13. Syntaxe avancée

13. Syntaxe avancée

- 13.1. Nombre variable d’arguments

Nombre variable d'arguments dans une méthode

Via utilisation d'ellipsis …

Conditions

- Un seul argument de type ellipsis
- Obligatoirement le dernier de tous les arguments
- Arguments en nombre variable: de type primitif ou référence
- Syntaxe: ***typeArgument*** **…** ***nomArgument***

Dans le code de la méthode:
Accès aux différents arguments via un tableau dont le nom est ***nomArgument***

Exemple de méthode avec un nombre variable d'arguments

```
public class MyClass {
    public int bookTotalPages (Book ... books)
    {  int totalPages = 0;
       for (int i = 0; i < books.length; i++)
      { totalPages += books[i].getPagesCount( ); }

 /* Autre boucle possible:
      for (Book book : books)
      { totalPages += book.getPagesCount( ); }*/

       return totalPages;
    }
}
```

Arguments: 0, 1 ou plusieurs objets de type Book

Exemple d'appel de méthode avec un nombre variable d'arguments

```
Book book1, book2, book3, book4;
 MyClass m = new MyClass ();
 int totalPages;
 …
// Exemples d'appel de la méthode
 totalPages  = m.bookTotalPages(  ) ;
 totalPages  = m.bookTotalPages( book1 ) ;
 totalPages  = m.bookTotalPages( book1, book2, book3, book4 ) ;
```

Autre exemple

public static void main(String... args)

13. Syntaxe avancée

- 13.1. Nombre variable d’arguments
- 13.2. Expressions Lambda

A partir de Java 8

Une expression lambda ressemble à une déclaration de méthode sans nom

Syntaxe:

- Liste d'arguments
  - Séparés par des virgules
  - Entre parenthèses
    - Si un seul argument: parenthèses facultatives
- -> (flèche)
- Corps
  - Soit une expression
  - Soit un bloc d'instructions entre entre { }

Exemples d'expressions lambda

- **(a,b,c) -> (a+b)*c**
  - A la place d'une méthode déclarée avec
    - Arguments : 3 int
    - Type de retour : 1 int
- **(book1, book2) -> {if (book1.getPagesCount() > book2.getPagesCount())**
  return book1;
  else return book2;}
  - A la place d'une méthode déclarée avec
    - Arguments : 2 objets de la classe Book
    - Type de retour : 1 objet de la classe Book
- **p -> p.getGender() = = Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25**
  - A la place d'une méthode déclarée avec
    - Argument : 1 objet de la classe Person
    - Type de retour : 1 booléen
- **email -> System.out.println(email)**
  - A la place d'une méthode déclarée avec
    - Argument : 1 String
    - Type de retour : void

Interface fonctionnelle =

une interface qui ne contient qu'une seule déclaration de méthode

Une expression lambda peut s'utiliser là où on attend le code d'une méthode

⇨ Ex: là où une interface fonctionnelle est déclarée comme argument

Exemple 1

```
public interface OperationInterface
   {   int operation (int a, int b);   }
```

→ Interface fonctionnelle car ne contient qu'une méthode

```
public class Calculator
  { public int calculate (int a, int b, OperationInterface op)
            {return op.operation (a, b); }
```

→ Classe qui utilise l'interface

Sans expression Lambda ⇨ *Il faut créer des classes qui implémentent l'interface pour pouvoir appeler la méthode calculate*

```
public class Addition implements OperationInterface
   { @Override
     public  int operation (int a, int b) { return a + b; }
   }
public class Subtraction implements OperationInterface
   {@Override
     public  int operation (int a, int b) { return a - b; }
   }
```

→ Classes qui implémentent l'interface

Appel de la méthode calculate: création d'objets de type Addition et Subtraction

```
public static void main(String... args) {
   Calculator calculator = new Calculator();

   Addition addition = new Addition();
   System.out.println("40 + 2 = " + calculator.calculate (40, 2, addition));

   Subtraction subtraction = new Subtraction();
   System.out.println("20 - 10 = " + calculator.calculate (20, 10, subtraction));
}
```

Avec expression Lambda ⇨ *Pas de création de classes qui implémentent l'interface*

```
public interface OperationInterface
   {   int operation (int a, int b);   }
```

→ Interface fonctionnelle

```
public class Calculator
  { public int calculate (int a, int b, OperationInterface op)
            {return op.operation (a, b); }
```

→ Classe qui utilise l'interface

```
public static void main(String... args) {

   Calculator calculator = new Calculator();

   System.out.println("40 + 2 = " + calculator.calculate (40, 2, (a, b) -> a + b ) );

   System.out.println("20 - 10 = " + calculator.calculate (20, 10, (a, b) -> a – b ) );
```

Expressions lambda

13. Syntaxe avancée

- 13.1. Nombre variable d'arguments
- 13.2. Expressions Lambda
- 13.3. Opérations sur agrégats

Flux (stream) d’éléments

⇨ éléments traités à travers un pipeline d’opérations

Pour obtenir la source de données du flux à partir d’une collection:

Appel de la méthode **stream()** sur la collection

Opérations d’agrégats sur un flux

- **filter**
  - Paramètre: un prédicat (retourne un booléen)
  - Sortie: un stream
- **map**
  - Paramètre: une fonction à appliquer
  - Sortie: un stream
- **opération terminale (ex: forEach)**
  - Paramètre: une opération terminale
  - Sortie: une valeur primitive, une collection ou void (ex: forEach)

Ces opérations d’agrégats acceptent les lambda expressions en paramètres

Exemple

ArrayList <Person> persons = … ;

```
persons
.stream( )
.forEach( p -> System.out.println(p) );
```

→ Affiche la description de toutes les personnes

```
persons
.stream( )
.filter( p ->  p.getAge( ) > 12 )
.forEach( p -> System.out.println(p.getName( ) ) );
```

→ Affiche le nom des plus de 12 ans

```
persons
.stream( )
.map( p -> p.getAge( ) )
.forEach( p -> System.out.println(p) );
```

→ Affiche l'âge de toutes les personnes

```
persons
.stream( )
.filter( p ->  p.getAge( ) >= 12 && p.getAge( ) <= 60 )
.map( p -> p.getName( ) )
.forEach( p -> System.out.println(p) );
```

→ Affiche le nom des personnes entre 12 et 60 ans