Généricité

Université Française d'Égypte
Richard Grin
Version O 0.9.1 – 15/9/13

Plan

- Pourquoi la généricité ?
- Présentation de la généricité
- Méthodes génériques
- Instanciation de type générique avec joker
- Paramètres de types contraints
- Implémentation (*erasure*)
- Interopérabilité avec du code non générique

Définition

- La généricité permet de paramétrer du code avec des **types de données**
- Exemple :
  - Classe **`ArrayList<T>`** dont le code est paramétré par un type **`T`**
  - Pour l'utiliser il faut passer un type en argument : **`new ArrayList<Employe>()`**

Pourquoi la généricité ?

Avant le JDK 5

- Les éléments des collections étaient déclarés de type **`Object`**
- Impossible de déclarer une collection d'**`Employe`** ou une collection de **`Livre`**
- Classe **`ArrayList`** :
  - **`boolean add(Object o)`**
  - **`Object get(int i)`**

Exemple d'utilisation de `ArrayList`

```
List employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
. . . // On ajoute d'autres employés
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe) employes.get(i)).getNom());
}
```

Que se passe-t-il sans ce *cast* ?

Exemple de problème

```
List employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
. . . // On ajoute d'autres employés
Livre livre = new Livre(...);
employes.add(livre);
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe) employes.get(i)).getNom());
}
```

Où est le problème ?

Ça compile ?
Ça s'exécute ?

Conséquences

- Des erreurs ne sont repérées qu'à l'exécution et pas à la compilation
- Il faut sans arrêt *caster* les éléments des collections

Le plus grave ?

Présentation de la généricité

Vocabulaire

- **`ArrayList<E>`** est un type générique
- **`E`** est un paramètre de type
- **`E`** sera remplacé par un argument de type :
  ```
  ArrayList<Integer> l =
      new ArrayList<Integer>();
  ```
- **`ArrayList<Integer>`** est une instanciation du type générique ; c'est un type paramétré
- Types « *raw* » : ancien type non générique **`ArrayList`**

Extraits de la classe `ArrayList`

```
public class ArrayList<E>
    extends AbstractList<E> {
  public ArrayList()
  public boolean add(E element)
  public E get(int index)
  public E set(int index, E element)
  public Iterator<E> iterator()
  . . .
```

Pas `ArrayList<E>` !

Utilisation de `ArrayList`

```
List<Employe> employes =
    new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      employes.get(i).getNom());
}
```

Plus besoin de *cast* ; pourquoi ?

Autre exemple d'utilisation

```
List<Employe> employes =
  new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
// Ajoute un livre au milieu des employés
Livre livre = new Livre(...);
employes.add(livre);
```

Ça compile ?
Ça s'exécute ?

Utilisations des paramètres de type

- Peuvent être utilisés pour déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes :

```
E element;
E[] elements;
```

- Ne peuvent pas être utilisés pour
  - créer des objets ou des tableaux
  - comme super-type d'un autre type
- ~~`new E()`~~
- ~~`new E[10]`~~

Création d'une instance de classe paramétrée

- Un argument de type pour chaque paramètre de type formel
- `Map<String,Integer> map =`
  `new HashMap<String,Integer>();`
- Depuis le JDK 7 :

```
Map<String,Integer> map =
    new HashMap<>();
```


Méthode générique

Méthode générique

- Une méthode peut être paramétrée par des types :
  `<T1,T2,…> type-retour m(…)`
- Exemple de l'interface **`Collection<E>`** :

```
<T> T[] toArray(T[] a)
```

- Exemple de la classe **`Collections`** :
-

```
static
<T> void fill(List<? super T> list,
              T obj)
```


Instanciation d'une méthode générique

- On peut appeler une méthode paramétrée en la préfixant par un argument de type :
  `<String>m()`
- Mais le plus souvent on peut omettre ce préfixe car le compilateur peut « deviner » le type d'après le contexte

Inférence de type

```
ArrayList<Personne> liste;
. . .
Employe[] res =
  liste.toArray(new Employe[0]);
```

Rappel : `<T> T[] toArray(T[] a)`
Que « devine » le compilateur pour `T` ?

Le compilateur infère
`liste.<Employe>toArray(new Employe[0]);`

Instanciation de type générique avec joker (*wildcard instantiation*)

Problème de sous-typage pour les types paramétrés

- **Attention**, si **B** hérite de **A**, **ArrayList<B>** n'hérite pas de **ArrayList<A>**
- Par exemple, **ArrayList<Employe>** n'hérite pas de **ArrayList<Personne>**

Raison

- Sinon, ce code compilerait

Pourquoi ça compilerait ?

```
ArrayList<Object> liste =
    new ArrayList<Personne>;
liste.add(new Velo());
```

- Ce code autoriserait l'ajout d'un **Velo** dans une liste de **Personne** !

En fait, pourquoi ça ne compile pas ?

Employe hérite de **Personne**

Conséquence

Que fait **afficheNoms**?

```
public static void afficheNoms(
    List<Personne> liste) {
  for(Personne p : liste) {
    System.out.println(p.getNom());
  }
}
```

```
List<Employe> liste =
    new ArrayList<>();
...
afficheNoms(liste);
```

Compilation ? Exécution ?

La solution : instanciation avec joker

- Permet de rendre une méthode plus réutilisable
- Exemple :
  List<? extends Number>
  est une liste dont l'argument de type est un sous-type de **Number** (mais il n'est pas connu exactement)

`Employe` hérite de `Personne`

Une bonne méthode pour afficher les noms d'une liste de personnes

- ```
  public static void afficheNoms(
      List<? extends Personne> liste) {
    for(Personne p : liste) {
      System.out.println(p.getNom());
    }
  }
  ```
  Pourquoi ça compile ?
- ```
  List<Employe> liste =
      new ArrayList<Employe>();
  ...
  afficheNoms(liste);
  ```
  Pourquoi ça compile ?

Les « types » avec joker

- **<?>** désigne un type *inconnu*
- **<? extends A>** désigne un type *inconnu* qui est un sous-type de **A** (**A** compris)
- **<? super A>** désigne un type *inconnu* qui est un sur-type de **A** (**A** compris)
- **A** peut être un type quelconque, sauf un type primitif
- On dit que **A** contraint le joker

Où peuvent apparaître les « types » avec joker ?

- Attention, abus de langage ! ce ne sont pas des vrais types
- Seulement pour instancier un type paramétré :
  `List<? extends Number>`
- Ne peuvent pas typer une variable :
  ~~`<? extends Number> n;`~~

Où peuvent apparaître les instanciations de types avec joker ?

- Dans une classe quelconque (générique ou non)
- Pour typer des variables, des paramètres, des tableaux ou les valeurs retour des méthodes :
  `List<? extends Number> l;`
- Ne peuvent pas être utilisées pour créer des objets ou des tableaux
- Ne peuvent pas être des super types :
  ~~`class C implements Comparable<? super C>`~~

Exemples d'utilisation

- Classe **ArrayList<E>** :
  ```
  public boolean addAll(
      Collection<? extends E> c)
  ```
  Explications ?
- Classe **Collections** :
  ```
  public static boolean disjoint(
      Collection<?> c1,
      Collection<?> c2)
  ```
  Les **?** de **c1** et **c2** sont indépendants

  Explications ?

Exemple de code de la classe Collections

```
static <T> void
fill(List<? super T> liste, T elem) {
  int size = liste.size();
  for (int i = 0; i < size; i++)
    liste.set(i, elem);
}
```
Que fait cette méthode ?

- Remplace tous les éléments d'une liste par un certain élément
- Intuition : pour remplir avec un objet de type **T**, le type des éléments de la liste doit être un sur-type de **T**

Exemple de code de la classe Collections

```
static <T> void
fill(List<? super T> liste, T elem) {
  int size = liste.size();
  for (int i = 0; i < size; i++)
    liste.set(i, elem);
}
```

Possible de remplacer par une boucle for-each ?

- Remplace tous les éléments d'une liste par un certain élément
- Intuition : pour remplir avec un objet de type **T**, le type des éléments de la liste doit être un sur-type de **T**

Sous-typage

- Soit **B** une classe fille de **A**
- **List<B>** est-il un sous-type de **List<? extends A>** ?
- Oui, car **B** est bien un sous-type de **A**
- **List<? extends A>** est-il un sous-type de **List<B>** ?
- Non, le « **?** » peut représenter un sous-type de **A** qui n'est ni **B**, ni un sous-type de **B**

Question

- Donc, **ArrayList<Personne>** est un sous-type de **ArrayList<? extends Object>**
- On a vu que si **ArrayList<Personne>** était un sous-type de **ArrayList<Object>**, on pourrait ajouter un **Velo** dans une liste de **Personne**
- Ne va-t-on pas avoir le même problème avec les instanciations avec joker ?

Problème ?

- Ne peut-on avoir le code suivant ?

```
ArrayList<? extends Object> l =
    new ArrayList<Personne>;
l.add(new Velo());
```

- Non, le compilateur impose des restrictions sur l'utilisation des instanciations de type avec joker
- Intuition : on ne peut ajouter un **Velo** car le type inconnu « **? extends Object** » peut ne pas être **Velo**
- Et si on remplace **Velo** par **Personne** ?

Échanges d'une classe avec l'extérieur

- Une classe fournit des objets à l'extérieur par ses méthodes qui renvoient des objets : **TypeRetour m2()**
- Elle reçoit des objets de l'extérieur par ses méthodes qui prennent des objets en paramètre : **Ret m1(TypeDuParametre x)**

Classe générique

- **C<E>** : classe qui manipule des objets de type **E**
- Restrictions des instanciations de type avec joker pour l'utilisation des méthodes qui échangent des objets de type **E** avec l'extérieur :
  - **void m1(E e)** (extérieur → C)
  - **E m2()** (C → extérieur)

Objets passés à C<? extends T>

- Utilisation de la méthode m1 de signature « void m1(E) » : `c.m1(x);`
- « Signature » de m1 pour cette instantiation : « void m1(? extends T) »
- Comment doit être le type **X** de **x** pour que l'affectation « x → ? extends T » soit acceptée par le compilateur ?
- Aucun **X** pour lequel cette affectation est assurée de ne pas poser de problème
- Donc l'appel d'une telle méthode est interdite

Application

- Cette règle s'applique à la méthode **add** de **List** :

```
ArrayList<? extends Object> l =
    new ArrayList<Personne>();
l.add(new Velo());
```

Ouf !

Autres restrictions

- Il existe d'autres restrictions pour toutes les instanciations avec joker

Objets fournis par C<? extends T>

- Utilisation de la méthode m2 de signature « E m2() » : `X x = c.m2();`
- **X** doit être de type **T** ou d'un sur-type de **T**

Exemple

```
ArrayList<? extends Personne> l =
      new ArrayList<Employe>();
...
Personne e = l.get(0);
```

Ca compile ?

Et si on remplace **Personne** par **Employe** ?

Objets passés à C<? super T>

- Utilisation de la méthode m1 de signature « void m1(E) » : `c.m1(x);`
- **X** doit être le type **T**, ou un sous-type de **T**

Exercice

```
void f(List<? super Cercle> liste) {
   liste.add(machin);
}
```

De quel type peut être machin ?

- **machin** doit être déclaré du type **Cercle** (ou d'un sous-type)

Objets fournis par C<? super T>

- Utilisation de la méthode m2 de signature « E m2() » : `X x = c.m2();`
- Le seul cas possible : **X** doit être le type **Object**

Exemple

```
ArrayList<? super Employe> l =
      new ArrayList<Personne>();
...
Employe e = l.get(0);
```

Ca compile ?

Et si on remplace **Employe** par **Personne** ?

Instanciation C<?>

- En toute logique on a les contraintes des 2 autres types d'instanciation :
  - l'appel d'une méthode qui prend un paramètre de type **E** (le paramètre de type) est interdit
  - la valeur retournée par une méthode qui renvoie un **E** ne peut être affectée qu'à une variable de type **Object**

Utilité de ces règles

- Ces règles sont nécessaires pour obtenir du code correct
- On l'a vu pour la méthode **add(E elt)** de la classe **ArrayList<E>**
- Quelquefois, cependant, ces règles ajoutent une contrainte qui ne sert à rien
- C'est en particulier le cas pour les méthodes qui ne modifient pas la collection

Exemple

- Méthode de **ArrayList<E>** : `boolean contains(E elt)`
- On ne peut pas appeler la méthode pour une **ArrayList<? extends Number>**, alors que cet appel ne peut causer de problème
- C'est pour cette raison que la méthode de l'API a la signature `boolean contains(Object elt)`

Types paramétrés contraints (*bounded* en anglais)

Pourquoi des types contraints ?

- On ne peut rien supposer sur le type d'un paramètre de type **T**
- ```
  T v;
  …
  v.m(…);
  ```
  ne compile que si **m** est une méthode de **Object**

Problème

- Cependant, pour écrire une méthode de tri « **sort(ArrayList<E>)** », il est indispensable de pouvoir comparer les éléments de la liste
- Il faut un moyen de dire que **E** contient une méthode pour comparer les instances du type
- Par exemple en disant que **E** implémente **Comparable**

Syntaxe

- Soit **T** un paramètre de type d'un type ou d'une méthode générique
- On peut indiquer que **T** hérite d'une classe mère **M** (au sens large) :
  ```
  <T extends Mere>
  ```
- ou qu'il implémente une ou plusieurs interfaces :
  ```
  <T1 extends I1 & I2, T2, T3>
  ```
  2 autres paramètres de type

Syntaxe (2)

- Si **T** hérite d'une classe mère et implémente des interfaces, la classe mère doit apparaître en premier :
  ```
  <T extends Mere & I1 & I2>
  ```


Quels types pour contraindre un paramètre de type ?

- Tous les types sont permis, sauf les types primitifs et les tableaux :
  ```
  <T extends Number>
  <T extends List<String>>
  <T extends ArrayList<? extends Number>>
  <T extends E>
  ```
  ```

Exemple

Que fait cette méthode ?

```
public static <T extends Comparable<? super T>>
T min(List<T> liste) {
  if (liste.isEmpty())
    return null;
  T min = liste.get(0);
  for (int i = 1; i < liste.size(); i++) {
    if (liste.get(i).compareTo(min) < 0)
      min = liste.get(i);
  }
  return min;
}
```

Meilleure signature ?

Exemple

```
public static <T extends Comparable<? super T>>
T min(List<? extends T> liste) {
  if (liste.isEmpty())
    return null;
  T min = liste.get(0);
  for (int i = 1; i < liste.size(); i++) {
    if (liste.get(i).compareTo(min) < 0)
      min = liste.get(i);
  }
  return min;
}
```

Encore mieux ?

Exemple

```
public static <T extends Comparable<? super T>>
T min(Collection<? extends T> coll) {
  if (coll.isEmpty())
    return null;
  T min = coll.get(0);
  for (int i = 1; i < coll.size(); i++) {
    if (coll.get(i).compareTo(min) < 0)
      min = coll.get(i);
  }
  return min;
}
```

Remplacer `List` par `Collection`

Problème ?

Comment faire ?

Exemple

```
public static <T extends Comparable<? super T>>
T min(Collection<? extends T> coll) {
  if (coll.isEmpty()) return null;
  Iterator<? extends T> it = coll.iterator();
  T min = it.next();
  while (it.hasNext()) {
    T next = it.next();
    if (next.compareTo(min) < 0)
        min = next;
  }
  return min;
}
```

Utiliser un itérateur plutôt que `get`

Implémentation de la généricité et restrictions associées

Instanciation des classes génériques

- À l'exécution **`ArrayList<Integer>`** et **`ArrayList<Employe>`** sont représentés dans la JVM par la seule classe **`ArrayList`**

Une contrainte importante

- Dans le cahier des charges de la généricité en Java :
  tout le code écrit avant la généricité doit pouvoir être utilisé avec la version de Java qui offre la généricité
- La solution choisie pour respecter cette contrainte est appelée « effacement de type » (*type erasure*)

C'est le compilateur qui fait tout

- Il transforme le code générique en code non générique

Exemple du travail du compilateur

```
List<Employe> employes =
    new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      employes.get(i).getNom());
}
```


Exemple du travail du compilateur

```
List employes =
    new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe)employes.get(i)).getNom());
}
```


Des complications

- Cet effacement implique des compromis et des impossibilités qui compliquent l'utilisation de la généricité en Java