Fonctions anonymes en Java
(mini programmation fonctionnelle)

Problématique

- Imaginons que l'on désire écrire une méthode **sortInv** qui trie, en ordre lexicographique inverse, un tableau de chaînes de caractères qu'on lui passe en paramètre

```
public static void sortInv(String[] tab){
 // ... code
 }
```

- Trier le tableau par ordre naturel puis inverser l'ordre de ses éléments n'est pas une bonne solution, car cela nécessite un parcours supplémentaire
- Re-écrire le tri pour trier en ordre inverse… ok, mais cela nécessite de maintenir artificiellement un code particulier. Et si on souhaite un nouveau tri, il faut revoir aussi la version « inverse ». Bref, pas pratique.

Solution du JDK

- Une bien meilleure solution consiste à utiliser la méthode sort de Arrays. en lui passant un « comparateur » et qui inverse l'ordre naturel des chaînes dans la comparaison :

```
public static <T> void sort(T[] a, int fromIndex, int toIndex,
                            Comparator<? super T> c)
```

- <T>, < ? super T> ===> ????
- On verra cela plus tard (c’est un peu compliqué, plutôt L2 voir Ing1). Voyons ce que nous pouvons faire simplement...

Solution 1

- Utilisation d'une classe séparée, privée (éventuellement) et imbriquée statiquement

```
private static class InvComparator implements Comparator<String> {
     @Override
    public int compare(String s1, String s2) {
         // Le code nécessaire
     }
}
```

- Puis :

```
Arrays.sort(tab,0,tab.length,new InvComparator())
```

- Clairement lourd et avec des éléments qui ne seront vus que l'année prochaine…

Solution 2

- Pour simplifier ce code, on peut bien entendu utiliser une classe anonyme

```
Arrays.sort(tab,0,n, new Comparator<String>(){
                @Override
                public int compare(String s1, String s2) {
                    return s2.compareTo(s1);
                }
              });
```

- Toujours lourd (mais moins)

Solution 3

- Utilisation d’une fonction anonyme (lambda expression en hommage au lambda-calcul inventé dans les années 1930 et toujours étudié d’ailleurs ou aussi appelé fermeture/closure)

```
Arrays.sort(tab,0,n,
            (String s1, String s2) -> {
                return s2.compareTo(s1);
        });
```

- En fait, on définie une fonction (sans nom) à 2 paramètres (s1 et s2) et qui retourne une comparaison entre ces 2 chaînes.
- Et on peut encore faire mieux

Solution 3 bis

- On peut encore simplifier car:
  - le type des paramètres est optionnel car inféré (trouvé automatiquement par javac) ; Idem pour le retour
  - si le corps de la fonction est composé d'une seule expression, elle peut être écrite telle quelle, sans accolades englobantes ni return.

```
Arrays.sort(tab,0,n,(s1, s2) -> s2.compareTo(s1));
```

- Ce principe peut être utilisé dans bcp d’autres cas. Il nous faut aussi connaître comment déclarer des paramètres qui soient maintenant des fonctions (du calcul) et non plus de simple données.
- On parle de « higher order programming » ou que les fonctions sont des valeurs de première classe (pas complètement vrai en Java mais ce n’est pas le moment d’en discuter)
- Nous allons devoir écrire des interfaces. Cette notion sera détaillée en L2.

Interface fonctionnelle

- Nous devons en Java décrire qu'elle est le type de la fonction qui sera attendu en paramètre de la méthode (une fonction de fonction) :
- Exemple, pour représenter une fonction mathématique de ℝ dans ℝ :

```
public interface RealFunction {
    public double valueAt(double x);
}
```

- Pour la comparaison (tri) :

```
Interface Comparator<T> {
   int    compare(T o1, T o2)
   // Bcp d'autres choses }
```

- Pour ce cours, une interface sera dans la classe (public class truc{…})
- Voir https://docs.oracle.com/javase/10/docs/api/java/util/Comparator.html

Fonction anonyme

- En Java, une fonction anonyme (anonymous function) est donc une expression créant une instance d'une classe anonyme qui implémente une interface fonctionnelle (interface avec une seule méthode abstraite)
- La fonction anonyme spécifie uniquement les arguments et le corps de l'unique méthode abstraite de l'interface fonctionnelle. Ceux-ci sont séparés par une flèche symbolisée par la chaîne -> :

arguments -> corps

Attention (en L2)

- Une fonction anonyme ne peut apparaître que dans un contexte qui attend une valeur dont le type est une interface fonctionnelle.
- Exemple valide :

```
Comparator<Integer> c = (x, y) -> x.compareTo(y);
```

- Car Comparator est une interface fonctionnelle.
- Exemple Invalide :

```
Object c = (x, y) -> x.compareTo(y);
```

- Car Object n'est pas une interface fonctionnelle. Java ne peut donc savoir à quelle méthode doit correspondre le code de la fonction anonyme.

Paramètres

- Dans leur forme générale, les paramètres d'une fonction anonyme sont entourés de parenthèses et leur type est spécifié avant leur nom, comme d'habitude.
- Par exemple, une fonction anonyme à deux arguments, le premier de type Integer et le second de type String, peut s'écrire ainsi :

```
(Integer x, String y) -> // … corps
```

- Cette notation peut être allégée de deux manières :
  - 1. le type des arguments peut généralement être omis, car inféré par le compilateur,
  - 2. lorsque la fonction ne prend qu'un seul paramètre, les parenthèses peuvent être omises.
- Pour notre tri, nous aurions pu écrire :

```
public interface StringComparator { int  compare(String o1, String o2) ; }
public static void sortInv(String[] tab,  StringComparator cmp)
...
sortInv(mytab,(String s1,String s2)-> {...//Code de comparaison}) ;
```

Fonctions-blocs

- Dans sa forme la plus générale, le corps d'une fonction anonyme est constitué d'un bloc entouré d'accolades. Comme toujours, si la fonction anonyme retourne une valeur — c'est-à-dire que son type de retour est autre chose que void — celle-ci l'est via l'énoncé return.
- Par exemple, le comparateur ci-dessous compare deux chaînes d'abord par longueur puis par ordre alphabétique :

```
Comparator<String> c = (s1, s2) -> {
    int lc = Integer.compare(s1.length(), s2.length());
   //return lc != 0 ? lc : s1.compareTo(s2);
   If (lc!=0) return lc else return s1.compareTo(s2);
};
```

- Si le corps de la fonction anonyme est constitué d'une seule expression (comme dans notre cas de Comparator), alors on peut l'utiliser en lieu et place du bloc. Cela implique de supprimer les accolades englobantes et l'énoncé return.

Exercice

- Programmez une méthode

```
public static void Map(int[] tab, Function f)
```

- Et qui applique (apply) la fonction « f » a chaque élément du tableau. Pour ce faire on aura besoin de l’interface

```
public interface Function { int    apply(int x); }
```

- Testez avec

java.util.function

- On l'a vu, les fonctions anonymes ne peuvent être utilisées que dans un contexte qui attend une valeur dont le type est une interface fonctionnelle. Dès lors, il est utile d'avoir à disposition un certain nombre de telles interfaces, couvrant les principaux cas d'utilisation.
- Le paquetage java.util.function a pour but de définir un ensemble d'interfaces fonctionnelles, et les principales d'entre-elles sont présentées ci-après.
- L'interface `Function` représente une fonction à un argument. Le type de cet argument et le type de retour de la fonction sont les paramètres de type de cette interface, nommés respectivement T et R.

```
public interface Function<T, R> { public R apply(T x); }
```

  Exemple, la fonction pour la longueur d'une chaîne pourrait se définir :

```
Function<String, Integer> stringLength =  s -> s.length();
```

- et s'utiliser de la sorte :

```
stringLength.apply("bonjour"); // → 7
```

Composition de fonctions

- En plus de la méthode abstraite `apply`, l'interface `Function` offre une méthode compose permettant de composer deux fonctions entre elles, au sens mathématique.
- Exemple, deux fonctions sur les entier `f` et `g` et leur composition f ∘ g peuvent se définir ainsi :

```
Function<Integer,Integer> f = x -> 2*x;
Function<Integer,Integer> g = x -> x + 1;
Function<Integer,Integer> fg = f.compose(g);
```

- Et la composition s'utilise comme toute autre fonction :

```
fg.apply(10); // → 22
```

Encore plus conçis !

- Pour simplifier l'écriture de fonctions anonymes (qui se contentent d'appeler une méthode en lui passant les arguments qu'elle a reçus), Java offre la notion de référence de méthode (method reference).
- Exemple :

```
Comparator<String> c = String::compareTo;
```

- Notez que le `String` auquel on applique la méthode `compareTo` devient le premier argument de la fonction anonyme !
- Une référence à une méthode s'obtient simplement en séparant le nom de la classe et celui de la méthode par un double deux-points
- Il est également possible d'obtenir une référence de méthode sur un constructeur, en utilisant le mot-clef `new` en lieu et place du nom de méthode statique. Exemple :

```
Supplier<List> lists = ArrayList::new;
Supplier<List> lists =  () -> new ArrayList();
```

- On peut aussi utiliser des codes comme :

```
Function<Integer, Character> alphabetChar = "abcdef"::charAt;
```

Suite

- Il existe aussi (pour info) :
  - UnaryOperator pour des operateurs unaires (valeur absolue p. ex.)
  - BiFunction pour des fonctions a 2 arguments
  - BinaryOperator
  - Predicate : si la valeur de retour est un boolean (un test)

```
Predicate<Integer> p = x -> x >= 0;
Predicate<Integer> q = x -> x <= 5;
Predicate<Integer> r = p.and(q);
Predicate<Integer> s = r.negate();
```

  - BiPredicate
  - Consumer, supplier etc.
- On peut utiliser des « for each » sur des « flots »

For each ?

- On peut itérer sur un tableau (et sur d’autres choses qu’on verra plus tard) de manière très simple.

```
for (type var : array) {
  // code utilisant var;
}
```

```
for (int i=0; i<arr.length; i++) {
   type var=arr[i] ;
  // code utilisant var ;
}
```

- Exemple :

```
int[] marks = { 125, 132, 95, 116, 110 };
for (int x:marks) { println(x) ; }
```

- Limitations :
  - Il ne faut pas modifier le tableau pendant la boucle
  - On accède pas à l’indice du tableau
  - On ne parcours que d’un en un et dans l’ordre

Flots

- Un flot (stream) est une séquence de valeurs auxquelles on accède l'une après l'autre, de la première à la dernière
- La programmation par flots consiste à exprimer un calcul sur des données sous la forme d'un enchaînement d'opérations simples appliquées au flot de ces données
- Un fichier, un tableau, le réseau, la vidéo, des capteurs sur une voiture peuvent être vus comme des flots (et c’est vraiment utilisé en pratique)
- java.util.stream

Exemple : Conversion °F en °C

- Admettons que l'on désire afficher un tableau contenant des températures en degrés Fahrenheit (sous forme de String) mais avoir ces mêmes températures en degrés Celsius et en Float et en éliminant les températures aberrantes (erreur de mesure) qui sont des String vide
- Par exemple, ["0"," ","100"] ⇒ [-17.777,37.777]
- On pourrait faire :

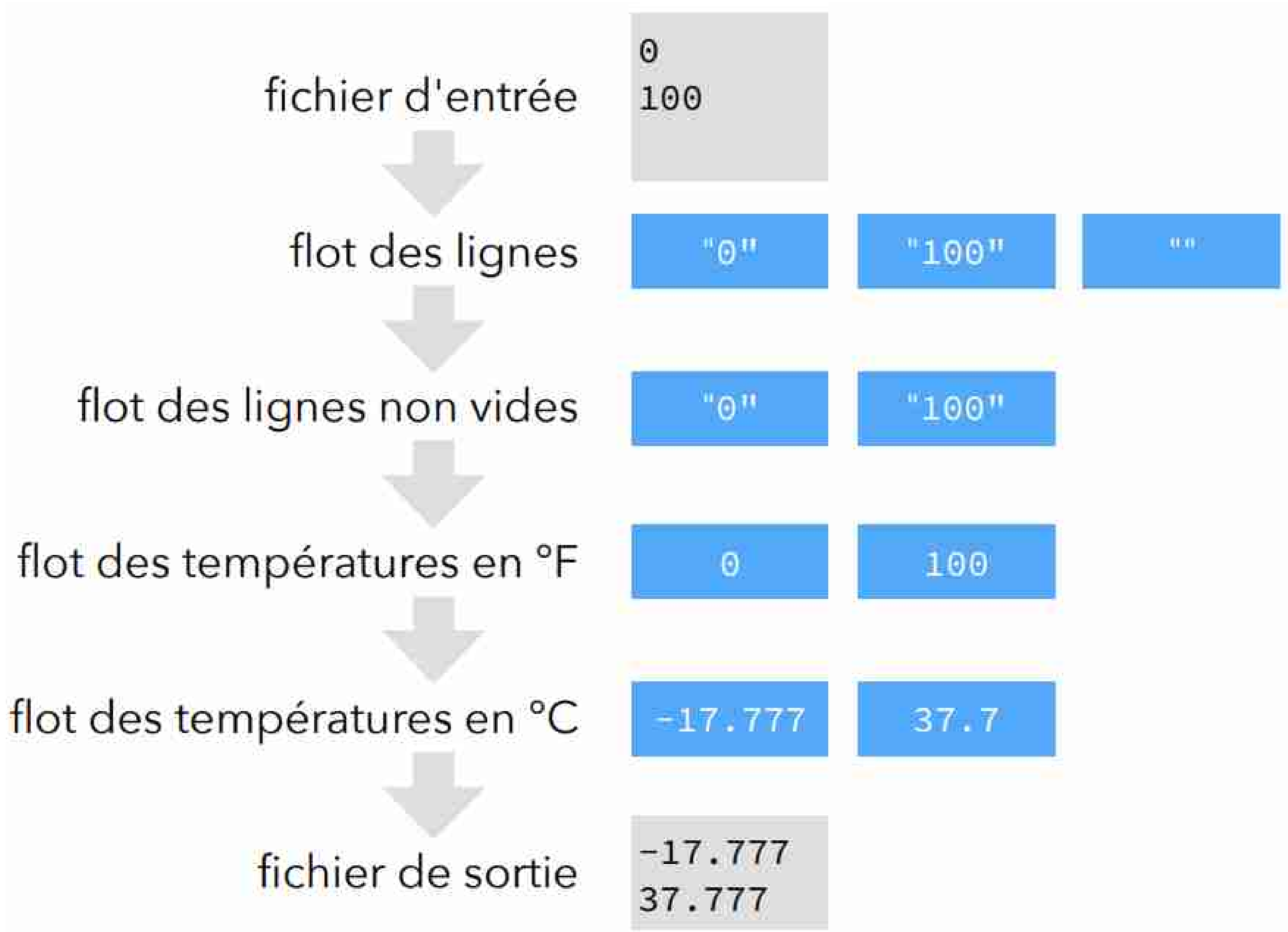
  1) Convertir ce tableau en flot (en pratique, on aurait plutôt directement un flot provenant d'un appareil)

  2) Convertir ce flot de String en un de Floats

  3) Filtrer ce flot pour ne garder que les valeurs utilisables

  4) Convertir avec la formule °C=(°F − 32)×5/9

  5) Afficher


fichier d'entrée
0
100
flot des lignes
"0"
"100"
""
flot des lignes non vides
"0"
"100"
flot des températures en °F
0
100
flot des températures en °C
-17.777
37.7
fichier de sortie
-17.777
37.777

forEach

- La méthode forEach (provenant de Java.Util.Iterable) des Stream prend un consommateur (une fonction anonyme retournant void) en argument et l'applique à chaque élément de l'entité itérable.
- Etant donné que les flots suivent l'interface Iterable, cette méthode est disponible — entre autres —
- Par exemple, pour afficher les éléments d'un flot provenant d'un tableau, on peut écrire :

```
int[] tab={1, 2, 3, 4, 5};
(tab.stream).forEach(x -> {System.out.println(x); });
```

Filtrer des éléments

- La méthode `filter` prend un prédicat (une fonction anonyme retournant un boolean) en argument et supprime tous les éléments de la qui le satisfont. Un nouveau flot est engendré !
- Par exemple, l'extrait de programme ci-dessous supprime tous les nombres pairs puis affiche :

```
(tab.stream).filter(x -> x % 2 == 0)
                    .forEach(x -> {System.out.println(x); });
```

- La méthode `toArray()` permet de transformer un flot en un tableau si besoin

Map et Reduce

- Ici 2 ENORME classiques. Des fonctionnalités très proches (mais dont l’implantation tourne sur les milliers de machines) servent à Google pour le traitement de leurs données
- <R> Stream<R> map(Function<? super T,? extends R> mapper) applique une fonction sur un flot et retourne un nouveau flot.
- reduce(T identity, BinaryOperator<T> accumulator) réduit les éléments (voir tableau)
- Ces concepts seront traités en détail pendant les cours de Calcul Distribué et Big-Data en Ing2 et Ing3

Solution pour les degrés

```
int tabF = {"100","-257",0} ;
tabF.stream.filter(s -> !s.isEmpty())
          .map(Double::parseDouble)
          .map(f -> (f - 32d) * (5d / 9d))
          .forEach(w::println);
```

- Bonus, les programmes Java avec Stream sont capables d’utiliser les capacités multi-coeurs des machines = diviser le travail sur les coeurs

Jeu de la vie

- Implantez le jeu de la vie pour une grille de 30*30
- Faire un affichage de telle sorte qu’a chaque étape on ne voit que la grille
- Faire une boucle infinie...ctrl-c pour forcer l’arrêt d’un programme (Windows, Linux, IOS)
- Testez avec vos propres grilles

Exercice : Automates cellulaires

- Faire une implantation des automates cellulaires dont la fonction de transition est fournit par l'utilisateur
- On supposera que seuls les 2 cases voisines peuvent être prises en comptes
- Faire un affichage « jolie »
- Testez avec une transition proposé par la vidéo
- Testez avec une fonction de votre choix