Généricité

Michel Schinz

2015-10-30

1 Introduction

Lors de l'écriture d'une classe, il peut arriver que certains types soient inconnus. Cela est par exemple le cas lors de l'écriture de classes « conteneurs » dont le but est de stocker d'autres objets de type quelconque.

La classe des tableaux dynamiques, `ArrayList`, est une telle classe conteneur, et il est clair qu'un problème se pose lors de son écriture : quel type utiliser pour les éléments du tableau ?

Pour résoudre ce problème, Java offre la notion de généricité, présenté ci-dessous au moyen d'un exemple plus simple que celui des tableaux dynamiques : les cellules.

2 Cellule

Une **cellule** est un objet (ici immuable) dont le seul but est de contenir un, et un seul, autre objet. A première vue, les cellules ne sont pas très utiles en pratique, mais elles permettent d'illustrer facilement l'intérêt de la généricité.

2.1 Cellule de chaîne

Une cellule de chaîne est une version restreinte d'une cellule, uniquement capable de stocker une chaîne de caractères, et non pas un objet de type quelconque. La chaîne contenue dans une telle cellule lui est passée au moment de la construction, et peut s'obtenir au moyen d'une méthode nommée `get`.

La définition d'une classe nommée `CellOfString` et représentant une cellule de chaîne est triviale :

```
public final class CellOfString {
  private final String s;

  public CellOfString(String s) {
    this.s = s;
```

```
  }
  public String get() {
    return s;
  }
}
```

Une fois cette classe définie, on peut p.ex. l'utiliser ainsi :

```
CellOfString message =
  new CellOfString("Bonne␣année␣2016");
System.out.println(message.get());
```

2.2 Cellule de date

Admettons maintenant que l'on ait également besoin d'une classe de cellule capable de stocker une date d'un type `Date` que l'on suppose exister. Une fois encore, la définition d'une telle classe, nommée `CellOfDate`, est triviale :

```
public final class CellOfDate {
  private final Date d;

  public CellOfDate(Date d) {
    this.d = d;
  }
  public Date get() {
    return d;
  }
}
```

Ce nouveau type de cellule peut s'utiliser avec le précédent :

```
CellOfString message = new CellOfString("Bonne␣année␣");
CellOfDate date = new CellOfDate(Date.today());
System.out.println(message.get() + date.get().year());
```

2.3 Généralisation

Les classes `CellOfString` et `CellOfDate` représentent des cellules de chaîne et de date, respectivement. Que faire si on désire également stocker de nombreux autres types dans des cellules, p.ex. des entiers, des tableaux, etc. ?

Une solution évidente serait de poursuivre sur la même voie et d'écrire une nouvelle classe cellule pour chaque cas. Cette solution, parfois nommée **spécialisation**, n'est toutefois clairement pas réaliste à grande échelle en raison de la (quasi-)duplication de code qu'elle implique.

En Java, il existe toutefois une autre solution, qui tire parti du fait que le type `Object` est un super-type de tous les autres types — sauf les types primitifs comme `int` ou `double`, nous y reviendrons. L'idée consiste à définir une classe de cellule dont l'élément a le type `Object` et de l'utiliser pour y stocker différents types d'objets comme des chaînes, des dates, etc. Examinons cette solution.

2.4 Cellule d'objet

La classe `Cell` ci-dessous représente une cellule capable de stocker un objet de type `Object`. Elle est bien entendu très similaire aux classes `CellOfString` et `CellOfDate`, la seule différence étant le type de l'attribut contenant l'objet, et le type de retour de la méthode `get` :

```
public final class Cell {
  private final Object o;

  public Cell(Object o) {
    this.o = o;
  }
  public Object get() {
    return o;
  }
}
```

Au moyen de cette unique classe de cellule, il est possible de récrire le programme d'exemple donné plus haut. Malheureusement, étant donné que la méthode `get` a pour type de retour `Object`, il faut rajouter quelques transtypages qui alourdissent le code :

```
Cell message = new Cell("Bonne␣année␣");
Cell date = new Cell(Date.today());
System.out.println((String)message.get()
                   + ((Date)date.get()).year());
```

En plus d'alourdir le code, ces transtypages ne sont pas sûrs, dans le sens où ils peuvent échouer à l'exécution si le programmeur fait une erreur et place une chaîne au lieu d'une date dans la cellule `date` :

```
Cell message = new Cell("Bonne␣année␣");
Cell date = new Cell("2016");
System.out.println((String)message.get()
                   + ((Date)date.get()).year());
```

Ce programme est valide mais lève l'exception `ClassCastException` lors de l'exécution, au moment où il tente de transtyper la chaîne extraite de la cellule `date` en `Date`.

Cette solution basée sur le type `Object`, même si elle est plus réaliste que la spécialisation et était utilisée dans les premières versions de Java, n'est donc pas non plus

satisfaisante : premièrement car elle implique l'ajout de beaucoup de transtypages, et deuxièmement car ces transtypages peuvent échouer à l'exécution si le programmeur commet une erreur.

3 Généricité

En raison des défauts de la spécialisation et de la solution basée sur le type `Object`, la notion de **généricité** (*genericity*), aussi appelée **polymorphisme paramétrique** (*parametric polymorphism*) a été introduite dans la version 5 de Java.

Au moyen de la généricité, il est possible de définir une cellule *générique*, c'est-à-dire capable de contenir un élément d'un type arbitraire, sans devoir faire de transtypages pour autant.

3.1 Cellule générique

Une classe pour les cellules générique (d'un type arbitraire) peut se définir ainsi :

```
public final class Cell<E> {
  private final E e;

  public Cell(E e) {
    this.e = e;
  }
  public E get() {
    return e;
  }
}
```

Cette classe est **générique**, et `E` est son **paramètre de type**. Il s'agit d'une variable (de type !) représentant le type de l'élément de la cellule. Dans le corps de la classe, `E` peut s'utiliser comme n'importe quel autre type — ou presque, comme nous le verrons plus loin.

Pour utiliser un type générique tel que `Cell`, il faut spécifier le type concret à utiliser pour son paramètre de type, comme dans les exemples suivants :

```
Cell<String>        // lire : cell of string
Cell<Date>          // lire : cell of date
Cell<Cell<String>>  // lire : cell of cell of string
```

On appelle ces types des **instanciations** du type générique `Cell`.

Contrairement aux cellules basées sur `Object`, les cellules génériques n'impliquent aucun transtypage :

```
Cell<String> message = new Cell<String>("Bonne␣année␣");
```

```
Cell<Date> date = new Cell<Date>(Date.today());
System.out.println(message.get() + date.get().year());
```

et ne comportent pas les mêmes risques, le programme ci-dessous étant refusé au moment de la compilation :

```
Cell<String> message = new Cell<String>("Bonne année ");
Cell<Date> date = new Cell<Date>("2016"); // interdit !
System.out.println(message.get() + date.get().year());
```

A noter que l'exemple plus haut peut encore être allégé, puisque les paramètres de type dans un énoncé `new` sont optionnels. S'ils sont omis, ils sont automatiquement inférés (c-à-d calculés) par le compilateur. L'exemple peut donc être récrit ainsi :

```
Cell<String> message = new Cell<>("Bonne année ");
Cell<Date> date = new Cell<>(Date.today());
System.out.println(message.get() + date.get().year());
```

Attention, lorsqu'on omet ainsi les paramètres de type, il est néanmoins obligatoire de placer une paire de crochets vide (`<>`) après le nom du constructeur. En raison de son apparence, cette paire de crochets vide est souvent appelée le **diamant** (*diamond*).

4 Paire générique

Considérons maintenant un cas à peine plus complexe que celui de la cellule en essayant d'écrire une classe représentant une paire de valeurs (c-à-d une cellule à deux éléments). Bien entendu, ces valeurs doivent être de type quelconque !

Une première tentative pourrait ressembler à ceci :

```
public final class Pair<E> {
  private E f, s; // f = first, s = second

  public Pair(E f, E s) {
    this.f = f;
    this.s = s;
  }
  public E fst() { return f; }
  public E snd() { return s; }
}
```

Cette première version est toutefois trop limitative, car elle force les deux éléments de la paire à avoir le même type. Afin de les autoriser à avoir chacun un type différent, il suffit d'ajouter un second paramètre de type à la classe :

```
public final class Pair<F, S> {
  private final F f;
```

```
  private final S s;

  public Pair(F f, S s) {
    this.f = f;
    this.s = s;
  }
  public F fst() { return f; }
  public S snd() { return s; }
}
```

(Pour clarifier leur signification, ces deux paramètres de type ont été nommés `F` et `S`, pour *first* et *second*.)

Au même titre qu'une méthode peut avoir un nombre quelconque de paramètres (de valeur), une classe peut avoir un nombre quelconque de paramètres (de type).

4.1 Méthode générique

Admettons que l'on désire maintenant ajouter une méthode `pairWith` à `Cell` permettant d'obtenir une paire dont le premier élément est celui de la cellule, et le second lui est passé en argument.

```
public final class Cell<E> {
  private final E e;
  // … comme avant
  public Pair<E, XXX> pairWith(XXX s) {
    return new Pair<>(e, s);
  }
}
```

Quel type utiliser à la place de `XXX` pour le paramètre de `pairWith` ?

Une première idée serait d'utiliser le paramètre de type `E` :

```
public final class Cell<E> {
  private final E e;
  // … comme avant
  public Pair<E, E> pairWith(E s) {
    return new Pair<>(e, s);
  }
}
```

Cela est toutefois bien trop limitatif, car on ne peut ainsi créer que des paires dont le second élément a le même type que celui de la cellule…

Une seconde idée serait d'ajouter un second paramètre de type à la classe `Cell` :

```
public final class Cell<E,S> {
```

```
  private final E e;
  // … comme avant
  public Pair<E, S> pairWith(S s) {
    return new Pair<>(e, s);
  }
}
```

Mais il s'agit d'une mauvaise solution pour plusieurs raisons, entre autres parce qu'elle associe le paramètre de type `S` à la classe `Cell`, alors qu'il appartient clairement à la méthode `pairWith`.

La bonne solution consiste à associer le paramètre de type `S` à la méthode `pairWith`, ce qui peut se faire en Java. Attention toutefois, la syntaxe est assez surprenante !

```
public final class Cell<E> {
  private final E e;
  // … comme avant
  public <S> Pair<E, S> pairWith(S s) {
    return new Pair<>(e, s);
  }
}
```

Une méthode qui prend un ou plusieurs paramètres de types, comme `pairWith`, est dite **méthode générique** (*generic method*).

Pour appeler une méthode générique, il faut logiquement spécifier les types à utiliser pour ses paramètres de type. Là aussi, la syntaxe est surprenante au premier abord — même si elle est cohérente avec la syntaxe de définition des méthodes génériques. Ainsi, la méthode `pairWith` peut s'utiliser comme suit :

```
Cell<String> message = new Cell<>("Bonne␣année␣");
Pair<String, Date> pair =
  message.<Date>pairWith(Date.today());
System.out.println(pair.fst() + pair.snd().year());
```

Heureusement, la valeur des paramètres de type des méthodes générique peut souvent être inférée, et on peut donc simplement écrire :

```
Cell<String> message = new Cell<>("Bonne␣année␣");
Pair<String, Date> pair =
  message.pairWith(Date.today());
System.out.println(pair.fst() + pair.snd().year());
```

Notez que dans ce cas, il ne faut pas mettre le diamant (`<>`), car cela est incorrect. La raison pour laquelle le diamant est nécessaire dans les énoncés `new` mais interdit dans les appels de méthodes génériques sera examinée dans une leçon ultérieure.

4.2 Types primitifs

Pour mémoire, Java possède huit types dits **primitifs** (*primitive types*) dont les valeurs ne sont pas des objets : `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` et `double`.

Malheureusement, ces types primitifs ne peuvent pas être utilisés comme paramètres de type d'un type générique. Dès lors, le code suivant est erroné :

```
Cell<int> s = new Cell<>(1); // interdit !
```

Que faire si l'on désire par exemple créer une cellule contenant un entier (de type `int`) ?

Une solution consiste à stocker l'entier dans un objet de type `java.lang.Integer`, et créer une cellule de ce type. L'exemple devient :

```
Cell<Integer> c = new Cell<>(new Integer(1));
```

(Notez au passage que la classe `Integer` est elle-même une cellule, dont l'élément a le type `int`, ce qui prouve que les cellules ont leur utilité !)

On dit alors que l'entier a été **emballé** (*wrapped* ou *boxed* en anglais) dans un objet de type `Integer`. En plus de la classe `Integer`, le paquetage `java.lang` contient une classe d'emballage pour les sept autres types primitifs (`Boolean` pour `boolean`, `Character` pour `char`, `Double` pour `double`, etc.)

Bien entendu, lorsqu'on ressort le valeur emballée de la cellule, il faut la **déballer** (*unwrap* ou *unbox* en anglais) avant de pouvoir l'utiliser. Dans le cas des entiers, cela se fait au moyen de la méthode `intValue` de la classe `Integer` :

```
Cell<Integer> c = new Cell<>(new Integer(1));
int succ = c.get().intValue() + 1;
```

Des méthodes de déballage similaires existent dans toutes les classes d'emballage.

L'emballage et le déballage manuels étant lourds à l'usage, le code nécessaire peut être généré automatiquement. On appelle cela l'**emballage** — et le **déballage** — **automatique** (*autoboxing*).

L'exemple précédent peut donc également s'écrire ainsi :

```
Cell<Integer> c = new Cell<>(1);
int succ = c.get() + 1;
```

et est automatiquement transformé afin que l'entier 1 soit emballé avant d'être passés au constructeur puis déballé avant l'addition.

5 Limitations

Pour des raisons historiques, la généricité en Java possède les limitations suivantes :

1. la création de tableaux dont les éléments ont un type générique est interdite,
2. les tests d'instance impliquant des types génériques sont interdits,

3. les transtypages (*casts*) sur des types génériques ne sont pas sûrs, c-à-d qu'ils produisent un avertissement lors de la compilation et un résultat éventuellement incorrect à l'exécution,

4. la définition d'exceptions génériques est interdite.

Ces différentes limitations sont illustrées au moyen d'exemples ci-dessous.

5.1 Tableaux génériques

Limitation n°1 : la création de tableaux dont les éléments ont un type générique est interdite.

Par exemple, le code suivant est invalide :

```
static <T> T[] newArray(T x) {
  return new T[]{ x }; // interdit
}
```

Attention : seule la *création* (via `new`) de tableaux d'éléments génériques est interdite. Il est tout à fait possible de *déclarer* un tableau de type générique.

Bien entendu, les tableaux dynamiques (`ArrayList`) étant une classe générique tout à fait normale, ils ne souffrent pas de cette limitation. Seuls les tableaux primitifs du langage sont affectés.

5.2 Tests d'instance générique

Limitation n°2 : les tests d'instance impliquant des types génériques sont interdits. Par exemple, le code suivant est refusé :

```
<T> int printIfStringCell(Cell<T> c) {
  if (c instanceof Cell<String>) // interdit
    System.out.println(c);
}
```

5.3 Transtypages génériques

Limitation n°3 : les transtypages impliquant des types génériques ne sont pas sûrs.

Par exemple, le code suivant ne lève pas d'exception à l'exécution, alors qu'il devrait en lever une :

```
Cell<Integer> c = new Cell<>(1);
Object o = c;
Cell<String> c2 = (Cell<String>)o;
```

Un avertissement est toutefois produit.

5.4 Exceptions génériques

Limitation n°4 : une classe définissant une exception ne peut pas être générique.

En d'autres termes, aucune sous-classes de `Throwable` ne peut avoir de paramètres de type. Par exemple, la définition suivante est refusée :

```
class InvalidException<T> extends Exception {}
```

6 Références

- *Java Generics and Collections* de Maurice Naftalin et Philip Wadler, O'Reilly Media,
- *Java Generics FAQ* d'Angelika Langer, une liste des questions fréquentes liées à la généricité Java, et leur réponse.