Introduction à la Programmation Orientée Objet

7 - Les classes génériques

Valentin Honoré

valentin.honore@ensiie.fr

FISA 1A

Héritage et réutilisation de code (rappel)

- On veut souvent réutiliser une structure de données
  - Exemple typique : le tableau extensible du DM

```
class LocationSet{
    private Location[] locations = new Location[2];
    private int nbLocations=0;
    /* Ajouter une nouvelle location */
    void add (Location l) {
        if (nbLocations == locations.length-1) {
            Location[] newLocations = new ... ;
            for (int i=0; i<locations.length; i++) {
                newLocations[i] = locations[i];
            }
            locations = newLocations;
        }
        locations[nbLocations] = location;
        nbLocations++;
    }}
```

Héritage et réutilisation de code (rappel)

- Pour rendre le code réutilisable dans d'autres contextes, il suffit de :
  1. Remplacer `Location` par `ArrayList`
  2. Remplacer `Location` par `Object` puisque `Location` hérite de `Object`

```
class ArrayList {
    Object[] content; /* tableau */
    int top; /* occupation tableau */
    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
  - `ArrayList` renvoie des `Object` (car type pas connu avant exécution)
  - ce qui force un sous-typage explicite à l'utilisation

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

class LocationSet {
    ArrayList locations;
    void display(int i) {
        /* Force un typage explicite ici */
        Location m = (Location)
            locations.get(i);
        m.display();
    }
}
```

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 `Location`

$\rightarrow$ risque d'erreurs difficile à détecter si le programme stocke par erreur autre chose qu'une `Location` dans le `ArrayList`

La classe générique

- Classe générique = **classe paramétrée par une autre classe**

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

- Avantages
  - Détection des incohérences de types à la compilation (impossible de stocker autre chose qu'une `Location` dans `locations`)
  - 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)

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

- 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, int>` déclare un `Tree` avec E valant `String` et F valant `int`

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

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

Attention

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

Tableau de classes génériques

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

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

- 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");
```

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();
```

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];
```