** { return 1.20*x; }
    - ➢ () -> { throw new RuntimeException(); } // note: cette expression est aussi void-compatible !

Lambda Calculs

Lambdas – exemple (JLS 15.27)

- Exemples de lambdas sans paramètre d'entrée:

```
// Pas de paramètres ; lambda « bloc » ;
// pas « return » → retour est void
() -> {}
() -> { System.gc(); }
() -> System.gc();   // sous forme d'expression, System.gc() renvoie void

// HG2G, 42, La grande question sur la vie, l'univers et le reste
// Pas de paramètres ;  retourne 42 ;
// typage par inférence
// par ex. en fonction de l'I.F. à laquelle elle est assignée
() -> 42                  // lambda « expression » ;
() -> { return 42; }  // lambda « bloc »
```

Lambda Calculs

Lambdas – exemple (JLS 15.27)

- Exemples de lambdas sans paramètre d'entrée:

```
// Pas de paramètres ;  lambda « expression » ;
// retourne null ; typage par inférence
() -> null

// OK « value-compatible »
() -> { if (...) return 1; else return 0; }

// Erreur, ni « void » ni « value » compatible (un return pas toujours exécuté)
() -> { if (...) return "done"; System.out.println("done"); }
```

Lambda Calculs

Lambdas – exemples (JLS 15.27)

- Exemples avec paramètre d'entrée:

```
(int x) -> x+1                  // Lambda avec un paramètre d'entrée

(int x) -> { return x+1; }      // Idem mais avec un bloc au lieu d'une expression

(x) -> x+1                      // Idem mais avec un type de paramètre d'entrée qui
                                // sera inféré

x -> x+1                        // Idem pour montrer que les () sont optionnelles pour 1
seul
                                // paramètre (ne fonctionne pas avec 2 paramètres)
```

Lambda Calculs

Lambdas – exemples (JLS 15.27)

- Exemples avec paramètre d'entrée:

```
(String s) -> s.length()          // Lambda avec un paramètre de classe qui retourne un
nombre

(int x, int y) -> x+y             // Lambda avec 2 paramètres typés

(x, y) -> x+y                     // Lambda avec 2 paramètres inférés

(x, int y) -> x+y                 // ILLEGAL: interdiction de mixer paramètres typés
                                  // et inférés

(x, final y) -> x+y               // ILLEGAL: pas de modificateurs sur des types inférés
```

Lambda Calculs

Définition des lambdas « expressions »

- Les lambdas et les « variables » / « éléments externes »
  - ✓ Variable statique
  - ✓ Variable Locale du contexte parent /appelant de la lambda
    - ➢ Ce sont les variables locales non déclarés dans la lambda
  - ✓ Variable membre (d'instance)

- **La pureté, le déterminisme peuvent être compromis …**

- Une variable statique peut être accédée et manipulée
  - ✓ Pas de problème d'existence (pas besoin d'instance)
  - ✓ Pas de problème de changement de portée pour l'accès

Lambda Calculs

Définition des lambdas « expressions »

- Une lambda peut utiliser une variable locale du contexte parent
  - ✓ Les variables locales peuvent mourir avant que le code de la lambda soit exécutée (portée de la variable)
  - ✓ Le compilateur copie la valeur à la création de la lambda pendant l' exécution
  - ✓ Si utilisé pour une fonction d'une interface fonctionnelle, les valeurs des variables utilisées sont recopiées et envoyées en paramètre à la lambda
  - ✓ Une lambda qui capture des valeurs de variables mutables sera différentes à chaque appel (car la variable peut avoir une valeur différente).
    - ➢ La lambda ne peut pas être constante
  - ✓ Il n'est pas permis de capturer la valeur d'une variable dont on changera la valeur
    - ➢ Ex: var_local++ est interdit
    - ➢ Le compilateur vérifie que la variable est déclarée final ou est effectivement final

Lambda Calculs

Définition des lambdas « expressions »

- Une lambda peut utiliser une variable membre
  - ✓ Note: pour une classe anonyme, this → c'est l'instance de la classe anonyme
  - ✓ Note: pour une lambda, this → c'est le référence vers l'élément encapsulant (le père, le contexte extérieur, …)
  - ✓ « this » permet d'accéder aux valeurs des champs de l'objet utilisant la lambda
  - ✓ C'est la valeur du champ de this qui est capturé (this.lechamp)
  - ✓ il est possible de modifier la valeur de champs dans une lambda
- Des exemples seront présentés après les formes typiques de fonctions lambda que l'on peut rencontrer
  - ✓ Fonctions prédéfinies dans Java

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Streams

Compléments

Définitions (Hors Langage)

Exemple

Fonctions
Interface Fonctionnelles types prédéfinies

- Interfaces Fonctionnelles prédéfinies de **Package java.util.function**

Nom	Arguments	Retour	Description
`Consumer`	Oui (T)	Non	Consume une entrée et ne retourne rien **Stockage Lambda expression : signature (T) → void**
`Supplier`	Non	Oui (T)	Générer une sortie **Stockage Lambda expression : signature () → T**
`Predicate`	Oui (T)	boolean	Tests l'argument selon un critère et renvoie vrai ou faux. **Stockage Lambda expression : signature (T) → boolean**
`BiPredicate`	Oui (T,U) Arité: 2	boolean	Tests les 2 arguments de types potentiellement différents selon un critère et renvoie vrai ou faux. **Stockage Lambda expression : signature (T,U) → boolean**
`Function`	Oui (T)	Oui (R)	Convertie (map) un type vers un autre. Lambda avec une variable **Stockage de lambda expression : signature (T) → R**

Fonctions

Interface Fonctionnelles types prédéfinies

- Interfaces Fonctionnelles prédéfinies de **Package java.util.function**

Nom	Arguments	Retour	Description
`BiFunction`	Oui (T,U) Arité: 2	Oui (R)	Créer un résultat à partir de 2 paramètres vers un autre type **Stockage de lambda expression : signature (T,U) → R**
`UnaryOperator`	Oui (T)	Oui (T)	Représente une opération qui prend un paramètre et retourne un paramètre de même type. . Sous classe de **Function**. **Stockage de lambda expression : signature (T) → T** Exemple: ^2 est un UnaryOperator
`BinaryOperator`	Oui (T,T) Arité: 2	Oui (T)	Représente une opération qui prend deux paramètres et retourne un paramètre de même type. Sous classe de **BiFunction**. **Stockage de lambda expression : signature (T,T) → T** Exemple: + est un BinaryOperator
`Runnable`	Non	Non	Traitement à faire. **Stockage de lambda expression : signature () → void**

Fonctions

Interface Fonctionnelles types prédéfinies : Supplier

- Interface « Supplier » (Void → T) :

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

- Fonction dont le but est de créer des éléments
  - ✓ On peut utiliser le constructeur des objets via la référence « new »

    Supplier<String> = String::new
  - ✓ Il existe des interfaces fonctionnelles plus restrictives

    BooleanSupplier, DoubleSupplier, IntSupplier
  - ✓ pas de relation de sous typage entre Supplier et XxxxxxSupplier (ce sont des I.F)
  - ✓ Il peut retourner des valeurs différentes à chaque appel

Fonctions

Interface Fonctionnelles types prédéfinies : Supplier

- Exemples :

```
Random random = new Random();
Supplier<Integer> newRandomInt = () -> random.nextInt(10);

Supplier<Foo> makeFoo = () -> new Foo("hello“, 23, obj);

Supplier<User> userSupplier = User::new;
User user = userSupplier.get();
```

Fonctions

Interface Fonctionnelles types prédéfinies : Consumer

- Interface « Consumer » (T → Void) :

```
@FunctionalInterface
public interface Consumer<T> {
void accept(T t);
default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

- Fonction dont le but est de consommer des éléments
  - ✓ Produit des effets de bords
- andThen permet de renvoyer le résultat de accept puis d'appliquer sur l'élément original un autre accept
  - ✓ Permet donc d'appliquer plusieurs accept de Consumer indépendants sur un même élément
  - ✓ Permet de « dupliquer » un élément pour le faire consommer par plusieurs Consumer en //

Fonctions

Interface Fonctionnelles types prédéfinies : Consumer

- Exemple :

```
public class TestConsumerAndThen
{
    private static int sum  = 0; // Variable membre static non final !!
    private static int prod = 1; // Variable membre static non final !!
    public static void main(String[] args)
       {
              Consumer<Integer> consum  = x -> sum  += x;
              Consumer<Integer> conprod = x -> prod *= x;
              // Application 2 fois de 2 consumers INDEPENDANTS
              // On ne récupère pas le résultat de l'un pour l'autre
              // Malgré tout, il peut y avoir des effets de bords
              consum.andThen(conprod).accept(4);
              consum.andThen(conprod).accept(5);
              System.out.println("sum = " + sum + " prod =" + prod);
       }
}
```

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Predicate

- Fonction principale de l'I.F. « `Predicate` »
  - ✓ test qui prend un élément générique et renvoie une booléen « vrai » si la condition est vrai, faux sinon.
  - ✓ Rappel : on peut sur-définir les méthodes « `default` » pour gérer les null et éviter les NPE (par ex.)
  - ✓ La méthode « `static not` » inverse le résultat du calcul du prédicat passé en paramètre
  - ✓ La méthode « `negate` » renvoie un prédicat dont le résultat sera toujours le contraire du prédicat passé en paramètre (comportement de factory)
- Pour un « `BiPredicate` », on renvoie un booléen qui dépend de deux paramètres (T,U).

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Predicate

- Interface « Predicate » (T → Boolean) :

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
   default Predicate<T> and(Predicate<? super T> other) { … }
   default Predicate<T> negate() { … }
   static <T> Predicate<T> not(Predicate<? super T> target) { … }
   default Predicate<T> or(Predicate<? super T> other) { … }
   static <T> Predicate<T> isEqual(Object targetRef) { … }
```

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Predicate

- Interface « BiPredicate » (T,U) → Boolean :

```
@FunctionalInterface
public interface BiPredicate<T, U> {

   boolean test(T t, U u);

   default java.util.function.BiPredicate<T, U>
   and(java.util.function.BiPredicate<? super T, ? super U> other)
          { … }

   default java.util.function.BiPredicate<T, U>
   negate()
          { … }

   default java.util.function.BiPredicate<T, U>
   or(java.util.function.BiPredicate<? super T, ? super U> other)
          { … }
}
```

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Predicate

- Exemple :

```
Predicate<Integer> estPositif =  x -> x > 0;

// voir le code de l'I.F. Predicate, lazzy evaluation
System.out.println(  ((Predicate<Integer>)(x -> x > 0))
   .and(x -> x % 2 == 0)
   .test(2));

Predicate<String>  finitParPoint =
        (s) -> s.charAt(s.length()-1) == '.';
Predicate<String> p = String::equals;
```

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Function

- Function permet de stocker des lambdas calculs a 1 paramètre d'entrée et 1 paramètre de sortie
  - ✓ On peut utiliser le couple Void/null pour typer le sans paramètre

- La méthode « `compose` » permet de composer 2 « `function` »
  - ✓ Génére une « `function` » composée résultat
  - ✓ La nouvelle fonction appliquera d'abord la fonction passée en paramètre « `before` » de la méthode « `compose` » sur ses paramètres puis appliquera la fonction de l'objet de la méthode « `compose` » pour produire un retour
  - ✓ D'où la nécessite du type « `? extends T`» en retour de « `before` » pour nourrir « `apply (T t)` »

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Function

- La méthode « `andThen` » N'A PAS le même effet que pour la fonction « `Consumer` »
  - ✓ Génére une « `function` » composée résultat
  - ✓ La nouvelle fonction appliquera d'abord la fonction de l'objet de la méthode « `andThen` » puis appliquera la fonction passée en paramètre dans « `after` » pour produire un retour.
  - ✓ D'où la nécessité du type « `? super R` » en paramètre d'entrée de « `after` » pour nourrir la fonction de « `after` ».

- L'interface Fonction « `BiFunction` » :
  - ✓ « `apply` » utilise 2 paramètres typés T,U et retourne R
  - ✓ « `andThen` » adapté aussi
  - ✓ Pas de « `compose` »

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Function

- Interface Function T → R :

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
   default <V> java.util.function.Function<V, R> compose
      (java.util.function.Function<? super V, ? extends T> before)
      { ... }
   default <V> java.util.function.Function<T, V> andThen
      (java.util.function.Function<? super R, ? extends V> after)
      { ... }
   static <T> java.util.function.Function<T, T> identity()
      { return t -> t; }
}
```

Fonctions

Interface Fonctionnelles types prédéfinies : (Bi)Function

- Interface BiFunction (T,U) → R :

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    default <V> java.util.function.BiFunction<T, U, V>
        andThen(Function<? super R, ? extends V> after)
        {
              Objects.requireNonNull(after);
              return (T t, U u) -> after.apply(apply(t, u));
        }
}
```

Fonctions

Interface Fonctionnelles types prédéfinies : Function

- Exemples:

```
// mauvaise émulation d'un Supplier
Function<Void, Integer> funcSupplier = (x) -> 42;
Integer result = funcSupplier.apply(null);

// mauvaise émulation d'un Consumer
Function<Integer,Void> funcConsumer =
   (Integer x) -> {
     System.out.println ("then answer is : "+x); return null;
   };
funcConsumer.apply (42);

// emulation d'un prédicat
Function<Integer, Boolean> funcPredicate =
          (Integer i) -> i % 2 == 0;
funcPredicate.apply (42);
```

Fonctions

Interface Fonctionnelles types prédéfinies : UnaryOperator / BinaryOperator

- UnaryOperator est une limitation de Function, d'où son interface (T) → T

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
      static <T> UnaryOperator<T> identity()
            { return t -> t; }
}
```

- BinaryOperator est une limitation de BiFunction (fonction à 2 paramètres d'entrée), d'où son interface (T,T) → T

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T>
{ … }
```

Fonctions

Interface Fonctionnelles types prédéfinies : UnaryOperator / BinaryOperator

- Exemples :

```
IntUnaryOperator incrementeDe2 = x -> 2 + x;
private static IntUnaryOperator adder(int value) {
    return x -> x + value;
}

UnaryOperator<String> lower =
       String::toLowerCase;
System.out.println ("test: "+lower.apply("Bonjour"));

BinaryOperator<String> concat =
       (String s1, String s2) -> s1+" "+s2;
System.out.println ("test: "+concat.apply("Bonjour","Monde"));
```