Java - 1ère année

Programmation générique

Cours 11

1 - Le problème

- Soit une collection particulière d'objets :

```
Vector v = new Vector()
```

implémentée par un développeur pour y stocker des objets de classe A.

Le développeur ajoute régulièrement des objets A à cette collection :

```
A a = new A();
v.add(a)
```

2 - Le problème (2)

- Le vecteur enregistre tous ces objets dans un `Object[]` interne qu'il gère à sa manière.
- Tant que le développeur met des A explicitement, il sait (à peu près) ce qu'il fait.
- Mais il peut mettre des A indirectement :

```
Object o = aBlob.getSomeThingFromBlob(); // un B
v.add(o);
```

- Comment le développeur peut-il savoir à l'avance que cette utilisation est erronée ?

3 - Le principe générique

- La programmation générique répond à ce problème, en permettant d'écrire des modèles de classe dont on peut forcer le « type » d 'utilisation.
- Si le Vector était un « Vecteur de A », le compilateur aurait pu voir le problème

Rappel : « *The compiler is your friend* »

4 - Le principe générique (2)

- Ecrire une classe une fois, puis paramétrer quel type d'objets elle manipule.
- Très utile pour les collections, afin d 'obtenir des « ensemble de », « listes de », etc.
- La programmation générique est généralisable à toute classe.

5 - Type générique

- Un type générique est une « classe » paramétrée.

```
class BoiteGenerique<T>{
    private T t;
    public BoiteGenerique(T t){
        this.t = t;
    }
    public T getContent(){
       return t;
    }
}
```

- Chaque fois qu'on voit T, on peut imaginer de le remplacer par n'importe quelle classe Java.

6 - Invocation d'un type générique

- On invoque un type en précisant pour quelle classe on l'utilise :

```
BoiteGenerique<Integer> bgi = new
        BoiteGenerique<Integer>(new
                                Integer(4));
```

Une fois obtenue l'instance, la méthode `getContent()` renvoie un `Integer` :

```
Integer i = bgi.getContent();
```

7 - Conventions

- La lettre T s'appelle un « paramètre de type »
- On admet certaines conventions lorsqu'on choisit une notation pour les paramètres de type :
  - E - Elément (utilisé dans les collections Java)
  - K - Clef
  - N - Nombre
  - T - Type
  - V - Valeur
  - S,U,V etc. - 2nd, 3ème, 4ème types.

8 - Méthode générique

- Le principe de la programmation générique peut être localisée à une méthode :

```
public <U> void putInBox(U u,
  BoiteGenerique<U>){
  ...
}
```

Cette méthode, placée dans une classe tout à fait normale, permet de mettre un objet de type variable dans une boîte appropriée.

9 - Méthode générique (2)

- Utiliser une méthode générique :
  - Pour utiliser la méthode précédente

```
public <U> void putInBox(U u,
  BoiteGenerique<U>){
  ...
}
```

il faudrait écrire :

```
<Crayon>putInBox(crayonRouge, trousse);
```

10 - Méthode générique (3)

- Simplification des appels
  - en fait, on admet que l ’on puisse simplement écrire :

```
putInBox(crayonRouge, trousse);
```

cette simplification s’appelle

« inférence de type »

11 - Multigénéricité

- On peut utiliser plusieurs paramètres de type dans la même construction générique :

```
Mapping<String,String> m;
```

- associe nécessairement des chaînes de caractères à des chaînes de caractères et serait une utilisation d'une instance d'une classe :

```
class Mapping<U, V>{
  ... U ... V ...
}
```

12 - Multigénéricité (2)

- La généricité peut s'utiliser en cascade :
  - Si Boite est une classe générique

    ```
    class Boite<T>{...
    ```

  - et List est une classe générique

    ```
    class List<T>{…
    ```

  - alors on peut utiliser des listes de boîtes

    ```
    List<Boite<Crayon>> trousses = ...
    ```

13 - Multigénéricité (3)

– Dans l'expression :

```
List<Boite<Crayon>> trousses = ...
```

Le compilateur peut vérifier une double contrainte :

- que la Boîte utilisée contient des crayons,
- que la List utilisée contient bien des « Boîtes de crayons »

14 - Types contraints

- Problème :
  - Peut-on imposer la nature des classes qui sont utilisables au moment d'exploiter un type paramétrique ?

```
public <T> void metGen(T t){ … }
```

admet un paramètre de type, mais n'impose rien sur la nature de T. On voudrait interdire l'utilisation de certaines classes au moment de l'utilisation.

15 - Types contraints (2)

- Solution :
  - Contraindre en imposant la classe de départ :

```
public <T extends Number> void
            metGen(T t){ … }
```

n'admet à l'utilisation que Number ou ses sous-types.

```
metGen("10");
```

échoue à la compilation (utilisation d'une String pour T).

16 - Types contraints (3)

- Contraintes multiples :
  - Pour ajouter une interface obligatoire à la contrainte :

```
<T extends SuperType & Interface1>
```

17 - Génériques et héritage

- 1 - Pour des classes courantes :

```
Object o = new Object();
Integer i = new Integer(1);
o = i ; // Valide
```

- 2 - Pour des génériques :

```
Boite<Number> bn1 = new
   Boite<Number>(new Integer(1));
Boite<Number> bn2 = new
   Boite<Number>(new Double(3.14));
```

18 - Génériques et héritage (2)

- 3 - Le type générique comme un type à part entière ?
  - Examinons la méthode :

```
public void inspect(Boite<Number> bn);
```

  - Peut-on l'utiliser avec un type **`Boite<Integer>`**, puisque **`Integer`** est sous-classe de **`Number`** ?

19 - Génériques et héritage (3)

- NON :

« Un type G<U> ne vaut pas pour un type G<T>, même si U vaut pour un T. »

20 - Jokers de type

- Nous revenons au problème des contraintes.
  - Peut-on dire pour une déclaration de classe générique : utilise uniquement cette sous-famille d'objets ?

```
class Boite<? extends Number>{}
```

Seuls Number et ses sous-classes peuvent être utilisées avec la boîte :

```
Boite<Integer> bi = new Boite<Integer>(1); // légal
Boite<String> bs = new Boite<String>("1"); // illégal
```

21 - Jokers de type

- Nous revenons au problème des contraintes.
  - Peut-on dire pour une déclaration de classe générique : utilise uniquement des classes très générales (au dessus de la classe C ou C)

```
class Boite<? super C>{}
```

Alors :

```
class C extends B{…}
class D extends C{…}
Boite<B> bb = new Boite<B>(); // légal
Boite<C> bc = new Boite<C>(); // légal
Boite<D> bd = new Boite<D>(); // illégal
```

22 - Jokers de type

- Ressemblances :

```
class Boite<T extends Number>{…}
class Boite<? extends Number>{…}
```

  - Dans les deux cas, l'usage du générique est restreint à la seule descendance de Number
- Différences :
  - Boite<Integer> n'est pas sous-classe de Boite<Number>
  - Boite<Integer> est sous-type de Boite<? extends Number>, mais avec certaines restrictions d'usage.

23 - Effacement du type

- Est nommé « effacement du type » la technique qui permet au compilateur de produire une classe compilée résolue, à partir d'un générique.
- La classe produite est compatible avec des machines virtuelles ne connaissant pas les génériques.

24 - Effacement du type (2)

- Toute référence explicite au type ne peut être compilé :

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {  //Compiler error
            ...
        }
        E item2 = new E();   //Compiler error
        E[] iArray = new E[10]; //Compiler error
        E obj = (E)new Object(); //Unchecked cast warning
    }
}
```

- E n’est pas une classe. Le terme E disparaît dans le code compilé.

FIN DU COURS