Cours 6 :

Génériques (notions avancées),
Les exceptions (notions avancées),
Java 8 (interfaces, trait, lambda)

Auteurs : CHAUDET Christelle, MIGEON Frédéric – Intervenant : BODEVEIX Jean-Paul,

Génériques Avancés

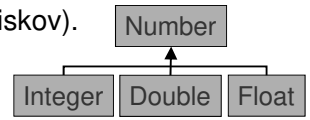
- Fonctionnement du sous-typage en Java
- Application du sous-typage aux génériques
  - Jokers (wildcards)
    - Avec extends
    - Avec super
  - Restriction sur les jokers

Sous-typage et principe de substitution (1/2)

- Sous-typage :
  - Caractéristique essentielle des langages orientés objet
  - Types et sous-types liés par la clause extends ou implements
  - Exemple :
    - Integer et Double sous-types de Number
    - ArrayList<E> sous-type de List<E>
    - List<E> sous-type de Collection<E>
  - Relation transitive : dans l'exemple précédent ArrayList<E> est sous-type de Collection<E>
  - Attention : Collection<Integer> n'est pas un sous-type de Collection<Object>

Sous-typage et principe de substitution (2/2)

- Principe de substitution : "il doit être possible de substituer n'importe quel objet instance d'une ***sous-classe*** à n'importe quel objet instance d'une ***superclasse*** sans que la sémantique du programme écrit dans les termes de la superclasse ne soit affectée." (Barbara Liskov).



- Exemple :
  - List<Number> nombres = new ArrayList<Number>()
    nombres.add(2); // int –boxing-> Integer
    // Integer sous-type de Number
    nombres.add(3.14);
    assert nombres.toString().equals("[2,3.14]");
  - Attention List<Integer> **n'**est **pas** sous-type de List<Number> **mais** Integer[] **est** sous-type de Number[]

Jokers (1/2)

- La spécification des paramètres réels de type n'implique pas le sous-typage entre les classes parametrées.
  Collection<Integer> ne specialise pas Collection<Object>

- Il existe un super type de toutes les instanciations de classe générique : le caractère joker " **?** "
  Collection<Integer> spécialise Collection<?>

- Peut servir :
  - à typer certaines expressions :
    - variables locales,
    - attributs,
  - à simplifier ou à préciser des écritures.

Jokers (2/2)

- Méthode ajoutée dans la classe Paire
  public static void affiche (Paire<?,?> p) {
  System.*out*.println(p.getPremier() + " " + p.getSecond());
  }
- Paire<?,?> paireChiffre = **new** Paire<>(1, "2");
  Paire.*affiche*(paireChiffre);
- Mais ensuite tout n'est plus possible !
  paireChiffre.setPremier(12)

The method setPremier(capture#3-of ?) in the type Paire<capture#3-of ?,capture#4-of ?> is not applicable for the arguments (int)

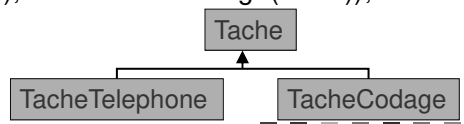
Contrainte extends (1/2)

- Permet d'élargir le cadre d'utilisation des méthodes.
- Exemple

Interface Collection<E>
↳ public boolean addAll(Collection<? extends E> c);

```
List<Tache> taches = new ArrayList<>();
List<TacheTelephone> appels = Arrays.asList(
      new TacheTelephone("Eric", "02 11 22 33 44"),
      new TacheTelephone("Marc", "04 33 99 88 77")
);
List<TacheCodage> codages = Arrays.asList(
      new TacheCodage("bdd"), new TacheCodage("ihm"));

taches.addAll(appels);
taches.addAll(codages);
```



Contrainte extends (2/2)

- Ajoutons dans la classe Paire les méthodes suivantes.
  public <X **extends** T> **void** prendListePremier(List<X> c){
  setPremier(c.get(0));}
  public void prendListeDeuxieme(List<? **extends** U> c){
  setSecond(c.get(1));}
- Ces deux méthodes sont équivalentes

```
List liste = Arrays.asList(1,2,3,4);
Paire<?,?> paireChiffre = new Paire<>();
paireChiffre.prendListePremier(liste);
paireChiffre.prendListeDeuxieme(liste);
Paire.affiche(paireChiffre);
```

1 2

- A noter l'abréviation

<? extends Object> ⇔ **<?>**

  - Exemple : Collection<?>
    est le raccourci de Collection<? extends Object>

Contrainte super

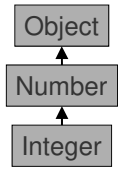
- Dans certains cas on souhaite borner inférieurement un type.
- Exemple de Jokers avec super :

```
List<Object> objets = Arrays.<Object>asList(2, 3.14, "quatre");
List<Number> nombres = Arrays.<Number>asList(1, 2, 3.14);
List<Integer> entiers = Arrays.asList(1,2);

List<? super Number> nombresOuObjets;

nombresOuObjets = objets;
assert nombresOuObjets.toString().equals("[2, 3.14, quatre]");

nombresOuObjets = nombres;
assert nombresOuObjets.toString().equals("[1, 2, 3.14]");
```




```
    nombresOuObjets = entiers;
Type mismatch: cannot convert from List<Integer> to List<? super Number>
}
```


Restriction sur les jokers (1/2)

- Exemple : solutions pour qu'une collection :
  - accepte les éléments d'un type donné
    Collection**<T>** c;
  - accepte les éléments dont le type est :
    - sous-type d'un type donné
      Collection**<? extends T>** c;
    - sur-type d'un type donné
      Collection**<? super T>** c;
- Les jokers sont interdits :
  - Au niveau supérieur des expressions de création d'une classe
    - Exemple incorrect :
      List<?> liste = new ArrayList<?>();
    - Exemple correct :
      List<Number> nombres = new ArrayList<Number>();
      List<? super Number> dest = nombres;
      List<? extends Number> src = nombres;

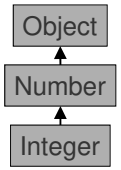
Restriction sur les jokers (2/2)

- Méthode copy de la classe Collections :

```
public static <T> void copy(List<? super T> dst, List<? extends T> src){
    for (int i=0; i<src.size(); i++) {
        dest.set(i, src.get(i));
}   }
```

- On garde les mêmes listes que précédement :

```
List<Object> objets = Arrays.<Object>asList(2, 3.14, "quatre");
List<Number> nombres = Arrays.<Number>asList(1, 2, 3.14);
List<Integer> entiers = Arrays.asList(1,2);
```



- On teste avec :
  - List<Object> listeDest = Arrays.<Object>*asList("un","deux","trois");*
  - Collections.*copy(listeDest, objets);* `[2, 3.14, quatre]`
  - Collections.*copy(listeDest, nombres);* `[1, 2, 3.14]`
  - Collections.*copy(listeDest, entiers);* `[1, 2, trois]`

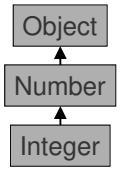
Restriction sur les jokers (2/2)

- Méthode copy de la classe Collections :

```
public static <T> void copy(List<? super T> dst, List<? extends T> src){
    for (int i=0; i<src.size(); i++) {
        dest.set(i, src.get(i));
}   }
```

- On garde les mêmes listes que précédement :

```
List<Object> objets = Arrays.<Object>asList(2, 3.14, "quatre");
List<Number> nombres = Arrays.<Number>asList(1, 2, 3.14);
List<Integer> entiers = Arrays.asList(1,2);
```



- On teste avec :
  - List<Number> listeDest = Arrays.<Number>*asList("un","deux","trois");*
  - Collections.*copy(listeDest, entiers);* `[1, 2, trois]`
  - Collections.*copy(listeDest, nombres);* `[1, 2, 3.14]`
  - `Collections.copy(nombres, objets);`

```
The method copy(List<? super T>, List<? extends T>) in the type Collections is not applicable for the arguments (List<Number>, List<Object>)
```

Les exceptions

- Clause finally
- La gestion des ressources
- Le multi catch
- Le rethrow

Clause finally

- Le mot clé **finally**, généralement associé à un **try**, permet l'exécution du code situé dans son bloc et ceci quelle que soit la manière dont s'est déroulée l'exécution du bloc try.
- Exemple

```
public static void main(String[] args) {
    int valeur = 10;
    int part = 0;
    try {
        int erreur = valeur / part;
        System.out.println(erreur);
    }
    catch (ArithmeticException aE){
        System.out.println("Une exception a été levée");
    }
    finally {
        System.out.println("La valeur est : "+ valeur+", le nombre de part est : "+part);
    }
}
```

```
Une exception a été levée
La valeur est : 10, le nombre de part est : 0
```


Mise en garde sur le bloc finally

- Bonne pratique : évitez d'employer des instructions de rupture de séquence telles que *break*, *continue* ou *return* à l'intérieur d'un bloc try.
  Si c'est inévitable, assurez-vous qu'aucune clause **finally** ne modifie la valeur de retour de votre méthode.


```
public class ReturnFinally {

    public int methode1(){
        try{
            return 1;
        }catch(Exception e){
            return 2;
        }
    }

    public int methode2(){
        try{
            return 3;
        }finally{
            return 4;
        }
    }

    public static void main(String[] args) {
        ReturnFinally rf=new ReturnFinally();
        System.out.println("methode1 renvoie : "+rf.methode1());
        System.out.println("methode2 renvoie : "+rf.methode2());
    }
}
```

```
methode1 renvoie : 1
methode2 renvoie : 4
```

Les exceptions & les entrées/sorties

- Bonne pratique : Concernant les entrées/sorties, utiliser le pattern suivant.

```
try{
        //déclaration de la ressource
        try{
                //utilisation de la ressource
        }finally{
                //fermeture de la ressource
        }
}catch(ExceptionEntreeSortie ex){
        //traitement de l'exception
}
```


Les exceptions : libération de ressources (1/4)

- Il existe plusieurs types de ressources qui doivent être libérées explicitement, (appel à la méthode **close()**).
  - toutes les ressources gérées par le système d'exploitation (fichiers, sockets),
  - et assimilés (connexion JDBC).

- Res r = ... // 1. Création de la ressource
  try {
  // 2. Utilisation de la ressource
  ...
  } finally {
  // 3. Fermeture de la ressource
  r.close();
  }

Les exceptions : libération de ressources (2/4)

- Problème : pollution du code
  - chaque ressource doit être associée à un bloc **try**/**finally** -> plusieurs blocs d'indentation.
  - on doit utiliser un bloc **try**/**catch** supplémentaire pour un traitement des erreurs efficace, sous peine de ne pas intercepter toutes les exceptions...

```
try {
    InputStream input = new FileInputStream(in.txt);
    try {
        OutputStream output = new FileOutputStream(out.txt);
        try {
            byte[] buf = new byte[8192];
            int len;

            while ( (len=input.read(buf)) >=0 )
                output.write(buf, 0, len);
        } finally {
            output.close();
        }
    } finally {
        input.close();
    }
} catch (IOException e) {
    System.err.println("Une erreur est survenue lors de la copie");
    e.printStrackTrace();
}
```


Les exceptions : libération de ressources (3/4)

- **Java 7** :
- Le **try-with-resources** vient pallier tous ces problèmes via une nouvelle syntaxe plus simple. Les ressources déclarées dans un **try()** seront automatiquement libérées à la fin du bloc correspondant, quoi qu'il arrive.
- Le code précédent en **Java 7** :

```
try (InputStream input = new FileInputStream(in.txt);
  OutputStream output = new FileOutputStream(out.txt)) {
    byte[] buf = new byte[8192];
    int len;

    while ( (len=input.read(buf)) >=0 )
        output.write(buf, 0, len);
} catch (IOException e) {
    System.err.println("Une erreur est survenue lors de la copie");
    e.printStrackTrace();
}
```

Les exceptions : libération de ressources (4/4)

- **Java 7** :
- introduit un nouveau concept, les "Suppressed Exceptions :
  - gère proprement les multiples exceptions qui peuvent survenir lors de la fermeture des flux.
  - Si ces dernières surviennent alors qu'une exception est déjà en train de remonter, elles seront "ajoutées" à l'exception originale via **addSuppressed()** au lieu de la remplacer, ce qui permet d'éviter de perdre de l'information (elles seront bien visibles dans le stacktrace).
- Le **try-with-ressources** ne peut être utilisé qu'avec des instances d'objets implémentant la nouvelle interface **java.lang.AutoCloseable**. Cette dernière se contente de définir la méthode **close() throws Exception** qui sera utilisée pour libérer la ressource.

Les exceptions : MultiCatch

- **Java 5/6 :**

```
try {
    ...
} catch(IOException e) {
    // traitement
} catch(SQLException e) {
    // traitement
}
```

- **Java 7** : Si les traitements sont identiques, le code ci-dessus peut désormais s'écrire avec un seul et unique bloc catch

```
try {
    ...
} catch(IOException | SQLException e) {
    // traitement
}
```


Les exceptions : le rethrow (1/2)

- Définition : consiste à remonter une exception après l'avoir catchée

```
public void method() throws Exception {
try {
        // code qui remonte uniquement des IOException ou SQLException...
    } catch (Exception e) {
        // traitement
        throw e; // throws Exception
    }
}
```

- **Java** utilise un mécanisme de vérification des exceptions (les checked-exceptions).
  - Le code ci-dessus déclarera remonter n'importe quelle **Exception**,
  - En réalité il ne peut s'agir que :
    - d'une **IOException**, une **SQLException**,
    - de n'importe quelle "unchecked-exception" (**RuntimeException** ou **Error**, qui peuvent toujours être remontées sans avoir à être déclarées).

Les exceptions : le rethrow (2/2)

- **Java 7** : le **rethrow** a été affiné afin de mieux correspondre à la réalité :
  - le type d'exception remontée dépend également du type d'exception pouvant être remontée par le bloc **try**,
  - la clause **throws** est plus précise

```
public void method() throws IOException, SQLException {
try {
        // code qui remonte uniquement des IOException ou SQLException...
    } catch (Exception e) {
        // traitement
        throw e; // throws IOException, SQLException (ou une unchecked exception)
    }
}
```

- Le **rethrow** ne peut fonctionner que si la variable "e" du **catch** n'est pas modifiée dans le bloc **catch**. Elle doit être implicitement **final**.

Introduction JAVA 8

- Possibilité de donner un corps de méthode aux interfaces
- Implémentation des « traits »
- Prise en compte du langage fonctionnel lambda *Project Lambda* (le package java.util.function définit une quarantaine d'interfaces fonctionnelles).
- …

Interfaces (1/2)

- La syntaxe est simple :
  - fournir un corps à la méthode,
  - la qualifier avec le mot-clé default.

```
public interface Itf {

    /** Pas d'implémentation - comme en Java 7 et antérieur */
    public void foo();

    /** Implémentation par défaut, qu'on surchargera dans la classe fille */
    public default void bar() {
        System.out.println("Itf -> bar() [default]");
    }

    /** Implémentation par défaut, non surchargée dans la classe fille */
    public default void baz() {
        System.out.println("Itf -> baz() [default]");
    }

}
```


Interfaces (2/2)

```
public class Cls implements Itf {

    @Override
    public void foo() {
        System.out.println("Cls -> foo()");
    }

    @Override
    public void bar() {
        System.out.println("Cls -> bar()");
    }

    /* NON SURCHARGE
    @Override
    public void baz() {
        System.out.println("Cls -> baz()");
    }*/

}
```

```
public class Test {
    public static void main(String[] args) {
        Cls cls = new Cls();
        cls.foo();
        cls.bar();
        cls.baz();
    }
}
```

```
Cls -> foo()
Cls -> bar()
Itf -> baz() [default]
```


Héritage multiple (1/2)

- Nous pouvons donc utiliser les interfaces pour implémenter l'héritage multiple en JAVA.

```
interface I1 {
  default int m1() {
    return 1;
  }
}
interface I2 {
  default int m2() {
    return 2;
  }
}
```

```
class C implements I1,I2 {
}
```

Héritage multiple (2/2)

- En cas d'ambiguïté la méthode doit être redéfinie et peut faire appel à l'une ou l'autre des super méthodes héritées.

```
interface I1 {
  default int m() {
    return 1;
  }
}
interface I2 {
  default int m() {
    return 2;
  }
}
```

```
class C implements I1,I2 {
    public int m() {
        return I1.super.m(); }
}
```


Les Traits (1/4)

- Un "trait", ou "extension" : encapsule un ensemble cohérent de méthodes à caractère transverse et réutilisable.
- En général, un trait est composé de :
  - une méthode abstraite qui fait le lien avec la classe sur laquelle il est appliqué,
  - un certain nombre de méthodes additionnelles, dont l'implémentation est fournie par le trait lui-même car elles sont directement dérivables du comportement de la méthode abstraite.

Les Traits (2/4)

- ***Exemple : Comparable***
- La méthode *compareTo()* renvoie un *int*, ce qui n'est pas très... sémantique. Des méthodes comme *greaterThan()* / *lessThan()* ou *isBefore()* / *isAfter()*, renvoyant des booléens, seraient plus parlantes et sont directement dérivées de *compareTo()*.
- Comme l'interface *Comparable* appartient au JDK, nous ne pouvons pas la modifier, mais il est toujours possible de l'étendre.
  Notre interface s'appellera *Orderable* et ne contiendra que des méthodes par défaut s'appuyant sur la méthode *compareTo()* héritée de *Comparable*.

Les Traits (3/4)

```
public interface Orderable<T> extends Comparable<T> {

    // La méthode compareTo() est définie
    // dans la super-interface Comparable

    public default boolean isAfter(T other) {
        return compareTo(other) > 0;
    }

    public default boolean isBefore(T other) {
        return compareTo(other) < 0;
    }

    public default boolean isSameAs(T other) {
        return compareTo(other) == 0;
    }

}
```

Les Traits (4/4)

```
public class Person implements Orderable<Person> {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public int compareTo(Person other) {
        return name.compareTo(other.name);
    }
}
```

```
Laurel compareto Hardy : 4
Laurel >  Hardy : true
Laurel <  Hardy : false
Laurel == Hardy : false
```

```
public class Test {
    public static void main(String[] args) {
        Person laurel = new Person("Laurel");
        Person hardy = new Person("Hardy");
        System.out.println("Laurel compareto Hardy : " + laurel.compareTo(hardy));
        System.out.println("Laurel >  Hardy : " + laurel.isAfter(hardy));
        System.out.println("Laurel <  Hardy : " + laurel.isBefore(hardy));
        System.out.println("Laurel == Hardy : " + laurel.isSameAs(hardy));
    }
}
```


Expressions lambda

- Depuis Java 1.1, la solution pour passer des traitements en paramètres d'une méthode est d'utiliser les classes anonymes internes.

```
NavigableSet<Gaulois> gauloisOrdreImpose =
        new TreeSet<>(new Comparator<Gaulois>() {
                public int compare(Gaulois gaulois1, Gaulois gaulois2) {
                        Integer ageGaulois1 = gaulois1.getAge();
                        Integer ageGaulois2 = gaulois2.getAge();
                        return ageGaulois1.compareTo(ageGaulois2);
                }
        });
```

Source : https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm

Expressions lambda

- Pour faciliter cette mise en œuvre, Java 8 propose des closures ou fonctions anonymes appelées expressions Lambda

```
NavigableSet<Gaulois> gauloisOrdreImposeFn =
  new TreeSet<>(
    (gaulois1, gaulois2) -> gaulois1.getAge() - gaulois2.getAge()
);
```

Source : https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm

Syntaxe des Lambda Expressions

- Une lambda expression consiste en :
  - Une liste de paramètres entre parenthèses
  - Une flèche
  - Le corps de la fonction, qui peut être :
    - Une expression unique
    - Un bloc d'instructions

```
(gaulois1, gaulois2) -> gaulois1.getAge() - gaulois2.getAge();
```

- Une lambda expression est une implantation d'une **interface fonctionnelle**

Source : https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax

Interface fonctionnelle

- C'est une interface pour laquelle il n'y a qu'une méthode à implanter

Interface Comparator<T>

int	**compare**(T o1, T o2) Compares its two arguments for order.

Interface Comparable<T>

int	**compareTo**(T o) Compares this object with the specified object for order.

Interface Iterable<T>

Iterator<T>	**iterator**() Returns an iterator over elements of type T.

Interface fonctionnelle

- C'est une interface pour laquelle il n'y a qu'une méthode à implanter

```
NavigableSet<Gaulois> gauloisOrdreImpose =
        new TreeSet<>(new Comparator<Gaulois>() {
                public int compare(Gaulois gaulois1, Gaulois gaulois2) {
                        Integer ageGaulois1 = gaulois1.getAge();
                        Integer ageGaulois2 = gaulois2.getAge();
                        return ageGaulois1.compareTo(ageGaulois2);
                }
        });
```

⇕

```
NavigableSet<Gaulois> gauloisOrdreImposeFn =
  new TreeSet<>(
    (gaulois1, gaulois2) -> gaulois1.getAge() - gaulois2.getAge()
);
```

- Nommage implicite (ou inutile) !
- Inférence du type !

Précisions sur la syntaxe

- Paramètres :
  - Pas de paramètre -> ()
  - Un seul paramètre -> parenthèses inutiles
  - Typage implicite
  - Typage explicite -> parenthèses obligatoires
- Corps de la lambda :
  - Toute fonction retourne une valeur, éventuellement void
  - Une suite d'instructions est encadrée d'accolades
  - L'instruction return peut être utilisée pour définir le résultat

Source : https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax

Précisions sur la syntaxe : exemples

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

IntegerMath addition = (a, b) -> a + b;
IntegerMath additionRet = (a,b) -> {return (a + b);};

System.out.println(addition.operation(1, 2));
System.out.println(additionRet.operation(2, 3));
```

```
3
5
```

```
Runnable task = () -> System.out.println("Hello");
task.run();
```

```
Hello
```

```
Predicate<Person> pred =
        p -> p.getGender() == Sex.MALE
           && p.getAge() >= 18
           && p.getAge() <= 25;
```

Source : https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax

Précisions sur l'inférence de type

```
Runnable task = () -> System.out.println("Hello");
```

- Remarques :
  - L'objet *task* est typé
  - Le nom de la méthode n'est pas précisé
    - Interface fonctionnelle -> 1 seule méthode
  - L'objet *task* est manipulé comme :
    - Un objet classique
    - Avec son type déclaré (*Runnable*)

```
task.run();
```

Source : https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax

Références : lambda <-> méthodes (1/2)

- Les références de méthodes :
  - syntaxe simplifiée pour référencer une méthode comme une expression lambda.
  - utilisent un opérateur ::
  - concernent les constructeurs et tous types de méthodes
    - Référence à une méthode de classe
    - Référence à une méthode d'instance
    - Référence à un constructeur

Source : https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm

Références : lambda <-> méthodes (2/2)

- Pour les exemples suivants nous avons besoin des méthodes ci-dessous.

Class String

`String`	**`substring(int beginIndex, int endIndex)`** Returns a string that is a substring of this string.
`int`	**`compareToIgnoreCase(String str)`** Compares two strings lexicographically, ignoring case differences.

Class Integer

`static int`	**`parseInt(String s)`** Parses the string argument as a signed decimal integer.

Class Arrays

`static <T> void`	**`sort(T[] a, Comparator<? super T> c)`** Sorts the specified array of objects according to the order induced by the specified comparator.

Références : lambda <-> méthodes (2/2)

```
public class MethodReferences {
  public static void main(String[] args) {
    String[] personnes = {  new String("nom3"),
                            new String("nom001"),
                            new String("nom20") };
    Arrays.sort(personnes, (st1, st2) -> st1.compareTo(st2));
    System.out.println(Arrays.toString(personnes));

}
```

```
[nom001, nom20, nom3]
```

Class Arrays

`static <T> void`	**`sort(T[] a, Comparator<? super T> c)`** Sorts the specified array of objects according to the order induced by the specified comparator.

Références : lambda <-> méthodes (2/2)

```
public class MethodReferences {
   static class ComparaisonString {
      public int comparerParLongueur(String p1, String p2) {
         return p1.length() - p2.length();
      }
   }
   public static void main(String[] args) {
   String[] personnes = {   new String("nom3"),
                            new String("nom001"),
                            new String("nom20") };
   ComparaisonString comparaisonChaine = new ComparaisonString();
   Arrays.sort(personnes, comparaisonChaine::comparerParLongueur);
   System.out.println(Arrays.toString(personnes));
}
```

```
[nom001, nom20, nom3]
```


Références : lambda <-> méthodes (2/2)

```
public class MethodReferences {
  public static void main(String[] args) {
    String[] personnes = {   new String("nom3"),
                             new String("nom001"),
                             new String("nom20") };
    Arrays.sort(personnes, String::compareToIgnoreCase);
    System.out.println(Arrays.toString(personnes));
  }
}
```

```
[nom001, nom20, nom3]
```

Class String

`int`	`compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.

Références : lambda <-> méthodes (2/2)

```
public class MethodReferences {
  static int compareParSuffixe(String chaine1, String chaine2) {
    return (valFrom(chaine1) - valFrom(chaine2));
  }
  public static void main(String[] args) {
    String[] personnes = {       new String("nom3"),
                                 new String("nom001"),
                                 new String("nom20") };
    Arrays.sort(personnes, MethodReferences::compareParSuffixe);
    System.out.println(Arrays.toString(personnes));
  }
  private static int valFrom(String chaine1) {
    return Integer.parseInt(chaine1.substring(3));
  }
}
```

```
[nom001, nom20, nom3]
```


Les interfaces fonctionnelles de java

API java.util.function

- Function<T, R> R apply(T t);
- Consumer<T> void accept(T t);
- Predicate<T> boolean test(T t);
- Supplier<T> T get();

- BiFunction<T,U,R>
- BiConsumer<T,U>
- BiPredicate<T,U>

- XXX{Function/Consumer/Predicate/Supplier}
  - Pour XXX = Int, Double, Long

Fermetures

- Variable libre : variable qui n'est ni un paramètre ni déclarée localement

```
IntFunction<Integer> f1Avec_iLibre = x -> x + 2 + i;
```

- Il faut éviter d'utiliser une variable libre dans le corps d'une lambda
- Si c'est inévitable,
  - La variable est considérée comme *final*
  - Et ne doit pas être modifiée

```
Local variable i defined in an enclosing scope must be final or effectively final
```

- Ceci est (évidemment !) le cas pour les références d'objets
  - L'état de l'objet peut être modifié
  - Mais la référence ne peut l'être

Streams et Pipelines

- Avec la version 8, Java introduit les notions de :
  - Pipeline, ayant
    - une source (collection, array, générateur, flux I/O)
    - une séquence d'opérations d'aggregation
    - une opération terminale
      - produisant une valeur primitive, une collection ou pas de valeur du tout
  - Stream
    - séquence d'éléments (≠ collection, pas de stockage)

https://docs.oracle.com/javase/tutorial/collections/streams/index.html

Streams et Pipelines : exemples

```
Person[] roster = { new Person("Paul", LocalDate.of(1998, 6, 23),
                    Sex.MALE, "paul.luap@me.com"),
                    new Person("Marie", LocalDate.of(2005, 3, 12),
                    Sex.FEMALE, "marie.eiram@me.com"),
                    new Person("Pierre", LocalDate.of(2010, 8, 15),
                    Sex.MALE, "pierre.erreip@me.com") };
for (Person p : roster) {
        System.out.println(p.getName());
}
```

```
Paul
Marie
Pierre
```

```
Arrays.stream(roster).forEach(e -> System.out.println(e.getName()));
```

```
double average =
        Arrays.stream(roster)
            .filter(p -> p.getGender() == Sex.MALE)
            .mapToInt(Person::getAge)
            .average()
            .getAsDouble();
```

```
Paul
Marie
Pierre
```

```
moyenne : 12.666666666666666
```

Mignardises ☺

```
public static void main(String[] args) {
        final int nbOfItems = 5;
        List<String> messagesList = new ArrayList<>(nbOfItems);

        for(int i = 0; i < nbOfItems; i++)
                messagesList.add("Hello R2D"+i);

        messagesList.stream()
                .forEach(System.out::println);

        messagesList.parallelStream()
                .forEach(System.out::println);
}
```

```
Hello R2D0
Hello R2D1
Hello R2D2
Hello R2D3
Hello R2D4
```

```
Hello R2D2
Hello R2D1
Hello R2D0
Hello R2D3
Hello R2D4
```

```
Hello R2D2
Hello R2D4
Hello R2D0
Hello R2D3
Hello R2D1
```