

Les classes génériques

Algorithmique et langage de programmation
Gaël Thomas



Héritage et réutilisation de code (rappel)

■ Une structure de données peut souvent être réutilisée

Exemple typique : le tableau extensible de `Monster` (voir CI3)

```
class Army {
    Monster[] content; /* tableau de monstres */
    int top;           /* qui est occupé jusqu'à top */

    void add(Monster m) {
        if(top == content.length) {
            Monster[] tmp = new Monster[content.length * 2];
            for(int i=0; i<content.length; i++)
                tmp[i] = content[i];
            content = tmp;
        }
        content[top++] = m; } }
```

Héritage et réutilisation de code (rappel)

Pour rendre le code réutilisable dans d'autres contextes, il suffit de :

1. Remplacer `Army` par `ArrayList`
2. Remplacer `Monster` par `Object` puisque **Monster** hérite de **Object**

```
class ArrayList {  
    Object[] content;    /* tableau d'objets */  
    int top;              /* qui est occupé jusqu'à top */  
  
    void add(Object m) {  
        if(top == content.length) {  
            Object[] tmp = new Object[content.length * 2];  
            for(int i=0; i<content.length; i++)  
                tmp[i] = content[i];  
            content = tmp;  
        }  
        content[top++] = m; } }
```

Pourquoi la programmation générique

- Problème : nécessite des sous-typage explicites à l'utilisation

```
class ArrayList {  
    ...  
    Object get(int i) { return content[i]; }  
}
```

renvoie **Object** car le type des objets contenus n'est pas connu à la conception

```
class Game {  
    ArrayList army;  
  
    void display(int i) {  
        Monster m = (Monster) army.get(i);  
        m.display();  
    }  
}
```

ce qui force un sous-typage explicite à l'utilisation

Pourquoi la programmation générique

Ces sous-typages explicites ne sont pas idéals pour le développeur !

- Les sous-typages explicites rendent le code confus et difficile à lire
- Et ne permettent pas au compilateur de s'assurer que `ArrayList` ne contient que des `Monster`
=> risque d'erreurs difficiles à détecter si le programme stocke par erreur autre chose qu'un `Monster` dans le `ArrayList`

La classe générique

- Classe générique = classe paramétrée par une autre classe

```
class Game {  
    /* ArrayList stocke des Monster et non des Object */  
    ArrayList<Monster> army;  
  
    void display(int i) {  
        /* plus besoin de sous-typer le résultat */  
        Monster m = army.get(i);  
        m.display();  
    }  
}
```

- Avantages

- Détection des incohérences de types à la compilation (impossible de stocker autre chose qu'un **Monster** dans `army`)
- Code plus facile à lire car plus de sous-typages explicites

Utilisation simple (1/2)

```
// la classe Bag est paramétrée par E
// ⇔ E inconnu ici, il sera connu à la déclaration
class Bag<E> {
    private E[] elements;
    public Bag() { elements = new E[42]; }
    public E get(int i) { return element[i]; }
}
```

```
Bag<Potion> b;    // déclare un Bag avec E valant Potion
b = new Bag<>(); // alloue un Bag (le paramètre Potion
                // est automatiquement déduit à partir du type de b)
```

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Utilisation simple (1/2)

```
// la clas  
// ⇔ E in  
class Bag<  
    private  
    public B  
    public E  
}
```

Remarque 1 : on peut aussi
compacter déclaration et allocation avec

on

```
Bag<Potion> b = new Bag<> ();
```

```
Bag<Potion> b; // déclare un Bag avec E valant Potion  
b = new Bag<> (); // alloue un Bag (le paramètre Potion  
est automatiquement déduit à partir du type de b)
```

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Utilisation simple (1/2)

Remarque 2 : on peut omettre le "<Potion>", dans ce cas, un "<Object>" est ajouté de façon transparente par le compilateur

Pas recommandé

créée par E
connu à la déclaration

Remarque 3 : on peut omettre le "<>"
Pas recommandé

```
}  
  
Bag<Potion> b; // déclare un objet de type Bag de type Potion  
b = new Bag<> (); // alloue un Bag (le paramètre Potion est automatiquement déduit à partir du type de b)
```

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Utilisation simple (2/2)

```
class Tree<E, F> { /* paramétré par E, F */  
    private Tree<E, F> left;  
    private Tree<E, F> right;  
    private E          key;  
    private F          value;  
    ...  
}
```

⇒ `Tree<String, Account>` déclare un `Tree` avec `E` valant `String` et `F` valant `Account`

Utilisation avancée

- Utilisation de `extends` si il faut que le type paramètre hérite d'une classe précise

```
class Bag<E extends Item> {  
    private E[] elmts;  
    public int getWeight() {  
        for(...) tot += elmts[i].getWeight(); ...  
    }  
}
```

`E` doit hériter de `Item` car `E` doit posséder la méthode `getWeight`

```
class Item {  
    abstract public int getWeight();  
}
```

Attention

Si un type paramètre doit mettre en œuvre une **interface** précise, on utilise **quand même** `extends`

■ U
d'

```
class Bag<E extends Item> {  
    private E[] elmts;  
    public int getWeight() {  
        for(...) tot += elmts[i].getWeight(); ...  
    }  
}
```

```
interface Item {  
    public int getWeight();  
}
```

Tableau de classes génériques

- Attention : pas d'allocation d'un tableau de classes génériques avec "<>"

```
Truc<String>[] t = new Truc<>[10];
```

- Allocation du tableau "comme si" `Truc` n'était pas générique

```
Truc<String>[] t = new Truc[10]; /* autorisé */
```

- En revanche, allocation des éléments de façon normale

```
t[0] = new Truc<>("Hello");
```

Notions clés

- Classe générique = classe paramétrée par d'autres types

```
class Truc<X, Y, Z extends Bidule>
```
- Déclaration en spécifiant les paramètres

```
Truc<A, B, C> truc;
```
- Allocation en ajoutant "<>" après le mot clé `new`

```
truc = new Truc<>();
```
- Pas de "<>" à côté du `new` pour allouer un tableau de classes génériques

```
Truc<A, B, C>[] trucs = new Truc[10];
```

Notion avancée : déduction de type

- Le compilateur Java déduit automatiquement que le paramètre de `new Bag<>()` est `Potion` via le type de `b` dans

```
Bag<Potion> b = new Bag<>();
```

- Mais le compilateur n'est pas toujours capable de déduire le type

```
class Potion { void drink() { ... } }  
class Bag<E> { E e; E get() { return e; } }  
(new Bag<>()).get().drink(); // déduction impossible
```

- Dans ce cas, il faut explicitement donner le paramètre lors de l'allocation

```
(new Bag<Potion>()).get().drink();
```