

# Type et méthode paramétrés

Rémi Forax

# Type paramétré

Ajouté en 2004 à Java 5

Permettre au compilateur de suivre/tracker les types des éléments des collections

Exemple avant Java 5

```
ArrayList list = new ArrayList();  
list.add("hello");
```

```
String s = list.get(0); // ne compile pas
```

```
String s = (String) list.get(0); // Ok, mais dangereux
```

# Cast d'objets en Java

Les casts d'objets sont vérifiés à l'exécution par la machine virtuelle

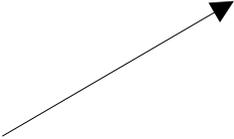
Toujours avant Java 5

```
ArrayList list = new ArrayList();  
list.add("hello");  
list.add(3);
```

// et plus tard dans le code

```
String s = (String) list.get(1); // plante avec CCE
```

ClassCastException



# CCE et Maintenance

Avoir des casts dans un programme veut dire que le programme peut planter à un endroit si on n'ajoute pas les bons objets à un autre endroit

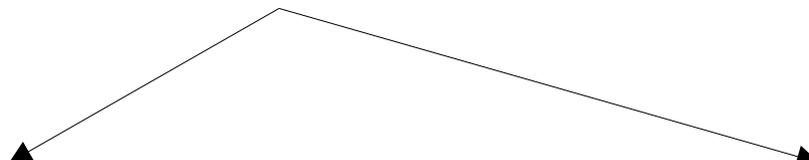
Et les deux endroits peuvent être éloignés

## Le but des types paramétrés

supprimer ces casts d'objet en ajoutant le type des objets stockés au type de la collection

# Type paramétré

A partir de Java 5,  
on ajoute le type des éléments



```
ArrayList<String> list = new ArrayList<String>();  
list.add("hello");
```

```
list.add(3); // ne compile pas
```

```
String s = list.get(0); // ok
```



plus besoin de cast

Déclaration d'un type paramétré

# Déclaration de type

On déclare les variables de type (E) entre “<” et “>” après le nom de la classe/record (séparées par des virgules s’il y en a plusieurs)

```
public record Holder<E>(E element) {  
    public E value(E defaultValue) {  
        return element != null? element: defaultValue;  
    }  
}
```

déclaration

utilisation

Après la déclaration, E est une variable de type que l’on peut utiliser là où habituellement on utilise un type (déclaration de champs, de variable, de paramètre, etc)

# Utilisation d'une variable de type dans un type paramétré

On utilise E comme type

- pour les champs d'instance (et les composants de record)
- dans les méthodes d'instances (et constructeur)

E n'est pas accessible dans les champs et méthodes static

```
public record Holder<E>(E element) {  
    public static void main(String[] args) {  
        E e = ... // ne compile pas  
    }  
}
```

# Utilisation d'un type paramétré

Avec la déclaration

```
public record Holder<E>(E element) {  
    public E value(E defaultValue) {  
        return element != null? element: defaultValue;  
    }  
}
```

Lorsque l'on déclare une variable "holder", le compilateur remplace la variable de type (E) par le type argument (String)

```
var holder = new Holder<String>("hello"); // pour "holder" E=String  
holder.value("") // Holder<String>.value(String) -> String
```

Déclaration d'une méthode paramétrée

# Déclaration de méthode paramétrée

Les méthodes sont paramétrées pour indiquer des relations entre le type des paramètres et le type de retour

```
public class Utils {  
    public static <T> List<T> from(T one, T two) {  
        return List.<T>of(one, two);  
    }  
}
```

déclaration

utilisation

On déclare les variables de type après les modificateurs de visibilité et avant le type de retour

# Utilisation d'une variable de type dans une méthode paramétrée

Dans une méthode paramétrée, la variable de type n'est accessible que dans cette méthode

```
public class Utils {  
    public static <T> List<T> from(T one, T two) {  
        T t; // ok  
    }  
    private final T t; // ne compile pas  
}
```

# Utilisation d'une méthode paramétrée

Pour appeler une méthode paramétrée, il faut mettre les “<” et “>” après le ‘.’ et avant le nom de la méthode

```
Utils.<String>from("foo", "bar")
```

Attention, ne pas écrire

```
Utils.from<String>("foo", "bar")
```

le “<” est considéré comme le inférieur ( $2 < 3$ ), pas comme le début d'un type argument

# Nommage des variables de type

Par convention, une variable de type est nommée par une seule lettre en majuscule

souvent E (type des éléments) ou T (type)

Exemple de code à ne pas écrire

```
public class Ahhh<String> {  
    public void m() {  
        List.<String>of("hello") // ne compile pas  
                                // java.lang.String n'est pas String  
    }  
}
```

# Inférence pour les types paramétrés

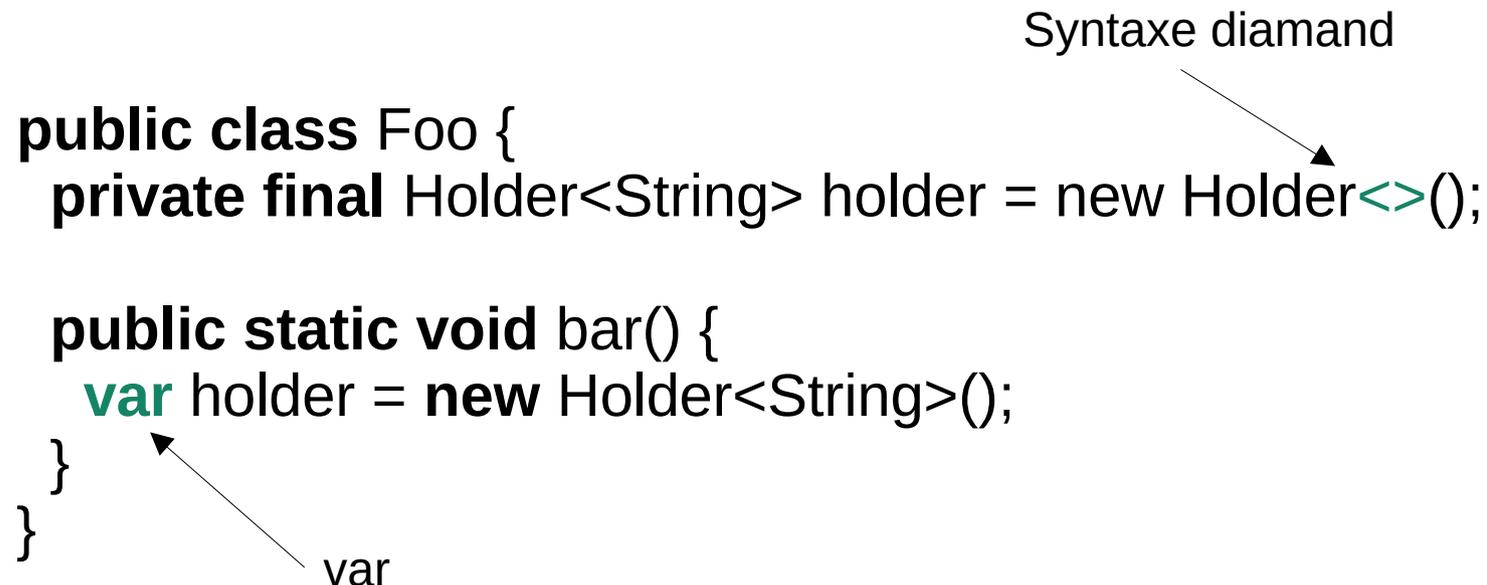
# Inférence avec un constructeur

On peut demander au compilateur de trouver le type argument tout seul

```
public class Foo {  
    private final Holder<String> holder = new Holder<>();  
  
    public static void bar() {  
        var holder = new Holder<String>();  
    }  
}
```

Syntaxe diamand

var



L'inférence se fait à partir du type des arguments / type de retour d'un appel au constructeur

# Inférence du constructeur

La syntaxe diamand infère de gauche à droite

```
List<String> strings = new ArrayList<>();
```

Le syntaxe var (seulement pour les variables locales) infère de droite à gauche

```
var strings = new ArrayList<String>();
```

Et si on fait les deux en même temps

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

quand le compilateur ne sait pas, il utilise Object

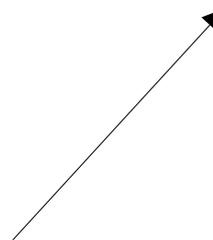
# Inférence pour les méthodes paramétrées

# Inférence lors de l'appel de méthode

Si il n'y a pas de "<" et ">", le compilateur essaye de trouver les types argument en fonction des arguments et du type de retour

```
public static List<CharSequence> m() {  
    return Utils.from("hello", "inference");  
}
```

Pas de "<", ">",  
donc le compilateur doit inférer



Les contraintes  
T est un super-type de String  
List<T> est un sous-type de List<CharSequence>

↓  
donc T = CharSequence

# L'inférence, bien ou mal ?

L'inférence marche assez bien donc il est assez rare d'avoir à spécifier les types argument explicitement

Cela dit c'est pratique de pouvoir spécifier le type argument pour débbugger (en particulier avec les lambdas)

Car avec l'inférence, la vraie erreur va être noyée parmi les contraintes qui seront listées

Limitation des generics

# generics

## Nom de l'implantation des types paramétrés en Java

Les types paramétrés n'existent que pour le compilateur pas pour la VM à l'exécution

Cette astuce s'appelle l'erasure

### Pratique

- car aller chercher des méthodes se fait toujours avec des String
- car c'est retro-compatible

### Pas pratique

- car certaines opérations ont besoin des types à l'exécution donc ces opérations sont interdites sur les variables de type/types paramétrés

# Limitation de l'erasure

Les opérations ci-dessous sont interdites

- `class A extends T {}`

  - La classe de T n'existe pas à l'exécution

- `new T, new T[]` ou `new List<T>[]`

  - La classe de T n'existe pas à l'exécution

- `instanceof T` ou `instanceof List<T>`

  - La classe de T n'existe pas à l'exécution

- `(T), (List<T>)` ou `(List<String>)` fait un warning

  - Le cast n'est pas vérifié à l'exécution

    - En même temps, le but des types paramétrés, c'est de supprimer les casts ... donc à ne pas utiliser

Wildcards et javadoc

# Les wildcards, les “?”

Une `List<String>` n'est pas un sous-type de `List<Object>` (cf cours de Master)

On doit écrire les règles de sous-typage explicitement

- `List<? extends String>`, liste d'un sous-type de `String`
- `List<? super String>`, liste d'un super-type de `String`
- `List<?>`, liste de n'importe quoi

`list.addAll()` marche avec une collection d'un sous-type

```
addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
```

`list.removeIf()` marche avec un predicate d'un super-type

```
removeIf(Predicate<? super E> filter) Removes all of the elements of this collection that satisfy the given predicate.
```

`list.retainAll()` marche avec une collection de n'importe quoi

```
retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
```

Dans le bytecode ...

```
public record Holder<E>(E element) {
    public E value(E defaultValue) {
        return element != null? element: defaultValue;
    }
}
```

```
...
var holder = new Holder<>("insert");
String result = holder.value("");
}
```

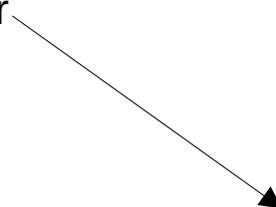
Inférence E = String



```
public final class Holder extends java.lang.Record {
    private final Object element;
        Signature: #26           // TE;
```

```
public Object value(Object)
    Signature: #37           // (TE;)TE;
    Code:
        0: aload_0
        1: getfield    #7           // Field element:LObject;
        4: ifnull    14
        7: aload_0
        8: getfield    #7           // Field element:LObject;
       11: goto      15
       14: aload_1
       15: areturn
```

Le compilateur ajoute un cast si il faut ressortir la valeur



```
public static void main(java.lang.String[]);
    Code:
        ...
       10: aload_1
       11: ldc      #18           // String
       13: invokevirtual #20       // Method value:(LObject;)LObject;
       16: checkcast #24         // class java/lang/String
       19: astore_2
       20: return
```

En résumé

# Type et méthode paramétrés

Le but des types paramétrés est d'éviter les casts non sûrs écrits par le programmeur

- en ajoutant le type des éléments aux types des collections
  - Le compilateur introduit les casts pour vous
- Attention à ne pas oublier les <...> si le type est paramétré (le warning est important)

Le compilateur essaye de deviner les types arguments en utilisant l'inférence (var, syntaxe diamand, par défaut pour les méthodes)