Généricité

Université Française d'Égypte
Richard Grin
Version O 1.11 – 15/12/20

1

Plan

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

2

Présentation

- La généricité permet de paramétrer une classe, une interface ou une méthode avec des **types de données**
- Exemple :
  - Classe `ArrayList<T>` paramétrée par le type `T`
  - Pour l'utiliser il faut passer un type en argument : `ArrayList<Employe>`

3

Pourquoi la généricité ?

4

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)`

5

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* ?

6

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 ?

7

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 ?

8

Présentation de la généricité

9

Vocabulaire

- `ArrayList<E>` est un type générique
- `E` est un paramètre de type (ou variable de type)
- `E` sera remplacé par un argument de type :
  ```
  ArrayList<Integer> l =
      new ArrayList<Integer>();
  ```
- `ArrayList<Integer>` instancie le type générique `ArrayList<E>`
- Types « *raw* » : ancien type non générique `ArrayList`

10

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 Iterator<E> iterator()
  . . .
```

Pas ~~ArrayList<E>()~~ !

11

Utilisation de `ArrayList`

```
List<Employe> 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(
      employes.get(i).getNom());
}
```

Plus besoin de cast ; pourquoi ?

12

Autre exemple d'utilisation

```
List<Employe> employes = new ArrayList<>();
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 ?

13

Utilisation d'un paramètre de type

- Pour déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes :

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

- Ne peut pas être utilisé pour créer des objets ou des tableaux :

~~new E()~~
~~new E[10]~~

14

Création d'une instance de classe générique

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

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


15

Méthode générique

16

Méthode générique

- Comme une classe 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)
```

Que fait cette méthode ?

Expliquez les types

17

Instanciation d'une méthode générique

- On appelle une méthode générique 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 par le compilateur)

18

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

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

Pourquoi ça compilerait ?

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

Pourquoi ça ne compile pas ?

Conséquence

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

Que fait `afficheNoms` ?

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

Que fait ce code ?

Compile ?

`Employe` hérite de `Personne`

La solution : instanciation avec joker

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

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<>();
...
afficheNoms(liste);
```

Pourquoi ça compile ?

`Employe` hérite de `Personne`

25

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

26

Où peuvent apparaître les « types » avec joker ?

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

27

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

28

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 ?

29

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 ?

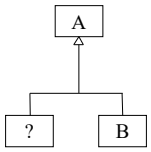
Possible de remplacer par une boucle for-each ?

- Remplace tous les éléments d'une liste par l'objet `elem` passé en paramètre
- Intuition : pour remplir avec un objet de type `T`, le type des éléments de la liste doit être un sur-type de `T`

30

Sous-typage

- Soit `B` une classe fille de `A`
- `List<B>` est-il un sous-type de `List<? extends A>` ?
- `List<? extends A>` est-il un sous-type de `List<B>` ?



31

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 ?

32

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` (ou un sur-type de `Velo`)

33

Problème ?

- Ne peut-on avoir le code suivant ?

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

- 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` (ou un sur-type de `Velo`)
- Et si on remplace `Velo` par `Personne` ?

Ça ne marche pas mieux

34

Échanges d'une classe avec l'extérieur

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

35

Restrictions sur instanciation avec joker

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

36

Extérieur → C<? extends T>

- Méthode … m1(E) dans la classe C<E>
- « Signature » de m1 pour C<? extends T> :
  « … m1(? extends T) » « E = ? extends T »
- c.m1(x)
  Comment doit être le type X de x pour que l'affectation « x → ? extends T » soit acceptée par le compilateur ?
- Aucun X ne convient
- Donc l'appel d'une telle méthode est interdite

37

Application

- Cette règle s'applique à la méthode add(E e) de ArrayList<E> :

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

Ouf !

38

Autres restrictions

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

39

C<? extends T> → extérieur

- Méthode E m2() de C<E> « E = ? extends T »
- X x = c.m2(); Quelle condition sur X ?
- m2 retourne une valeur de type « ? extends T »
- Donc x doit pouvoir recevoir une valeur de type « ? extends T »
- X doit être de type T ou d'un sur-type de T

40

Exemple

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

Ca compile ?

Et si on remplace Personne par Employe ?

41

Extérieur → C<? super T>

- Méthode void m1(E) « E = ? super T »
- c.m1(x) Quelle condition sur le type X de x ?
- On doit pouvoir affecter x à « ? super T »
- X doit être le type T, ou un sous-type de T

42

Exemple

- ```
  void f(List<? super Cercle> liste) {
    liste.add(machin);
  }
  ```
  De quel type peut être `machin` ?
- `machin` doit être déclaré du type `Cercle` (ou un sous-type)

43

`C<? super T>` → extérieur

- Méthode `E m2(…)` « `E = ? super T` »
- `X x = c.m2(…);` Quelle condition sur `X` ?
- Le seul cas possible : `X` doit être `Object`

44

Exemple

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

Ça compile ?

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

Quel type à la place de `Employe` pour que ça compile ?

45

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 retourne un `E` ne peut être affectée qu'à une variable de type `Object`

46

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

47

Exemple

- Méthode de `ArrayList<E>` :
  `boolean contains(E elt)`
- On ne peut pas appeler la méthode pour une `ArrayList<? extends Personne>`, 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)`

48
```

Types paramétrés contraints (*bounded* en anglais)

49

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`

50

Problème

- 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`

51

Syntaxe

- Soit `T` un paramètre de type
- On peut indiquer que `T` hérite d'une classe mère `M` :
  `<T extends Mere>`
- ou qu'il implémente une ou plusieurs interfaces :
  `<T1 extends I1 & I2, T2, T3>`

2 autres paramètres de type

52

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>`

53

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>
```


54

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 ?

Pourquoi ça compile ?

55

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 ?

56

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 ?

57

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`

58

Exemple – une erreur !

```
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()) {
    if (it.next().compareTo(min) < 0)
        min = it.next();
  }
  return min;
```

Quelle est l'erreur ?

59

Implémentation de la généricité et restrictions associées

60

Instanciation des classes génériques

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

61

C'est le compilateur qui fait tout

- Il transforme le code générique en code non générique

62

Exemple du travail du compilateur

```
List<Employe> 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(
      employes.get(i).getNom());
}
```


63

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());
}
```


64

Des complications

- Cet effacement de type (`List` remplace `List<Employe>`) implique des compromis et des impossibilités qui compliquent l'utilisation de la généricité en Java
- Par exemple, si `E` est un paramètre de type,
  `objet instanceof E`
  n'a aucun intérêt puisque qu'à l'exécution c'est
  `objet instanceof Object`
  qui sera testé

65