Java 8 : Expression Lambda

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Classe anonyme ?

- Déclarée au moment de l'instanciation de sa classe mère, qui peut être
  - concrète
  - abstraite
  - interface
- Pouvant accéder aux attributs et méthodes de la classe englobante

Dans un `Java Project`, commençons par créer la classe `Personne` suivante

```
package org.eclipse.model;

public class Personne {

  private String nom;
  private String prenom;
  private int age;

  public Personne(String nom, String prenom, int age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }

  // + getters et setters + toString()
}
```

Considérons la classe abstraite suivante

```
package org.eclipse.model;

public abstract class IMiseEnForme {
  public abstract void afficherDetails();
}
```

Créons une instance de la classe fille anonyme de `IMiseEnForme` dans `Personne`

```
public class Personne {

  private String nom;
  private String prenom;
  private int age;

  public IMiseEnForme iMiseEnForme =  new IMiseEnForme() {

    @Override
    public void afficherDetails() {
      // TODO Auto-generated method stub
      System.out.println("nom = " + nom + ", prenom = " + prenom);
    }
  };

  public Personne(String nom, String prenom, int age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }

  // + getters et setters
}
```

Avec le code suivant, avons nous réellement instancier une classe abstraite ?

```
public IMiseEnForme iMiseEnForme = new IMiseEnForme() {

    @Override
    public void afficherDetails() {
      // TODO Auto-generated method stub
      System.out.println("nom = " + nom + ", prenom = " + prenom);
    }

};
```

© Achref EL

Réponse : non

- `iMiseEnForme` : nom de l'instance de la classe fille anonyme dérivée de `IMiseEnForme`
- Une classe fille d'une classe abstraite doit implémenter ses méthodes abstraites : d'où la présence de l'annotation `@Override` sur la méthode `afficherDetails()`

Pour tester et vérifier que `iMiseEnForme` est une instance de la classe fille anonyme dérivée de `IMiseEnForme`

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
        public static void main(String[] args) {
                Personne personne = new Personne("el mouelhi", "achref", 34);
                personne.iMiseEnForme.afficherDetails();
                System.out.println(personne.getClass());
                System.out.println(personne.getClass().getSuperclass());
                System.out.println(personne.iMiseEnForme.getClass());
                System.out.println(personne.iMiseEnForme.getClass().getSuperclass());
        }
}
```

© Achref

Pour tester et vérifier que `iMiseEnForme` est une instance de la classe fille anonyme dérivée de `IMiseEnForme`

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
        public static void main(String[] args) {
                Personne personne = new Personne("el mouelhi", "achref", 34);
                personne.iMiseEnForme.afficherDetails();
                System.out.println(personne.getClass());
                System.out.println(personne.getClass().getSuperclass());
                System.out.println(personne.iMiseEnForme.getClass());
                System.out.println(personne.iMiseEnForme.getClass().getSuperclass());
        }
}
```

© Achref

Le résultat sera

```
nom = el mouelhi, prenom = achref
class org.eclipse.model.Personne
class java.lang.Object
class org.eclipse.model.Personne$1
class org.eclipse.model.IMiseEnForme
```

Remarques

À la compilation, deux fichiers d'extension `.class` seront générés

- Le premier : `Personne`
- Le deuxième : `Personne$1` pour la classe anonyme $\Rightarrow$ 1 étant l'indice de cette classe anonyme (les classes anonymes seront numérotées dans l'ordre et la première aura l'indice 1)

© Achref EL M

Remarques

À la compilation, deux fichiers d'extension `.class` seront générés

- Le premier : `Personne`
- Le deuxième : `Personne$1` pour la classe anonyme ⇒ 1 étant l'indice de cette classe anonyme (les classes anonymes seront numérotées dans l'ordre et la première aura l'indice 1)

© Achref EL M

Remarques générales

- Une classe anonyme n'a pas d'identificateur et ne peut donc être instanciée qu'une seule fois (d'où son nom).
- Une classe anonyme est implicitement considérée comme finale (et ne peut donc pas être abstraite)

Transformons la classe abstraite précédente en interface

```
public interface IMiseEnForme {
  public abstract void afficherDetails();
}
```

© Acme H&H©

Testons le `main` précédent

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
        public static void main(String[] args) {
                Personne personne = new Personne("el mouelhi", "achref", 34);
                personne.iMiseEnForme.afficherDetails();
                System.out.println(personne.getClass());
                System.out.println(personne.getClass().getSuperclass());
                System.out.println(personne.iMiseEnForme.getClass());
                System.out.println(personne.iMiseEnForme.getClass().getSuperclass());
        }
}
```

© Achref

Testons le `main` précédent

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
        public static void main(String[] args) {
                Personne personne = new Personne("el mouelhi", "achref", 34);
                personne.iMiseEnForme.afficherDetails();
                System.out.println(personne.getClass());
                System.out.println(personne.getClass().getSuperclass());
                System.out.println(personne.iMiseEnForme.getClass());
                System.out.println(personne.iMiseEnForme.getClass().getSuperclass());
        }
}
```

© Achref

Le résultat sera

```
nom = el mouelhi, prenom = achref
class org.eclipse.model.Personne
class java.lang.Object
class org.eclipse.model.Personne$1
class java.lang.Object // car une interface n'est pas réellement une classe mère
```

Implémentation par défaut ?

- Jusqu'à **Java 7**, une interface ne peut contenir que des méthodes abstraites.
- Depuis **Java 8**, une interface peut proposer une implémentation par défaut pour ces méthodes.

Ajoutons une méthode avec une implémentation par défaut à notre interface `IMiseEnForme`

```
package org.eclipse.model;

public interface IMiseEnForme {
    public void afficherDetails();

    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

Pour tester

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
  public static void main(String[] args) {
    Personne personne = new Personne("el mouelhi", "achref", 34);
    personne.iMiseEnForme.afficherDetails();
    personne.iMiseEnForme.afficherNomComplet(personne.getNom(),
       personne.getPrenom());
  }
}
```

Pour tester

```
package org.eclipse.test;

import org.eclipse.model.Personne;

public class Main {
  public static void main(String[] args) {
    Personne personne = new Personne("el mouelhi", "achref", 34);
    personne.iMiseEnForme.afficherDetails();
    personne.iMiseEnForme.afficherNomComplet(personne.getNom(),
       personne.getPrenom());
  }
}
```

Le résultat sera

```
nom = el mouelhi, prenom = achref
el mouelhi achref
```

Interface fonctionnelle ?

- interface contenant une seule méthode abstraite
- pouvant contenir plusieurs méthodes avec une implémentation par défaut
- introduite dans **Java 8**
- utilisée souvent avec **Lambda expression**

Interface fonctionnelle, pourquoi ?

- code plus facile à lire, écrire et maintenir (une seule méthode abstraite par interface)
- supprimer le maximum de logique applicative d'un programme

On peut ajouter une annotation Java 8 (`@FunctionalInterface`) pour vérifier si on a bien respecté la contrainte

```
package org.eclipse.model;

@FunctionalInterface
public interface IMiseEnForme {
    public void afficherDetails();

    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

En ajoutant une nouvelle méthode abstraite, une erreur (Invalid '@FunctionalInterface' annotation ; IMiseEnForme is not a functional interface) s'affiche

```
package org.eclipse.model;

@FunctionalInterface
public interface IMiseEnForme {
    public void afficherNomMajuscule();

    public void afficherDetails();

    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

Le code permettant d'instancier la classe anonyme implémentant l'interface `IMiseEnForme` suivant

```
public IMiseEnForme iMiseEnForme =  new IMiseEnForme() {

    @Override
    public void afficherDetails() {
        // TODO Auto-generated method stub
        System.out.println("nom = " + nom + ", prenom = " + prenom );
    }

};
```

© ACM

Le code permettant d'instancier la classe anonyme implémentant l'interface `IMiseEnForme` suivant

```
public IMiseEnForme iMiseEnForme =  new IMiseEnForme() {

    @Override
    public void afficherDetails() {
        // TODO Auto-generated method stub
        System.out.println("nom = " + nom + ", prenom = " + prenom );
    }

};
```

© Acm

Peut être réécrit en utilisant les expressions Lambda

```
public IMiseEnForme iMiseEnForme = () -> System.out.println("nom = " +
   nom + ", prenom = " + prenom);
```

Remarques

- L'utilisation des expressions Lambda pour instancier une classe anonyme ne fonctionne que si cette dernière implémente une interface ayant une seule méthode abstraite $\Rightarrow$ interface **fonctionnelle**.
- `iMiseEnForme` n'est plus le nom d'un objet de la classe anonyme mais plutôt le nom de l'expression Lambda.

Expressions Lambda

- permettent d'implémenter et d'instancier les interfaces fonctionnelles (une sorte d'instance d'interface fonctionnelle)
- doivent avoir les mêmes paramètres et valeur de retour (signature) que la méthode abstraite de l'interface fonctionnelle
- existent depuis **Java 8**
- utilisent `->` pour séparer la partie paramètres et la partie traitement

© ACM

Expressions Lambda

- permettent d'implémenter et d'instancier les interfaces fonctionnelles (une sorte d'instance d'interface fonctionnelle)
- doivent avoir les mêmes paramètres et valeur de retour (signature) que la méthode abstraite de l'interface fonctionnelle
- existent depuis **Java 8**
- utilisent -> pour séparer la partie paramètres et la partie traitement

© ACM

Syntaxe

```
NomInterface nomLambdaExpression = ([arguments]) -> {
  traitement
};
```

Considérons l'interface fonctionnelle suivante

```
package org.eclipse.model;

@FunctionalInterface
public interface ICalcul {
        public int operationBinaire(int x, int y);
}
```

© Achref EL MOUELHI ©

Considérons l'interface fonctionnelle suivante

```
package org.eclipse.model;

@FunctionalInterface
public interface ICalcul {
        public int operationBinaire(int x, int y);
}
```

Définissons une expression lambda dans `main` qui permettra de calculer la somme (le `return` est implicite ici)

```
ICalcul plus = (int x, int y) ->  x + y;
```

© R. EL MOUELHI

Considérons l'interface fonctionnelle suivante

```
package org.eclipse.model;

@FunctionalInterface
public interface ICalcul {
        public int operationBinaire(int x, int y);
}
```

Définissons une expression lambda dans `main` qui permettra de calculer la somme (le `return` est implicite ici)

```
ICalcul plus = (int x, int y) ->  x + y;
```

Pour calculer la somme de deux entiers 3 et 5, il faut faire

```
System.out.println(plus.operationBinaire(3, 5));
// affiche 8
```

© R... EL MOUELHI ©

Typage de paramètre non obligatoire

```
ICalcul plus = (x, y) ->  x + y;
```

© Achref EL MOUELHI

Typage de paramètre non obligatoire

```
ICalcul plus = (x, y) ->  x + y;
```

Accolades obligatoires à partir de deux instructions

```
ICalcul plus = (x, y) ->  {
    int resultat = x + y;
    return resultat;
};
```

EL MOUELHI©

Une expression lambda peut utiliser une variable (ou un attribut) définie dans le contexte englobant

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) ->  x * i + y * j;
```

© Achref EL MOU

Une expression lambda peut utiliser une variable (ou un attribut) définie dans le contexte englobant

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) ->  x * i + y * j;
```

Pour tester

```
System.out.println(calculComplexe.operationBinaire(3, 5));
// affiche 21
```

Achref EL MOU

Une expression lambda ne peut redéfinir une variable (ou un attribut) définie dans le contexte englobant (Ceci est faux car `i` a été déclaré et initialisé juste avant)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int i = 5;
    return x * i + y * j;
};
```

© Achref EL M

Une expression lambda ne peut redéfinir une variable (ou un attribut) définie dans le contexte englobant (Ceci est faux car `i` a été déclaré et initialisé juste avant)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int i = 5;
    return x * i + y * j;
};
```

Une expression lambda ne peut modifier la valeur d'une variable (ou un attribut) définie dans le contexte englobant (Ceci est faux)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    i++;
    return x * i + y * j;
};
```

Une solution possible pour le problème précédent

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int k = i + 1;
    return x * k + y * j;
};
```

© Achref EL M

Une solution possible pour le problème précédent

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int k = i + 1;
    return x * k + y * j;
};
```

© Achref EL M

Pour tester

```
System.out.println(calculComplexe.operationBinaire(3, 5));
// affiche 24
```

Expressions Lambda

- Une expression lambda $\Rightarrow$ interface fonctionnelle
- Faudrait-il créer une interface fonctionnelle chaque fois qu'on a besoin de définir et exécuter une expression lambda ?

© Achref EL M

Expressions Lambda

- Une expression lambda $\Rightarrow$ interface fonctionnelle
- Faudrait-il créer une interface fonctionnelle chaque fois qu'on a besoin de définir et exécuter une expression lambda ?

Solution

- **Java 8** nous propose une quarantaine d'interfaces fonctionnelles
- Ces interfaces sont définies dans `java.util.function`

`java.util.function.Function<T,R>`

- Interface fonctionnelle possédant une méthode `R apply(T t)`
- Un seul paramètre d'entrée : variable de type `T`
- Valeur de retour : variable de type `R`

Considérons l'objet de type `Personne` suivant

```
Personne personne = new Personne("el mouelhi", "achref", 34);
```

© Achref EL MOUELHI ©

Considérons l'objet de type `Personne` suivant

```
Personne personne = new Personne("el mouelhi", "achref", 34);
```

Définissons une expression lambda qui prend comme entrée un objet de type `Personne` et qui retourne son nom concaténé à son prénom

```
Function<Personne, String> personneToString = (Personne p) -> p
   .getNom() + " " + p.getPrenom();
```

© Achref EL MOUELHI ©

Considérons l'objet de type `Personne` suivant

```
Personne personne = new Personne("el mouelhi", "achref", 34);
```

Définissons une expression lambda qui prend comme entrée un objet de type `Personne` et qui retourne son nom concaténé à son prénom

```
Function<Personne, String> personneToString = (Personne p) -> p
   .getNom() + " " + p.getPrenom();
```

© Achref EL MOUELHI ©

Pour l'exécuter, on appelle la méthode `apply`

```
String nomComplet = personneToString.apply(personne);

System.out.println(nomComplet);
// affiche el mouelhi achref
```

On peut simplifier l'expression précédente en supprimant les parenthèses et le type du paramètre d'entrée

```
Function<Personne, String> personneToString =
      p -> p.getNom() + " " + p.getPrenom();
```

© Achref EL MOUEL

On peut simplifier l'expression précédente en supprimant les parenthèses et le type du paramètre d'entrée

```
Function<Personne, String> personneToString =
      p -> p.getNom() + " " + p.getPrenom();
```

Pour tester

```
String nomComplet = personneToString.apply(personne);

System.out.println(nomComplet);
// affiche el mouelhi achref
```

bref EL MOUEL

Exercice 1

Écrire une expression lambda `capitalize`

- de type `Function`,
- prenant comme paramètre une chaîne de caractère,
- retournant la même chaîne en remplaçant le premier caractère par sa majuscule.

© AC

Exercice 1

Écrire une expression lambda `capitalize`

- de type `Function`,
- prenant comme paramètre une chaîne de caractère,
- retournant la même chaîne en remplaçant le premier caractère par sa majuscule.

© AC

Résultat attendu

```
System.out.println(capitalize.apply("wick"));
// affiche Wick
```

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

© Achref EL MOUELHI ©

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

Écrire une expression lambda `majeurs`

- de type `Function`,
- prenant comme paramètre une liste de `Personne`,
- retournant une liste contenant les noms de personnes majeures.

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

Écrire une expression lambda `majeurs`

- de type `Function`,
- prenant comme paramètre une liste de `Personne`,
- retournant une liste contenant les noms de personnes majeures.

Résultat attendu

```
System.out.println(majeurs.apply(personnes));
// affiche [dalton, benamar]
```

Correction

```
Function<List<Personne>, List<String>> majeurs = liste -> {
        List<String> noms = new ArrayList<>();
        for (Personne personne : liste) {
                if (personne.getAge() >= 18) {
                        noms.add(personne.getNom());
                }
        }
        return noms;
};
```

Exercice 3

Écrire une expression lambda `listToString`

- de type `Function`,
- prenant comme paramètre une liste de `String`,
- retournant la concaténation de toutes les chaînes de la liste
  - Première lettre de chaque élément en majuscule
  - Éléments concaténés séparés par le caractère espace

La méthode `andThen()`

permet le chaînage des fonctions

© Achref

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p.getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

© Achref EL MOUELHI ©

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p.getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

On peut définir une troisième qui est le chaînage des deux précédents

```
// on applique personneToString ensuite strToInt
Function <Personne, Integer> personneToInt = personneToString.andThen(strToInt);

Personne personne = new Personne("el mouelhi", "achref", 34);
int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

© MOUELHI ©

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p.getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

On peut définir une troisième qui est le chaînage des deux précédents

```
// on applique personneToString ensuite strToInt
Function <Personne, Integer> personneToInt = personneToString.andThen(strToInt);

Personne personne = new Personne("el mouelhi", "achref", 34);
int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

Type de la valeur de retour de la première = Type du paramètre d'entrée de la deuxième.

On peut aussi utilise `compose` qui permet le chaînage mais dans le sens inverse

```
// on applique personneToString ensuite strToInt
Function<Personne, Integer> personneToInt = strToInt.compose(
   personneToString);

int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

Exercice

Écrire une expression lambda `personnesToString`

- de type `Function`,
- prenant comme paramètre une liste de `Personne`,
- retournant une chaîne contenant les noms de personnes majeures séparés par le caractère espace : première lettre de chaque nom en majuscule.

Exercice

Écrire une expression lambda `personnesToString`

- de type `Function`,
- prenant comme paramètre une liste de `Personne`,
- retournant une chaîne contenant les noms de personnes majeures séparés par le caractère espace : première lettre de chaque nom en majuscule.

Résultat attendu

```
System.out.println(personnesToString.apply(personnes));
// affiche Dalton Benamar
```

`java.util.function.BiFunction<T1,T2,R>`

- Interface fonctionnelle similaire à `Function` possédant une méthode `apply` avec la signature `R apply(T1 t1, T2 t2)`
- Paramètres d'entrée : variable de type `T1` et une deuxième de type `T2`
- Valeur de retour : variable de type `R`

Exemple

```
BiFunction <String, Integer, Character> getChar = (str, i) -> str.charAt(i);
```

© Achref EL MOUELHI ©

Exemple

```
BiFunction <String, Integer, Character> getChar = (str, i) -> str.charAt(i);
```

Pour exécuter, on appelle la méthode `apply()`

```
char resultat = getChar.apply("Bonjour", 2);
System.out.println(resultat);
// affiche n
```

© Achref MOUELHI ©

Exemple

```
BiFunction <String, Integer, Character> getChar = (str, i) -> str.charAt(i);
```

Pour exécuter, on appelle la méthode `apply()`

```
char resultat = getChar.apply("Bonjour", 2);
System.out.println(resultat);
// affiche n
```

Le chaînage est possible avec `andThen()`

© Achref MOUELHI ©

Exercice 1

Écrire une expression lambda `nombreOccurrences`

- de type `BiFunction`,
- prenant comme premier paramètre un caractère
- prenant comme deuxième paramètre une chaîne de caractères
- retournant le nombre d'apparition du premier paramètre dans le deuxième.

Exercice 1

Écrire une expression lambda `nombreOccurrences`

- de type `BiFunction`,
- prenant comme premier paramètre un caractère
- prenant comme deuxième paramètre une chaîne de caractères
- retournant le nombre d'apparition du premier paramètre dans le deuxième.

© AU

Résultat attendu

```
System.out.println(nombreOccurrences.apply('a', "alain"));
// affiche 2
```

Exercice 2

Écrire une expression lambda `somme`

- de type `BiFunction`,
- prenant deux paramètres de type `Integer`
- retournant la somme des deux paramètres.

© Achref

Exercice 2

Écrire une expression lambda `somme`

- de type `BiFunction`,
- prenant deux paramètres de type `Integer`
- retournant la somme des deux paramètres.

© Achref

Résultat attendu

```
int resultat = somme.apply(5, 7);
System.out.println(resultat);
// affiche 12
```

Une solution possible

```
BiFunction <Integer, Integer, Integer> somme = (a, b) -> a + b;
```

© Achref EL MOUELHI

Une solution possible

```
BiFunction <Integer, Integer, Integer> somme = (a, b) -> a + b;
```

Remarque

Si les deux paramètres et la valeur de retour sont tous de même type, alors on peut simplifier la déclaration en utilisant l'interface fonctionnelle `BinaryOperator`.

`java.util.function.BinaryOperator<T>`

- Similaire à `BiFunction` mais avec un seul type générique
- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(T t1, T t2)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

© Achref EL MOUELHI ©

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
System.out.println(somme.apply(5, 7));
// affiche 12
```

© R... MOUELHI ©

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
System.out.println(somme.apply(5, 7));
// affiche 12
```

Le chaînage est possible avec `andThen()`

© R. MOUELHI ©

Exercice

Écrire une expression lambda `max`

- de type `BinaryOperator`,
- prenant deux paramètres de type `int`
- retournant le max des deux.

© Ach

Exercice

Écrire une expression lambda `max`

- de type `BinaryOperator`,
- prenant deux paramètres de type `int`
- retournant le max des deux.

© Ach

Résultat attendu

```
System.out.println(max.apply(2, 7));
// affiche 7
```

Remarque

Les méthodes `apply` définies dans les interfaces fonctionnelles précédentes n'acceptent pas les **Varargs**.

© Achref EL MOU

Remarque

Les méthodes `apply` définies dans les interfaces fonctionnelles précédentes n'acceptent pas les **Varargs**.

Solution

Définir une interface fonctionnelle personnalisée avec une méthode `apply` acceptant un nombre variable de paramètres.

Définissons l'interface fonctionnelle suivante

```
package org.eclipse.function;

@FunctionalInterface
public interface VariableArgumentsFunction<T, R> {
    T apply(R... params);
}
```

© Achref EL MOUEL

Définissons l'interface fonctionnelle suivante

```
package org.eclipse.function;

@FunctionalInterface
public interface VariableArgumentsFunction<T, R> {
    T apply(R... params);
}
```

Exercice

Écrire une expression lambda `somme`

- de type `VariableArgumentsFunction`,
- prenant un nombre indéterminé de paramètres de type `Integer`
- retournant la somme de tous les paramètres.

Solution

```
VariableArgumentsFunction<Integer, Integer> somme = numbers -> {
            int result = 0;
            for (Integer num : numbers) {
                result += num;
            }
            return result;
};

int sum1 = somme.apply(1, 2, 3, 4, 5);
int sum2 = somme.apply(10, 20, 30);

System.out.println("Somme 1 : " + sum1);
// Affiche : Somme 1 : 15

System.out.println("Somme 2 : " + sum2);
// Affiche : Somme 2 : 60
```

`java.util.function.Consumer<T>`

- Interface fonctionnelle possédant une méthode `accept` avec la signature `void accept(T t)`
- Paramètre d'entrée : variable de type `T`
- Pas de valeur de retour : elle consomme l'objet reçu en paramètre

Exemple

```
Consumer <Personne> ageIncrement =  p -> p.setAge(p.getAge() + 1);
```

© Achref EL MOUELHI ©

Exemple

```
Consumer <Personne> ageIncrement =  p -> p.setAge(p.getAge() + 1);
```

Pour exécuter, on appelle la méthode `accept()`

```
Personne personne = new Personne("el mouelhi", "achref", 34);
ageIncrement.accept(personne);

System.out.println(personne);
// affiche Personne [nom=el mouelhi, prenom=achref, age=35]
```

EL MOUELHI©

Exercice

Écrire une expression lambda `supprimerMineurs`

- de type `Consumer`,
- prenant un paramètre de type liste de `Personne`
- supprimant les mineurs de la liste.

© Achref El

Exercice

Écrire une expression lambda `supprimerMineurs`

- de type `Consumer`,
- prenant un paramètre de type liste de `Personne`
- supprimant les mineurs de la liste.

© Achref El

Résultat attendu

```
supprimerMineurs.accept(personnes);
System.out.println(personnes);
// affiche [Personne [nom=dalton, prenom=jack, age=18],
   Personne [nom=benamar, prenom=sophie, age=20]]
```

`java.util.function.Predicate<T>`

- Interface fonctionnelle possédant une méthode `boolean test(T t)`
- Paramètre d'entrée : variable de type `T`
- Type de valeur de retour : un booléen précisant si le paramètre `t` respecte le test.

Exemple

```
Predicate <Personne> contrainteAge = p -> p.getAge() >= 18;
```

© Achref EL MOUELHI

Exemple

```
Predicate <Personne> contrainteAge = p -> p.getAge() >= 18;
```

Pour exécuter, on appelle la méthode `test()`

```
if (contrainteAge.test(personne)) {
  System.out.println("Vous êtes adulte");
}
// affiche Vous êtes adulte
```

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

© Achref EL MOUELHI ©

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 1

Écrire une première expression lambda `any`

- de type `Predicate`,
- prenant comme paramètre une liste de `Personne`,
- retournant `true` s'il existe une personne majeur dans la liste, `false` sinon.

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 1

Écrire une première expression lambda `any`

- de type `Predicate`,
- prenant comme paramètre une liste de `Personne`,
- retournant `true` s'il existe une personne majeur dans la liste, `false` sinon.

Résultat attendu

```
System.out.println(any.test(personnes));
// affiche true
```

Exercice 2

Écrire une deuxième expression lambda `all`

- de type `Predicate`,
- prenant comme paramètre une liste de `Personne`,
- retournant `true` si toutes les personnes de la liste sont majeures, `false` sinon.

© Achr

Exercice 2

Écrire une deuxième expression lambda `all`

- de type `Predicate`,
- prenant comme paramètre une liste de `Personne`,
- retournant `true` si toutes les personnes de la liste sont majeures, `false` sinon.

© Achr

Résultat attendu

```
System.out.println(all.test(personnes));
// affiche false
```

Exercice 3

Écrire une deuxième expression lambda `estPremier`

- de type `Predicate`,
- prenant comme paramètre un `Integer`,
- retournant `true` si le paramètre est un nombre premier, `false` sinon.

© Achref EL

Exercice 3

Écrire une deuxième expression lambda `estPremier`

- de type `Predicate`,
- prenant comme paramètre un `Integer`,
- retournant `true` si le paramètre est un nombre premier, `false` sinon.

© Achref EL

Résultat attendu

```
System.out.println(estPremier.test(6));
// affiche false

System.out.println(estPremier.test(5));
// affiche true
```

`java.util.function.BiPredicate<T,R>`

- Interface fonctionnelle possédant une méthode `test` avec la signature
  `boolean test(T1 t1, T2 t2)`
- Paramètres d'entrée : une variable de type `T1` et une de type `T2`
- Type de valeur de retour : un booléen précisant si les paramètre `t` et `r` respectent le test.

Exemple

```
BiPredicate <Personne, Integer> contrainteAge =
    (p, x) -> p.getAge() >= x;
```

© Achref EL MOUELHI

Exemple

```
BiPredicate <Personne, Integer> contrainteAge =
     (p, x) -> p.getAge() >= x;
```

Pour exécuter, on appelle la méthode `test()`

```
if (contrainteAge.test(personne, 18)) {
  System.out.println("Vous êtes adulte");
}
// affiche Vous êtes adulte
```

Exercice 1

Écrire une expression lambda `containsLength`

- de type `BiPredicate`,
- prenant comme premier paramètre une liste de `String`,
- prenant comme deuxième paramètre un entier,
- retournant `true` si la liste contient un élément dont la taille correspond au deuxième paramètre, `false` sinon.

© Achref EL M

Exercice 1

Écrire une expression lambda `containsLength`

- de type `BiPredicate`,
- prenant comme premier paramètre une liste de `String`,
- prenant comme deuxième paramètre un entier,
- retournant `true` si la liste contient un élément dont la taille correspond au deuxième paramètre, `false` sinon.

© Achref EL M

Résultat attendu

```
var noms = new ArrayList<String>(Arrays.asList("wick", "dalton", "
   benamar", "maggio"));
System.out.println(containsLength.test(noms, 3));
// affiche false
System.out.println(containsLength.test(noms, 4));
// affiche true
```

Exercice 2

Écrire une expression lambda `exists`

- de type `BiPredicate`,
- prenant comme premier paramètre une liste de `Personne`,
- prenant comme deuxième paramètre une `Personne`,
- retournant `true` si la liste contient le deuxième paramètre, `false` sinon.

© Achr

Exercice 2

Écrire une expression lambda `exists`

- de type `BiPredicate`,
- prenant comme premier paramètre une liste de `Personne`,
- prenant comme deuxième paramètre une `Personne`,
- retournant `true` si la liste contient le deuxième paramètre, `false` sinon.

© Achi

Résultat attendu

```
System.out.println(exists.test(personnes, new Personne("wick", "john",
   16)));
// affiche true
```

`java.util.function.IntFunction<T>`

- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(Integer t)`
- Paramètre d'entrée : variable de type `Integer`
- Valeur de retour : valeur de type `T`

Exemple

```
IntFunction <String> parity = i -> (i % 2 == 0) ? "pair" : "impair";
```

© Achref EL MOUELHI ©

Exemple

```
IntFunction <String> parity = i -> (i % 2 == 0) ? "pair" : "impair";
```

Pour tester

```
System.out.println(parity.apply(4));
// affiche pair

System.out.println(parity.apply(5));
// affiche impair
```

EL MOUELHI ©

Exercice 1

Écrire une expression lambda `diviseurs`

- de type `IntFunction`,
- prenant comme paramètre un `Integer`,
- retournant une liste contenant tous ses diviseurs.

© Achr

Exercice 1

Écrire une expression lambda `diviseurs`

- de type `IntFunction`,
- prenant comme paramètre un `Integer`,
- retournant une liste contenant tous ses diviseurs.

© Achr

Résultat attendu

```
System.out.println(diviseurs.apply(10));
// affiche [1, 2, 5, 10]
```

Exercice 2

Écrire une expression lambda `premiers`

- de type `IntFunction`,
- prenant comme paramètre un `Integer` nommé `n`,
- retournant une liste contenant tous les nombres premiers $\leq$ `n`.

© Achr

Exercice 2

Écrire une expression lambda `premiers`

- de type `IntFunction`,
- prenant comme paramètre un `Integer` nommé `n`,
- retournant une liste contenant tous les nombres premiers $\leq$ `n`.

© Achr

Résultat attendu

```
System.out.println(premiers.apply(10));
// affiche [1, 2, 3, 5, 7]
```

`java.util.function.ToIntFunction<T>`

- Interface fonctionnelle possédant une méthode `applyAsInt` avec la signature `Integer apply(T t)`
- Paramètre d'entrée : valeur de type `T`
- Valeur de retour : `Integer`

Exemple

```
ToIntFunction<String> length = (str) -> str.length();
```

© Achref EL MOUELHI

Exemple

```
ToIntFunction<String> length = (str) -> str.length();
```

Pour tester

```
System.out.println(length.applyAsInt("Bonjour"));
// affiche 7
```

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "cooper");
```

© Achref EL MOUELHI ©

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "cooper");
```

Exercice

Écrire une expression lambda `maxLength`

- de type `ToIntFunction`,
- prenant comme paramètre une liste de `String`,
- retournant la longueur de la chaîne contenant le plus grand nombre de caractères.

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "cooper");
```

Exercice

Écrire une expression lambda `maxLength`

- de type `ToIntFunction`,
- prenant comme paramètre une liste de `String`,
- retournant la longueur de la chaîne contenant le plus grand nombre de caractères.

Résultat attendu

```
System.out.println(maxLength.applyAsInt(marques));
// affiche 8
```

`java.util.function.Supplier<T>`

- Interface fonctionnelle possédant une méthode `get` avec la signature `T get()`
- Paramètre d'entrée : aucun
- Valeur de retour : valeur de type `T`

Exemple

```
Supplier <Double> reel = () -> Math.random() * 100;
```

© Achref EL MOUELHI

Exemple

```
Supplier <Double> reel = () -> Math.random() * 100;
```

Pour tester

```
System.out.println(reel.get());
// affiche un nombre réel entre 0 et 100
```

Exercice

Écrire une expression lambda `getRandomLetter`

- de type `Supplier`,
- retournant un caractère alphabétique aléatoire.

© Achref EL

Exercice

Écrire une expression lambda `getRandomLetter`

- de type `Supplier`,
- retournant un caractère alphabétique aléatoire.

© Achref EL

Résultat attendu

```
System.out.println(getRandomLetter.get());
// affiche un caractère
```

Autres interfaces fonctionnelles prédéfinies

- `DoubleFunction`, `DoubleConsumer`, `DoubleBinaryOperator`, `DoublePredicate`, `DoubleSupplier`...
- `LongFunction`, `LongConsumer`, `LongBinaryOperator`, `LongPredicate`, `LongSupplier`...
- `UnaryOperator`
- `...`

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

© Achref EL MOUELHI ©

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

Syntaxe

```
FirstPart::secondPart
```

© Achref EL MOUELHI ©

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

Syntaxe

```
FirstPart::secondPart
```

© Achref EL MOUELHI ©

Explication

- `FirstPart` : le nom d'une classe, interface ou objet
- `SecondPart` : le nom d'une méthode

Étant donné le contenu de la classe `Main` suivant

```
public class Main {
  public static int somme(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
  // TODO Auto-generated method stub

  }
}
```

© Acme

Étant donné le contenu de la classe `Main` suivant

```
public class Main {
  public static int somme(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
  // TODO Auto-generated method stub

  }
}
```

© Acme

Hypothèse

On veut utiliser l'interface fonctionnelle `ICalcul` pour qu'elle retourne la somme de `x` et `y`.

Un solution possible mais redondante

```
public class Main {
  public static int somme(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
  // TODO Auto-generated method stub
    ICalcul iCalcul = (a, b) -> a + b;
  }
}
```

© Achref EL MOUEL

Un solution possible mais redondante

```
public class Main {
  public static int somme(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
  // TODO Auto-generated method stub
    ICalcul iCalcul = (a, b) -> a + b;
  }
}
```

Avec Java 8, on a la possibilité de référencer une méthode existante

```
public class Main {
  public static int somme(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
  // TODO Auto-generated method stub
    ICalcul iCalcul = Main::somme;
  }
}
```

Exemple avec un `Consumer`

```
public class Main {

  public static void main(String[] args) {
  // TODO Auto-generated method stub
    Consumer <Personne> afficher = System.out::print;
        afficher.accept(personne);
        // affiche Personne [nom=el mouelhi, prenom=achref, age=34]
  }
}
```

© Achref EL MOUELHI

Exemple avec un `Consumer`

```
public class Main {

  public static void main(String[] args) {
  // TODO Auto-generated method stub
    Consumer <Personne> afficher = System.out::print;
        afficher.accept(personne);
        // affiche Personne [nom=el mouelhi, prenom=achref, age=34]
  }
}
```

Exemple avec un constructeur

```
public class Main {

  public static void main(String[] args) {
  // TODO Auto-generated method stub
    Supplier <Personne> constructeur =  Personne::new;
    Personne p = constructeur.get();
    System.out.println(p);
    // affiche Personne [nom=null, prenom=null, age=0]
  }
}
```