TP 3 : Types génériques

La programmation objet permet de modulariser le code et de définir par exemple un conteneur unique pour n'importe quelle type de donnée. Il est ainsi possible de définir une boîte qui peut contenir n'importe quel objet.

```
public class Box {
    private Object o;

    public void put(Object o){
        this.o = o;
    }

    public Object get(){
        return o;
    }
}
```

On pourrait l'utiliser ainsi :

```
Box b = new Box();
b.put (new Integer(15));
/* ... */
Integer i = (Integer) b.get();
b.put("15");
/* ... */
String s = (String) b.get();
```

Cependant, on voit rapidement les inconvénients d'avoir une définition unique utilisant le type `Object`. Tout d'abord, il faut systématiquement faire une conversion explicite vers le type voulu. Ensuite, on n'a pas de garantie immédiate sur le type donné qui se trouve vraiment derrière un `Object`. Ce qui peut générer des erreurs lors de l'exécution.

```
Box b = new Box();
b.put(new Integer (15));
/* ... */
String s = (String) s.get();
```

Un tel code va compiler, car on peut transtyper un `Object` en `String`, mais générera une erreur à l'exécution. On peut utiliser le mot-clef `instanceof` pour détecter ce genre d'erreurs, mais ce faisant, on alourdit encore le code. D'autre part, on souhaiterait détecter ces erreurs dès la compilation.

1 Types génériques

Depuis la version 1.5, Java dispose d'un mécanisme de polymorphisme qui permet de contourner ces difficultés : les *types génériques*. Ils permettent de définir de nouvelles classes paramétrées par un (ou plusieurs) type(s) de donnée. Nous pouvons définir notre boîte de cette manière :

```
public class Box<T>{
    private T t;

    public void put(T t){
        this.t = t;
    }

    public T get(){
        return t;
    }
}
```

La notation `<T>` indique qu'on définit un type générique en fonction d'une classe `T`. Dans le corps de la classe, on peut alors utiliser la notation `T` pour indiquer le type d'un argument ou d'une valeur de retour. On utilise un type générique en déclarant la classe voulue lors de la déclaration et de la déclaration d'un objet :

```
Box<Integer> bi = new Box<Integer>();
Box<String> bs = new Box<String>();
bi.put(new Integer(15));
bs.put("15");
/* ... */
Integer i = bi.get();
String s = bs.get();
```

On n'a donc plus besoin de faire une conversion de type explicite. D'autre part si l'on essayait d'obtenir une `String` depuis une `Box<Integer>` (ou inversement), on aurait une erreur à la compilation. Notez qu'on peut toujours obtenir une boîte générique en utilisant des `Box<Object>`. Il est également possible de rendre des méthodes génériques :

```
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
```

Cette méthode définie dans la classe `Box<T>` affichera le type manipulé par la boîte en question, et le type de l'argument qui est donné.

1.1 Sous-types génériques

Supposons maintenant que nous souhaitions travailler avec une boîte dont on veut que l'objet hérite d'une classe donnée, sans spécifier sa classe. On peut le faire de la façon suivante :

```
public static <U extends CharSequence> Box<U> greaterInBox(U u1, U u2) {
    Box<U> b = new Box<U>();
    if (u1.compareTo(u2) < 0)
        b.set(u2);
    else
        b.set(u1);
    return b;
}
```

Notez que le `extends` peut s'appliquer indifféremment à une classe ou à une interface. On peut mettre plusieurs limites sur une classe avec l'opérateur `&` :

```
public <U extends MyClass & MyInterface> void f(U u) {
    /* ... */
}
```

Quand une boîte indique qu'elle peut contenir un type d'objet, il est possible d'y ranger n'importe quel objet d'un sous-type de celui-ci. C'est possible grâce aux relations de sous-classes que vous avez vues :

```
Box<Number> b = new Box<Number>();
b.put(new Integer(5));  /* ok: Integer is a Number */
b.put(new Double(5.6)); /* ok: Double is a Number */
```

Ce code est correct, parce que `put` attend un `Integer`, et que `Integer` et `Double` en sont des sous-types. Par contre le code suivant n'est pas correct :

```
public static double doubleFromBox(Box<Number> b) {
    return b.doubleValue();
}

public static void main(String[] args) {
    Box<Integer> b = new Box<Integer>();
    /* ... */
    double d = doubleFromBox(b); /* wrong! */
}
```

En effet, même si `B` est sous-type de `A`, Java ne considère pas que `Box<B>` est sous-type de `Box<A>`. Mais il y a une notation pour indiquer « n'importe quel objet `Box` dont le type est sous-type de `A` » :

```
public static int intFromBox(Box<? extends Number> b) {
    return b.intValue();
}

public static void main(String[] args) {
    Box<Integer> b1 = new Box<Integer>();
    Box<Double> b2 = new Box<Double>();
    /* ... */
    int i1 = intFromBox(b1);
    int i2 = intFromBox(b2);
}
```

1.2 Super-types génériques

Il est parfois nécessaire de borner le type d'un paramètre générique par le bas. Si on considère la méthode `max` avec la signature suivante qui itère sur un ensemble d'objets et retourne le plus grand

```
public static <T extends Comparable<T>> T max(Collection<T> coll)
```

le compilateur Java ne va pas accepter le code suivant :

```
class Foo implements Comparable<Object> {
    ...
}
Collection<Foo> cf = ... ;
Collections.max(cf); // Should work.  But doesn't.
```

Le problème est que `Foo` n'implémente pas `Comparable<Foo>`.

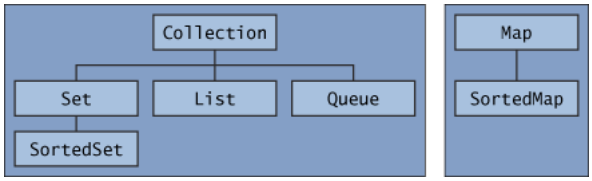
Pour la méthode `max`, il n'est pas nécessaire que `T` soit comparable avec lui-même uniquement. En fait, il faut que `T` soit comparable à l'un de ses super types :

```
public static <T extends Comparable<? super T>>
        T max(Collection<T> coll)
```

1.3 Limitations de la généricité

Enfin, il faut garder en tête qu'au moment de la compilation, Java ne peut pas deviner les classes exactes qui vont être utilisées. Aussi les manipulations que l'on peut faire avec ces types génériques anonymes sont limitées. Notamment, les instructions suivantes sont interdites :

```
public class Box<T> {
    public static void g(Object o) {
        if (o instance of T) { /* forbidden! */
            /* ... */
```




```
        }
        T t   = new T ();       /* forbidden! */
        T[] t = new T[5];       /* forbidden! */
        T t   = (E) o;          /* warning: cannot check cast */
    }
}
```

2 Les collections

Les types génériques servent particulièrement dans les collections de données. Dans l'API Java, la plupart de ces structures de données de bases sont déjà implémentées dans le package `java.util`, qui fournit une petite hiérarchie. Notez que toutes ces interfaces sont définies de manière générique : il faut donc spécifier le type qu'elles utilisent lorsqu'on les implémente et qu'on les manipule.

- L'interface de base est `Collection` qui désigne n'importe quel regroupement d'objets.
- Un `Set` est une collection qui n'autorise pas la duplication d'un élément.
- Une `List` est un ensemble d'éléments ordonnés, c'est à dire une séquence d'éléments.
- Une `Queue` est une collection qui conserve un ordre de manipulation, selon une logique FIFO. Elle est particulièrement utile pour traiter des suites d'événements.
- Une `Map` n'est pas une collection : c'est un objet qui associe des clefs à des valeurs. Une table de hachage par exemple, sera vue comme une `Map`.

D'autre part les `Set` et les `Map` peuvent être `Sorted`, c'est à dire que leurs éléments seront constamment ordonnés et que cet ordre sera conservé même après des opérations d'ajout ou de retrait d'éléments.

Quelques implémentations de ces interfaces sont fournies, et dont le nom indique la méthode utilisée pour leur implémentation : `ArrayList`, `LinkedList`, `HashSet`, `HashMap`, ... D'autre part la classe `Collections` (avec un 's') regroupe un grand nombre de fonctions statiques utiles aux manipulations de ces structures de données.

Enfin, il faut noter que l'interface `Collection` étend l'interface `Iterable`. Cette dernière demande juste qu'un objet puisse obtenir un `Iterator` avec la méthode `iterator()`. Un `Iterator` doit implémenter les méthodes `hasNext()` et `next()` (optionnellement `remove()`). Le gros avantage d'un objet `Iterable` est qu'on peut le manipuler avec la construction *foreach* :

```
List<Integer> l = new LinkedList<Integer>();
```

```
/* do something with l */
for (Integer i : l){
    /* do something with i */
}
```

3 Travaux Pratiques

Vous allez utiliser quelques classes de `java.util` et mettre en oeuvre les types génériques dans une classe qui vous sera (sûrement) utile plus tard.

1. Écrivez une classe `Pair<X, Y>` générique qui permet de manipuler des paires associant n'importe quels types de données. Vous doterez cette classe d'un constructeur à deux arguments et des méthodes publiques suivantes : `getFst`, `getSnd`, `setFst` et `setSnd`.
2. Écrivez une méthode statique qui prend en argument une paire d'objets de même classe `U` dont on suppose qu'elle implémente l'interface `Comparable` et qui renvoie le plus grand des deux.
3. Modifiez votre classe pour permettre la comparaison de deux paires, et écrivez une méthode statique de tri lexicographique d'une `List<Pair<U, V>>` où `U` et `V` sont deux classes d'objets `Comparable`.
4. Écrivez une méthode statique qui prend en argument une paire d'objets dont les classes étendent la classe `Number`. Renvoyez la valeur `double` correspondant à la somme des deux nombres.
5. Enfin, écrivez une méthode qui prend en argument une `Collection<Pair<...>>` et renvoie un tableau de type `double[]` contenant les sommes des deux nombres contenus dans chaque paire. Écrivez d'abord une version dont les paires sont uniformément du même type (mais paramétré), puis une version où chaque paire de la collection peut avoir des éléments de n'importe quels sous-types de `Number`.