Programmation 2 : Septième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

21 octobre 2019

Types paramétrés (notions avancées)

Condition sur les paramètres – Problématique

```
public interface Comparable<T> {
  public int compareTo(T element);
}

class Greatest {
  private String element;
  public void add(String element) {
    if (this.element==null ||
        this.element.compareTo(element)<0)
    this.element = element;
  }
  public String get() { return element; }
}
```

Comment rendre la classe `Greatest` générique ?

Condition sur les paramètres

```
class Greatest<T extends Comparable<T>> {
  private T element;
  public void add(T element) {
  if (this.element==null
    || element.compareTo(this.element)>0)
    this.element = element;
  }
  public T get() {
    return element;
  }
}
```

Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}
class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il n'est pas possible d'écrire les lignes suivantes car PrettyCard n'implémente pas l'interface Comparable<PrettyCard> :

```
Greatest<PrettyCard> greatest =
  new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

Syntaxe ? super

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}

class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il est possible d'écrire les lignes suivantes car `PrettyCard` implémente l'interface `Comparable<Card>` et `Card super PrettyCard`:

```
Greatest<PrettyCard> greatest =
  new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

Wildcard ?

Dans un paramètre de généricité, le symbole ? (appelé *wildcard*) dénote une variable de type anonyme.

On peut la contraindre avec les mot-clés `super` et `extends`.

Exemples :

- `List<?>` : une liste de type quelconque.
- `List<? extends Shape>` : une liste d'instances d'une sous-classe de `Shape`.
- `List<? super Disc>` : une liste d'instances d'une classe ancêtre de `Disc`.
- `E extends Comparable<? super E>` : un type `E` implémentant l'interface `Comparable<P>` pour `P` ancêtre de `E`.

? extends – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<T> list) {
    for (T element : list) add(element);
  }
}
```

Il n'est pas possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
/* ... */
list.addAll(list);
```

? extends

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<? extends T> list) {
    for (T element : list) add(element);
  }
}
```

Il est maintenant possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
/* ... */
greatest.addAll(list);
```

Méthodes paramétrées et conditions sur les types

```
class Tools {
  static <T extends Comparable<T>>
  boolean isSorted(T[] array) {
    for (int i = 0; i < array.length-1; i++)
      if (array[i].compareTo(array[i+1]) > 0)
        return false;
    return true;
  }
}
```

Exemple :

```
String[] array = { "ezjf", "aaz", "zz" };
System.out.println(Tools.isSorted(array));
```

Méthodes paramétrées et conditions sur les types

Méthode pour copier une liste `src` vers une autre liste `dest` :

```
static <T> void    copy(List<? super T> dest, List<? extends T> src)
```

On suppose qu'on a une classe `MovingPixel` qui étend `Pixel` qui elle-même étend `Point`.

On peut écrire :

```
List<MovingPixel> src = new ArrayList<>();
List<Point> dest = new ArrayList<>();
Collections.<Pixel>copy(dest, src);
```

Utilisation `extends` et `super`

Lorsqu'on a une collection d'objets de type `T` :

- En entrée/écriture, on veut donner des objets qui ont au moins tous les services des objets de type `T`.

  On doit donc donner des objets dont la classe étend `T` : `? extends T`

- En sortie/lecture, on veut récupérer des objets qui ont au plus tous les services des objets de type `T`.

  On doit donc récupérer des objets qui sont étendu par la classe `T` : `? super T`

Interfaces (notions avancées)

Les classes anonymes

Supposons que nous ayons l'interface suivante :

```
Interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Il est possible de :

- définir une classe anonyme qui implémente cette interface
- d'obtenir immédiatement une instance de cette classe

```
ActionListener listener = new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    counter++;
  }
});
```

Les classes anonymes

```
public class Window {
  private int counter;
  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          counter++;
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des attributs de la classe "externe" :

```
public class Window {
  private Counter counter = new Counter();
  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          counter.count();
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des variables finales de la méthode :

```
public class Window {
  public Window() {
    final Counter counter = new Counter();
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        counter.count();
      }
    };
  }
}
```

Java 8 : Lambda expressions

Avec Java 8, il est possible d'écrire directement :

```
public class Window {
  public Window() {
    Button button = new Button("button");
    button.addActionListener(
      event -> System.out.println(event)
    );
  }
}
```

Explication : ActionListener possède une seule méthode donc on peut affecter une lambda expression à une variable de type ActionListener.

```
public interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Interfaces fonctionnelles

En Java 8, une interface n'ayant qu'une méthode abstraite est une interface fonctionnelle. Les quatre interfaces fonctionnelles suivantes (et plein d'autres) sont déjà définies :

```
public interface Predicate<T> {
  public boolean test(T t);
}
public interface Function<T,R> {
  public R apply(T t);
}
public interface Consumer<T> {
  void accept(T t);
}
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Syntaxe d'une lambda expression

Pour instancier une interface fonctionnelle, on peut utiliser une lambda expression :

L'interface suivante :

```
public interface MyFunctionalInterface{
  public T myMethod(A arg1, B arg2, C arg2);
}
```

peut être instancier par :

```
MyFunctionalInterface fonc =
  (arg1, arg2, arg2)
  ->/* expression définissant le résultat de myMethod */
```

Si `T` est `void` alors l'expression peut être `void` comme un `println`.

Exemples de lambda expression

On considère une classe `Person` avec deux attributs `name` et `age` et les getters et setters associés.

On a le droit d'écrire les lambda expressions suivantes en Java :

- `person -> person.getAge() >= 18` de type `Predicate<Person>`
- `person -> person.getName()` de type `Function<Person,String>`
- `name -> System.out.println(name)` de type `Consumer<Person>`

Remarques

- Il n'est pas nécessaire de mettre le type des paramètres.
- On peut omettre les parenthèses dans le cas où il n'y a qu'un seul paramètre

Référence de méthodes

Dans un certain nombre de cas, une lambda expression se contente d'appeler une méthode ou un constructeur.

Il est plus clair dans ce cas de se référer directement à la méthode ou au constructeur.

Lambda expression	référence de méthode
`x -> Math.sqrt(x)`	`Math::sqrt`
`name -> System.out.println(name)`	`System.out::println`
`person -> person.getName()`	`Person::getName`
`name -> new Person(name)`	`Person::new`

java.util.stream.Stream

`Stream` = Abstraction d'un flux d'éléments sur lequel on veut faire des calculs

Ce n'est pas une `Collection` d'élément car un `Stream` ne contient pas d'élément

Création d'un Stream :

- À partir d'une collection comme une liste avec `list.stream()`
- À partir d'un fichier : `Files.lines(Path path)`
- À partir d'un intervalle : `IntStream.range(int start, int end)`

Exemples d'utilisation

```
persons
.stream()
.filter(person -> person.getAge() >= 18)
.map(person -> person.getName())
.forEach(name -> System.out.println(name));
```

Types des paramètres et retours des méthodes :

- `stream()` → `Stream<Person>`
- `filter(Predicate<Person>)` → `Stream<Person>`
- `map(Function<Person, String>)` → `Stream<String>`
- `forEach(Consumer<String>)`

Cycle de vie d'un `Stream`

Un `Stream` est toujours utilisé en trois phases :

- Création du Stream (à partir d'une collection, d'un fichier, ...),
- Opérations intermédiaires sur le `Stream` (suppression d'éléments, transformation de chaque élément, combinaison) qui prenne un `Stream` et renvoie un `Stream`,
- Une seule opération terminale du `Stream` (calcul de la somme, de la moyenne, application d'une fonction sans retour sur chaque élément, ...).

Opérations intermédiaires possibles sur un `Stream`

- `Stream<E> filter(Predicate<? super E>)` : sélectionne si un élement reste dans le `Stream`
- `<R> Stream<R> map(Function<? super E, ? extends R)` : transforme les éléments du `Stream` en leur appliquant une fonction
- `Stream<E> sorted(Comparator<? super E>)` : trie les éléments

Opérations terminales possibles sur un `Stream`

- `long count()` : compte le nombre d'éléments
- `long sum()` : somme les éléments (entiers ou double)
- `Stream<E> forEach(Consumer<? super E>)` : Appele le `consumer` pour chaque élément
- `allMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour tous les éléments
- `anyMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour au moins un élément
- `collect(Collectors.toList())` : crée une liste avec les éléments du `Stream`