Généricité

Pratique de la programmation orientée-objet
Michel Schinz – 2015-02-20

Cellules

Admettons que l'on désire écrire une classe très simple modélisant ce que nous appellerons une **cellule** (immuable), dont le but est de stocker un – et un seul – objet.
L'objet contenu dans une cellule lui est passé au moment de la construction, et peut s'obtenir au moyen d'une méthode nommée `get`.

Cellule de chaîne

Dans un premier temps, considérons le cas particulier d'une cellule de chaînes de caractères (type `String`). La définition de la classe est alors triviale :

```
final class StringCell {
  private final String s;
  public StringCell(String s) {
    this.s = s;
  }
  public String get() { return s; }
}
```


Cellule généralisée

La classe `StringCell` est limitée car elle ne peut rien stocker d'autre qu'une chaîne de caractères.
Comment faire pour définir une cellule capable de stocker un élément de type quelconque (mais connu) ?
Idées :

- écrire autant de classes qu'il existe de types (spécialisation), ou
- faire une cellule de `Object`, ou
- utiliser la généricité.

Spécialisation

La **spécialisation** consiste à écrire une classe cellule par type d'élément que celle-ci peut contenir. Cela devient vite irréaliste...

```
final class StringCell {
  private final String s; … }
final class IntCell {
  private final int i; … }
final class BooleanCell {
  private final boolean b; … }
final class StringCellCell {
  private final StringCell s; … }
final class ObjectCell {
  private final Object o; … }
```


Cellule de `Object`

Une solution plus réaliste que la spécialisation est l'utilisation du type `Object`, qui en Java est un super-type de tous les types (sauf les types de base) :

```
final class ObjectCell {
  private final Object o;
  public ObjectCell(Object o) {
    this.o = o;
  }
  public Object get() { return o; }
}
```

Cette solution était utilisée en Java avant l'introduction de la généricité.

Cellule de `Object`

Malheureusement, l'utilisation de telles cellules implique une grande quantité de transtypages (*casts*) :

```
ObjectCell o = new ObjectCell("hello");
char c = ((String)o.get()).charAt(0);
```

et comporte des risques...

```
ObjectCell o = new ObjectCell(new Object());
char c = ((String)o.get()).charAt(0);
```


Généricité

En raison des problèmes posés par la spécialisation et 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 capables de contenir un élément d'un type arbitraire.

Cellule générique

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

```
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, voir plus loin).

Instanciation

Pour utiliser un type générique comme `Cell`, il faut spécifier le type concret à utiliser pour le paramètre de type.

Exemples :

```
Cell<Object>
Cell<String>
Cell<Cell<String>>
```

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

Utilisation

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

```
Cell<String> o = new Cell<String>("hello");
char c = o.get().charAt(0);
```

et ne comportent pas les mêmes risques :

```
Cell<String> o =
  new Cell<String>(new Object());
char c = o.get().charAt(0);
```

interdit !

Inférence de types

Depuis la version 7 de Java, il n'est pas nécessaire de spécifier explicitement les paramètres de type dans un énoncé **new**. Ceux-ci sont alors inférés (calculés) par Java.

On utilise pour cela les crochets vides <>, aussi appelés le diamant (*diamond*).

Ainsi, au lieu d'écrire comme précédemment :

```
Cell<String> o = new Cell<String>("hello");
```

on peut simplement écrire :

```
Cell<String> o = new Cell<>("hello");
```

diamant

Paires (v1)

Considérons maintenant un cas à peine plus complexe que celui de la cellule et essayons d'écrire une classe représentant une paire de valeurs. Bien entendu, ces valeurs doivent être de type quelconque !

Première tentative :

```
final class Pair<E> {
  private E f, s; // (first, second)
  public Pair(E f, E s) { this.f = f; … }
  public E getF() { return f; }
  public E getS() { return s; }
}
```

En quoi cette version est-elle limitée ?

Paires (v2)

Forcer les deux éléments de la paire à avoir le même type est trop limitatif ! Afin d'autoriser les paires d'éléments ayant un type différent, il suffit d'ajouter un second paramètre de type :

```
final class Pair<F,S> {
  private F f;
  private S s;
  public Pair(F f, S s) { this.f = f; … }
  public F getF() { return f; }
  public S getS() { return s; }
}
```


Cellule vers paire

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.

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

Problème : quel type utiliser pour le paramètre de `pairWith` ?

Cellule vers paire

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

```
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…

Que faire ?

Cellule vers paire

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

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

Mais c'est 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`.

Méthodes génériques

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 !

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

paramètre de type de `pairWith`

Une méthode qui prend un ou plusieurs paramètres de types, comme `pairWith`, s'appelle **méthode générique**.

Méthodes génériques

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 :

```
Cell<String> c = new Cell<>("hello");
Pair<String, Integer> p =
  c.<Integer>pairWith(12);
```

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

```
Pair<String, Integer> p =
  c.pairWith(12);
```


Généricité et types de base

Types de base

Rappel : Java possède 8 types dits *de base* qui ne sont pas des objets, à savoir `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` et `double`.
Malheureusement, les types de base 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 créer une cellule contenant un entier ou un autre type de base ?

Emballage

Solution : stocker la valeur de type `int` 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));
```

On dit alors que les entiers ont été **emballés** (*wrapped* ou *boxed*) dans des objets de type `Integer`.
Le paquetage `java.lang` contient une classe d'emballage par type de base (`Boolean` pour `boolean`, `Character` pour `char`, `Double` pour `double`, etc.)

Déballage

Bien entendu, lorsqu'on ressort le valeur emballée de la cellule, il faut la **déballer** (*unwrap* ou *unbox*) 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 similaire existent dans toutes les classes d'emballage.

Emballage automatique

L'emballage et le déballage manuels étant lourds à l'usage, le code nécessaire peut être produit 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.

Limitations de la généricité en Java

Limitations de la généricité

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

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

Généricité et tableaux

Limitation n°1 : la création de tableaux dont les éléments ont un type générique est interdite.
Par exemple, le code suivant est refusé :

```
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.

Test 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);
}
```

Transtypage générique

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.

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 BadException<T>  // interdit
  extends Exception {}
```