Type Parametré

Rémi Forax
forax@univ-mlv.fr

Méthode paramétrée

Le but d'une méthode paramétrée est d'ajouter des contraintes sur les types des paramètres

Exemple:
une méthode qui remplie les case d'un tableau avec la méthode valeur

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

Problème

Le code de fill peut() planter à l'exécution

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}

public static void main(String[] args) {
  fill(args, 3);
}
```

Pourquoi ?

Les tableaux de référence

Il est possible de faire du sous-typage sur les tableaux !

```
String[] args = new String[1];
Object[] array = args;
```

Affreux car c'est faux en terme de typage

Un tableeau d'objet est un tableau dans lequel on peut mettre ou retirer des objets

=> on peut mettre un Object dans un tableau d'Object

```
array[0] = new Object();
```

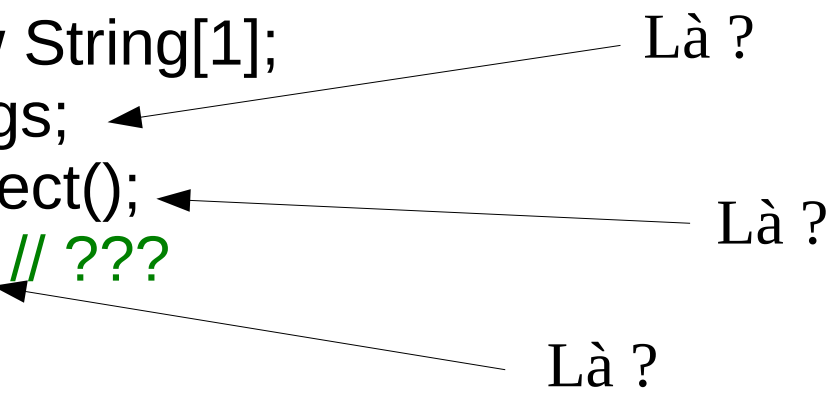
si cela fonctionne alors on peut écrire

```
String s = args[0];  // ???
```

Les tableaux de référence

Où est le problème ?

```
String[] args = new String[1];
Object[] array = args;
array[0] = new Object();
String s = args[0];  // ???
```



- Soit il est interdit de faire du sous-typage sur les tableaux
- Soit il faut vérifier à l'exécution que la référence sur le tableau est bien de la bonne classe par raport à la classe de la référence

Les tableaux de référence en Java

En Java,

- le sous-typage est permis
- la VM vérifie les classes (avec un instanceof) à chaque assignation
  - La VM lève une exception ArrayStoreException

approche valide si

- On lit les valeurs après sous-typage plutôt qu'on les écrits
- La VM à accès au même information que le compilateur à l'exécution

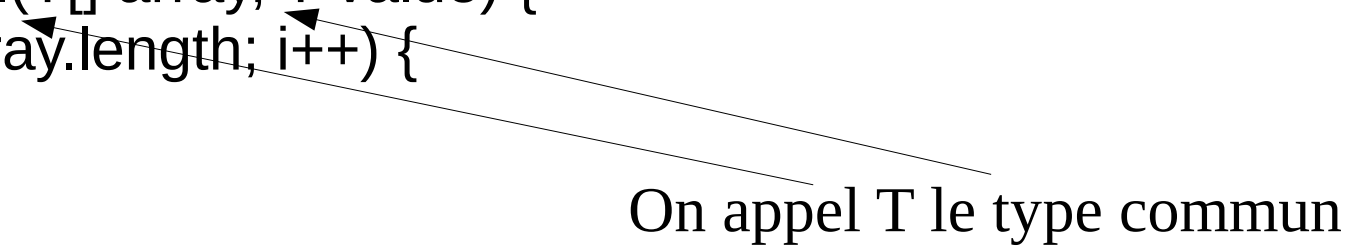
Solution

Au lieu de

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

Indiquer que le type du tableau et le type de value doivent être identiques

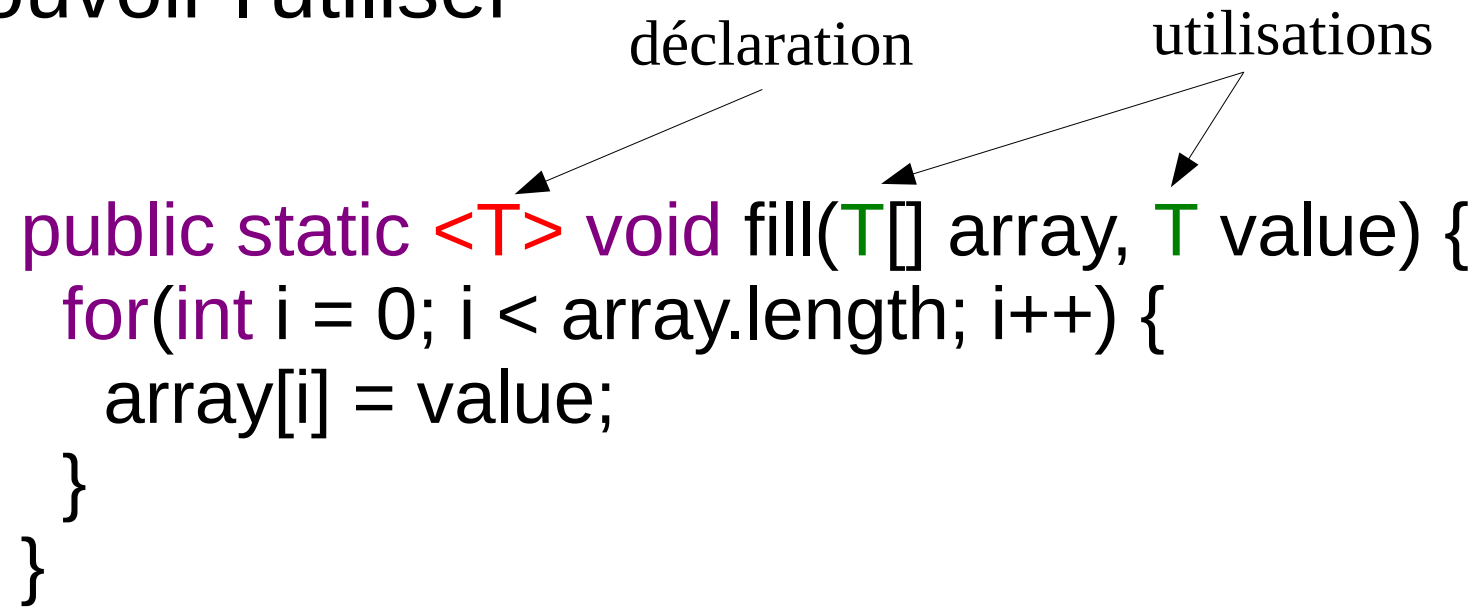
```
public static void fill(T[] array, T value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```



mais ça ne compile pas ?

Déclaration de variable de type

Il faut déclarer une variable de type avant de pouvoir l'utiliser



Une variable de type comme n'importe quelle variable doit être déclarée

(entre les modificateurs et le type de retour)

Solution

Si fill() est paramétrée

```
public static <T> void fill(T[] array, T value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

dans ce cas, notre code fautif ne compile plus

```
public static void main(String[] args) {
  fill(args, 3);   // compile pas !, fill(String[], Integer)
}
```

Variable de type

Introduire une variable de type permet de vérifier des contraintes sur les types

donc de rejeter des codes invalides à la compilation et pas à l'exécution

cela permet aussi de propager des types !

Objects.requireNonNull

```
public class Person {
  private final String name;
  private final Address address;

  public Person(String name, Address address) {
    this.name = Objects.requireNonNull(name);
    this.address = Objects.requireNonNull(address);
  }
}
```

Objects.requireNonNull() doit être paramétrée !

Objects.requireNonNull

```
public class Objects {
  public static <T> T requireNonNull(T value) {
    if (value == null) {
      throw new NullPointerException();
    }
    return value;
  }
}
```

Le type de retour est le type pris en paramètre !

Inférence de type

Dans la majorité des cas, il n'est pas nécessaire de spécifier le type argument pour une variable de type car le compilateur le calcul seul en fonction du context

- Le compilateur utilise pas uniquement le type des arguments
  - Il peut aussi utiliser le type attendu pour calculer le type de retour

Inférence de type

La méthode Collections.emptyList() permet d'obtenir une implantation d'une liste vide **non mutable**

```
public class Collections {
  public static <E> List<E> emptyList() {
    ...
  }
}
```

Inférence de type

```
public class Collections {
  public static <E> List<E> emptyList() { ... }
}
```

- Inférence lors d'une assignation
  - List<String> list = Collections.emptyList(); // ok
- Inférence en fonction du type de retour
  - List<String> foo() {
    return Collections.emptyList(); // ok
    }
- Inférence en fonction du type d'un paramétre
  - void foo(List<String> list) { ... }

    ...

    foo(Collections.emptyList());   // ok, depuis Java 8

Passage de type argument explicite

Dans certain cas, le compilateur n'arrive pas à inférer le type argument (par exemple quand il y a plusieurs possibilités), il faut lui indiquer explicitement

On indique le type argument entre le '.' et le nom de la méthode entre '<' '>'

Exemple

```
Collections.<String>emptyList();
```

Exemple

```
public class Foo {
  public void bar(Collection<String> c) { ... }
  public void bar(List<Integer> l) { ... }
}
...
Foo foo = ...
foo.bar(Collections.emptyList());  // compile pas
foo.bar(Collections.<String>emptyList()); // ok
foo.bar(Collections.<Integer>emptyList()); // ok
```

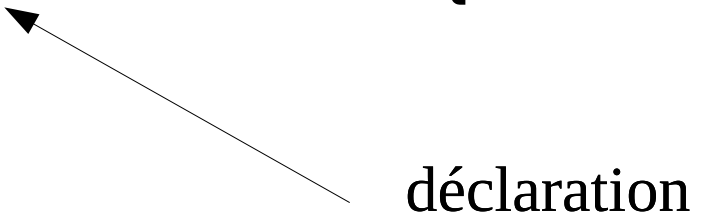
Note:

- Utiliser la surcharge est souvent **pas une bonne idée !**

Type paramétré

Il est possible de déclarer des types paramétrés

- public class ArrayList<E> extends ... {
  ...
  }

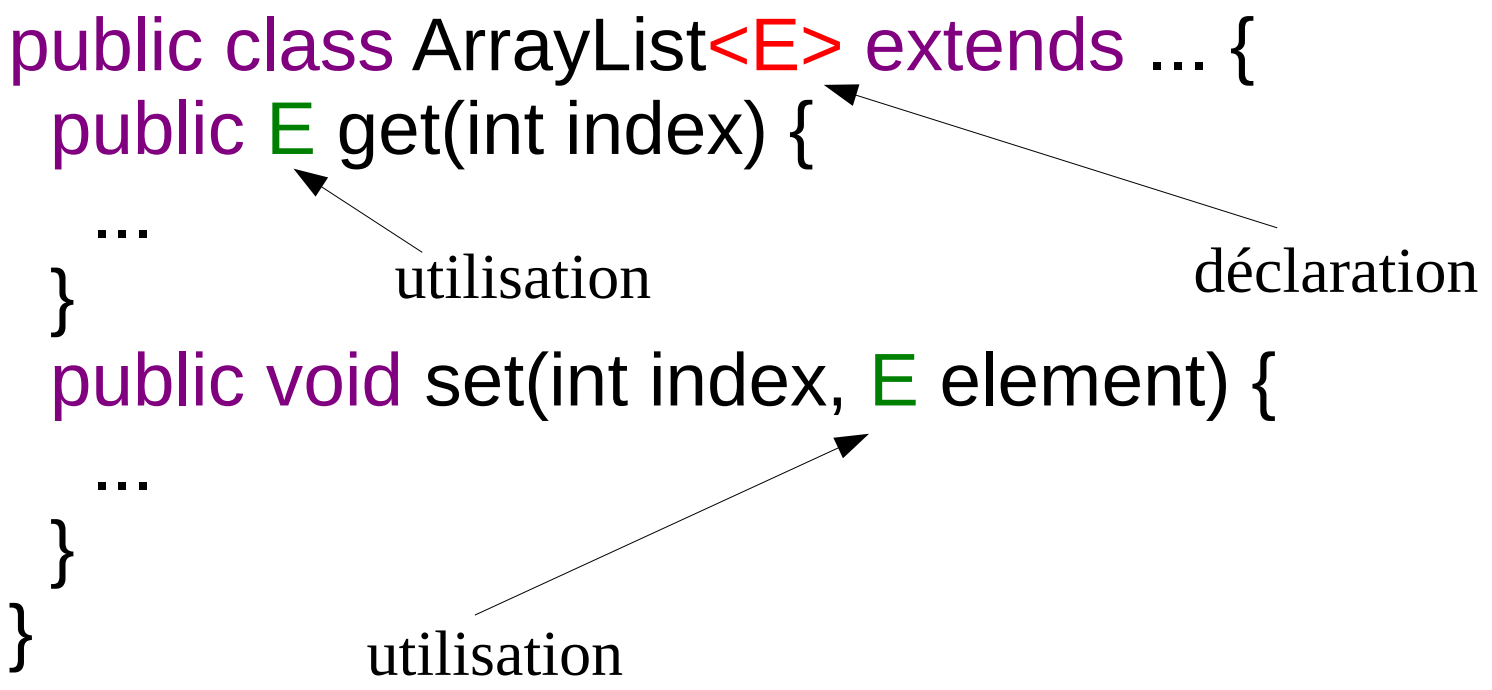


On associe un type argument à une instance d'un type paramétré

```
new ArrayList<String>();
new ArrayList<Integer>();
```

Type paramétré

Permet de vérifier et propager des types



Vérifier et propager

```
public class ArrayList<E> extends ... {
  public E get(int index) { ... }
  public void set(int index, E element) { ... }
}
```

- Vérifier un type

```
ArrayList<String> list = ...
list.set(3, new Object());   // compile pas
```

- Propager un type

```
ArrayList<String> list = ...
list.get(2).length();        // compile, list.get(2) est typé String
```

Inférence de type

La syntaxe dite diamand permet de demander l'inférence des types arguments comme avec les méthodes paramétrées

```
ArrayList<String> list = new ArrayList<>();
```

marche même avec plusieurs variables de type

```
HashMap<String, List<String>> =
                                    new HashMap<>();
```

Contexte statique

Une variable de type d'un type paramétré n'est pas accessible dans un context statique

```
public class Box<T> {
  private final T value;

  static void foo(T t) { } // compile pas
  static T t;              // compile pas
}
```

Limitations

L'implantation des types paramétrés en Java est appelé generics

Le generics existe pour le compilateur mais **pas pour la machine virtuelle**

donc toutes les opérations nécessitant que la machine virtuelle vérifie le type à l'exécution **ne fonctionne pas** !

Un seul code !

Cette limitation permet d'avoir un seul code à l'exécution

- ArrayList<Integer> et ArrayList<String> sont un seul et même code à l'exécution ArrayList.class

Evite les problème du C++

Explosion du code !

Mais comment on gère les types primitifs

On les gère pas :(

on utilise le boxing (mais ça coûte)

Opérations dynamiques problèmatiques

- instanceof T ou instanceof List<T>
  - interdit
- new T, new T[]
  - Interdit
- new ArrayList<String>[5]
  - Interdit
    - pas possible de faire le instanceof pour le ArrayStoreException
- (T) ou (ArrayList<String)
  - Permis mais le compilateur indique un warning (car pas de vérification à l'exécution)

Raw Type

Java laisse la possibilité d'utiliser des types sans '<' '>' pour appeler des bibliothèques qui ont été écritent avant l'introduction des types paramétrés (1.5)

- Dans ce cas, le compilateur emet un warning

A ne pas utiliser dans les nouveaux code !

```
ArrayList list =
        new ArrayList<String>(); // ok, mais beurk
ArrayList<String> list =
        new ArrayList(); // warning
```