Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)

2019-2020 - HLIN 505

Cours 6
Classes génériques

Enseignants
Marianne Huchard, Stéphane Bessy, Marie-Laure Mugnier,
Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours. Il peut développer des aspects vus en cours plus succinctement ou inversement vous aurez en cours plus de détails ou d'autres exemples sur certains points. Si vous y relevez des erreurs, n'hésitez pas à nous les signaler afin de l'améliorer.

1 La généricité (polymorphisme paramétrique)

La généricité (polymorphisme paramétrique) permet d'exprimer des algorithmes ou des types complexes paramétrés par d'autres types. Par exemple, dans l'hypothèse où on désirerait développer :

— une pile d'entiers,
— une pile de Strings,
— une pile de Pieces, etc.

Elle permettra de ne pas écrire plusieurs fois des codes proches, qui diffèrent seulement par le type des valeurs empilées.

C'est un mécanisme que l'on trouve dans les langages :

— de programmation comme Java ($\geq$ 1.5), Eiffel, Ada, C++, Haskell,
— de modélisation comme UML.

1.1 Représentation en UML

La figure 1 représente en UML un type Paire paramétré par :

— `A` , le type du premier élément (fst)
— `B`, le type du second élément (snd)

Ce type, qui représente des couples d'éléments de deux types potentiellement différents, est pour UML un "modèle de classes". Pour obtenir une classe, on doit lier les paramètres formels `A` et `B` avec des types réels,

dans l'exemple avec `Integer` et `String` respectivement. Nous l'écrirons en Java un peu plus tard dans ce cours.

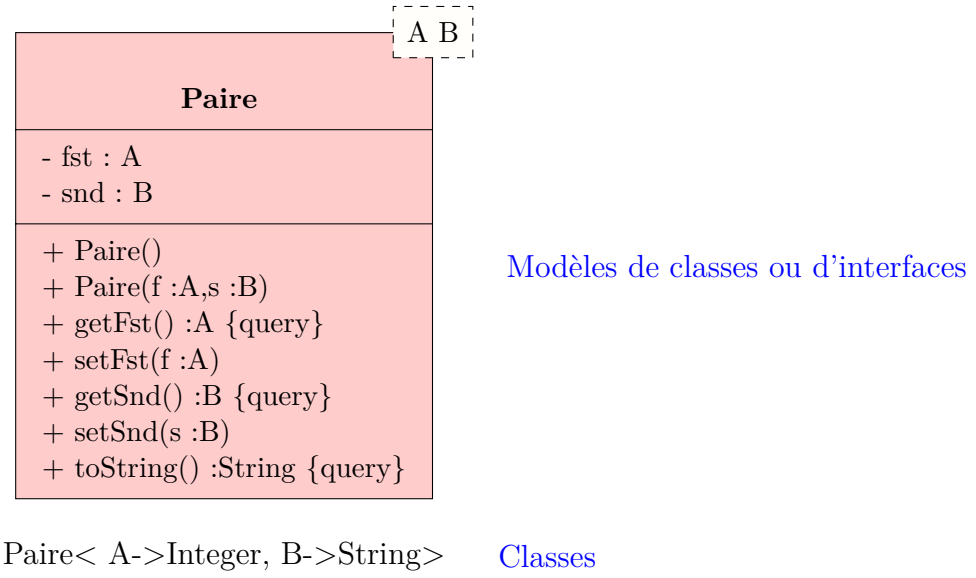

FIGURE 1 – Une pile paramétrée et des instanciations

1.2 Paramétrage d'une classe ou d'une interface en Java

La généricité a été introduite par différentes étapes dans le langage Java.

1991 Naissance du langage Oak (J. Gosling et son équipe, Sun)

1995 Renommage du langage en "Java" et sortie publique du langage

1999 Dépôt d'une JSR (Java Specification Request) par G. Bracha pour l'introduction des génériques en Java

99-04 Quelques langages sont proposés pour compléter Java sur ce point : Pizza, GJ, NextGen, MixGen, Virtual Types, Parameterized Types, PolyJ

2004 Sortie de Java 1.5 (Tiger) - JDK 5.0

- Paramétrage des classes et des interfaces
- L'API des collections devient générique

Avant la généricité, Java utilisait une stratégie (dite "homogène") de représentation de la généricité. Examinons une forme ultra-simplifiée de l'interface de la liste suivant cette stratégie. On y voit apparaître le type `Object` partout où l'on souhaite mentionner le type des objets stockés dans la liste : comme paramètre de l'opération `add` par exemple.

```
public interface List
{
    void add(Object element);
    Object get(int index);
    int size();
}
```

Ce n'est pas idéal car on devra faire parfois :

1. Des *tests de type* : par exemple, si on veut faire une liste ne contenant que des rectangles (construits sur le modèle d'une hypothétique interface `Irectangle`), il faudra contrôler ce que l'on met à chaque fois que l'on fait un `add`.
   Ce contrôle s'effectuera par exemple avec une expression comme
   `if (element instanceof Irectangle)`
   qui met en place un code non extensible. En effet, pour tout nouveau type, il faudra écrire une telle expression de test de type.
2. Des *coercitions* : si on récupère par un `get`, on obtient un objet de type statique `Object`, si on veut appliquer à un objet supposé de type rectangle une méthode comme le calcul du périmètre, il faudra faire un "typecast", par ex., avec une expression comme `((Irectangle) element).perimetre()`.

Une stratégie "hétérogène" aurait consisté à proposer une liste pour tous les types réels existants ou à venir, elle est génératrice de codes de grande taille.

Pour rendre générique l'interface et créer des listes d'objets d'un même type, nous allons :

— Indiquer un type formel (`E`) derrière le nom de l'interface.

— Remplacer `Object` par `E` partout dans l'interface.

```
public interface List<E>
{
    void add(E element);
    E  get(int index);
    int size();
}
```

`<E>` introduit ce que nous appellerons le paramètre de généricité. On déclare que l'on décrit une liste d'éléments de type `E`, ce type étant précisé plus tard. `E` est utilisé ensuite dans l'interface (ce sera la même chose dans une classe) comme si c'était un type ordinaire, connu, alors que c'est un paramètre formel.

Pour créer des listes, il faut ensuite procéder à ce que l'on appelle, suivant les langages de programmation, une *instanciation* (par analogie à la création d'objets), aussi appelée une *invocation* (par analogie à l'invocation d'une méthode dans laquelle on lie les paramètres formels avec des paramètres réels). On peut ainsi déclarer une variable de type liste de rectangles comme une invocation de l'interface générique `List<E>` (première instruction ci-dessous), puis créer une liste par invocation de la classe `ArrayList<E>` (deuxième instruction ci-dessous).

```
List<Irectangle> listeRectangles; // invocation de List<E>
listeRectangles = new ArrayList<Irectangle>(); // invocation de ArrayList<E>
// ou à partir de Java 1.7, la syntaxe en losange
// le compilateur infèrera le type à partir de la déclaration de la variable
listeRectangles = new ArrayList<>(); // invocation de ArrayList<E>
```

1.3 Classes génériques paramétrées par plusieurs paramètres

Nous continuons l'exploration de la généricité paramétrique qui autorise la définition d'algorithmes et de types complexes (classes, interfaces) paramétrés par des types.

On peut avoir plusieurs paramètres de généricité. Nous travaillons par la suite avec la classe `Paire` qui représente des couples d'éléments. Les éléments d'une paire peuvent avoir chacun un type différent. Les

paires sont des objets de base manipulés par certaines structures de données telles que les dictionnaires associatifs. Il existe donc un équivalent de cette classe dans l'API des collections Java, qui s'appelle `Map.Entry<K,V>`.

```
public class Paire<A,B>
{
  private A fst;
  private B snd;
  public Paire(){}
  public Paire(A f, B s){fst=f; snd=s;}
  public A getFst(){return fst;}
  public B getSnd(){return snd;}
  public void setFst(A a){fst=a;}
  public void setSnd(B b){snd=b;}
  public String toString(){return getFst()+''-''+getSnd();}
}
```

Nous savons déjà créer une liste d'étudiants avec une `ArrayList` paramétrée par le type `Etudiant`. Pour créer des paires, nous appliquons le même principe. Remarquez :

- qu'il n'y a pas de nécessité d'écrire une condition vérifiant ce que l'on met dans la paire, si les types des paramètres réels passés au constructeur ne sont pas corrects, la compilation échouera.
- qu'il n'y a pas de nécessité d'utiliser de `cast` (ou coercition) lorsque l'on récupère le premier membre (`fst`) ou le second membre (`snd`) d'une paire.

```
Paire<Integer,String> p1 = new Paire<Integer,String>(9,"plus grand chiffre");
Integer i=p.getFst();
String s=p.getSnd();
System.out.println(p);

Paire<String,Etudiant> pe1
       = new Paire<>("Mohamed", new Etudiant());
Paire<String,Etudiant> pe2
       = new Paire<>("Guillaume", new Etudiant());
```

Vous aurez peut-être remarqué que nous n'avons pas utilisé le type primitif `int`, mais la classe `Integer`. C'est dû à une limitation de la généricité paramétrique propre à Java :

On ne peut pas utiliser de type primitif comme paramètre ! Ce problème n'est pas très grave, il est modéré par un autre mécanisme de Java, *l'autoboxing*, qui convertit de manière transparente les valeurs des types primitifs en instances.

1.4 Le paramétrage de méthodes et la portée des paramètres de généricité

Une règle importante à noter : les paramètres de généricité portent sur les attributs et les méthodes d'instance. Ils ne portent pas sur les attributs et les méthodes de classe (`static`).

1.4.1 Paramétrage d'une méthode de classe (`static`)

Une méthode `static` va donc avoir elle-même ses propres paramètres de généricité. Ci-dessous on voit ainsi le paramétrage de la méthode de classe `copieFstTab` dont le rôle est de copier la première composante d'une paire dans un tableau à un certain indice.

```
class Paire<A,B>
{
    .....
    public static<X,Y> void copieFstTab
                                (Paire<X,Y> p, X[] tableau, int i)
    {tableau[i]=p.getFst();}
    // exercice : code à compléter pour vérifier que l'indice est correct
}
```

1.4.2 Compléments de paramétrage d'une méthode d'instance

Par ailleurs, on peut paramétrer les méthodes d'instance avec des types différents de ceux de la classe générique concernée. Par exemple, la méthode d'instance `memeFst` compare les deux premières composantes de deux paires dont la deuxième composante n'est pas forcément de même type. Elle demande d'introduire un paramètre de généricité pour le type de la deuxième composante. On compare ainsi des paires de `<A, B>` et des paires de `<A, C>` sur la première composante qui est de type `A` dans les deux cas.

```
class Paire<A,B>
{
    .....
    public <C> boolean memeFst(Paire<A,C> p)
    {return p.getFst().equals(getFst());}
}
```

1.4.3 Utilisation de méthodes avec des compléments de paramétrage

Voici une utilisation de ces méthodes. Dans un premier temps on copie le premier membre de la paire `p5` dans un tableau d'entiers à l'indice 0. Puis on compare les premiers membres des paires `p5` et `p2`.

```
    Paire<Integer,String> p5 = new Paire<Integer,String>(9,
                                            "plus grand chiffre");
    Integer[] tab=new Integer[2];

    Paire.copieFstTab(p5,tab,0);
    // l'appel est sur Paire car copieFstTab est static

    Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);

    System.out.println(p5.memeFst(p2));
    // memeFst n'est pas static : elle est appelée sur un objet
```

1.5 Héritage, généricité, invocation

On peut combiner de diverses manières les différents outils de réutilisation que sont l'héritage, la généricité et l'invocation comme l'évoquent les exemples suivants que vous pouvez développer :

— Classe générique dérivée d'une classe non générique

```
class Graphe{}
class GrapheEtiquete<TypeEtiq> extends Graphe{}
```

— Classe générique dérivée d'une classe générique

```
class TableHash<TK,TV> extends Dictionnaire<TK,TV>{}
```

— Classe dérivée d'une instanciation d'une classe générique

```
class Agenda extends Dictionnaire<Date,String>{}
```

— Classe dérivée d'une instanciation partielle d'une classe générique

```
class Agenda<TypeEvt> extends Dictionnaire<Date,TypeEvt>{}
```

Quelques exemples dans l'API des collections de Java :

— public interface Collection$< E >$ extends Iterable$< E >$

— public class Vector$< E >$ extends AbstractList$< E >$

— public class HashMap$< K, V >$ extends AbstractMap$< K, V >$ avec :

K - type des clefs (Keys)

V - type des valeurs (Values)

Nous pourrions redéfinir la classe `PaireSaisissable` comme une sous-classe de `Paire`. Les détails vous sont laissés à titre d'exercice.

1.6 Sous-typage des paramètres de généricité

Quoique `String` soit un sous-type de `Object`, on ne peut pas écrire qu'une liste de `String` est une liste d'`Object`, comme nous le voyons en testant :

```
ArrayList<Object> a = new ArrayList<Object>();
a = new ArrayList<String>(); // provoque une erreur de compilation
```

La raison en est que certaines opérations admises sur une liste d'`Object`, comme `add(new Integer())`, ne peuvent effectivement pas s'appliquer à une liste de `String`. Les opérations qui ne modifient pas la liste ne sont pas en cause, cela provient de celles qui la modifient. Du point de vue théorique, le sous-typage serait autorisé sur des listes immuables.

1.7 Généricité bornée

Lorsque l'on écrit une classe générique, il arrive souvent que l'on attende du type passé en paramètre qu'il respecte un certain nombre de contraintes :

— les objets du type doivent fournir certains services (au sens de fournir certaines opérations, plus rarement au sens d'avoir certains attributs),

— les objets du type correspondent à une certaine abstraction, à un certain rôle.

Par exemple, si on désire munir la classe `Paire<A,B>` d'une méthode de saisie, une *contrainte* que l'on va poser est que les types A et B doivent disposer d'une méthode de saisie également.

La contrainte peut être représentée sous forme d'une classe, d'une classe abstraite ou mieux d'une interface :

```
public interface Saisissable
{
        void saisie(Scanner c);
}
```

Nous introduisons une nouvelle classe `PaireSaisissable` pour éviter des ambiguïtés avec la classe `Paire` du cours précédent. En fin de cours, nous évoquons les relations qu'elles peuvent avoir. Les paires saisissables sont des paires, munies d'une méthode permettant leur saisie. On déclare dans l'entête de `PaireSaisissable` que les types formels doivent spécialiser l'interface `Saisissable`. C'est ce que l'on appellera la "borne" ou la "contrainte". Les paires saisissables sont de plus elles-mêmes saisissables, ce qui vous explique que la nouvelle classe étend également l'interface `Saisissable` (ce n'est pas toujours le cas dans des définitions de bornes).

```
class PaireSaisissable<A extends Saisissable, B extends Saisissable>
                       implements Saisissable
{
  private A fst; private B snd;
  public PaireSaisissable(A f, B s){fst=f; snd=s;}
  public A getFst(){return fst;}
  public B getSnd(){return snd;}
  public void setFst(A a){fst=a;}
  public void setSnd(B b){snd=b;}
  public String toString(){return getFst()+''-''+getSnd();}
  public void saisie(Scanner c){
    System.out.print("Valeur first:"); fst.saisie(c);
    System.out.print("Valeur second:"); snd.saisie(c);}
}
```

Pour utiliser cette classe générique, nous aurons besoin d'un type concret qui réponde à la contrainte (ce n'est pas le cas de `String` ni de `Integer` par exemple). On appelle parfois cette sorte de classe une classe *wrapper* (enveloppe). Ici nous définissons une classe enveloppe de `String`. Elle emballe une chaîne comme attribut, la munit de la méthode attendue et implémente `Saisissable`.

```
public class StringSaisissable implements Saisissable
{
    private String s;
    public StringSaisissable(String s){this.s=s;}
    public void saisie(Scanner c) {s=c.next();}
    public String toString(){return s;}
}
```

Nous pouvons à présent écrire le programme correspondant.

```
Scanner c = new Scanner(System.in);
StringSaisissable s1 = new StringSaisissable("");
StringSaisissable s2 = new StringSaisissable("");
PaireSaisissable<StringSaisissable,StringSaisissable> mp =
      new PaireSaisissable<StringSaisissable,StringSaisissable>(s1,s2);
mp.saisie(c);
```

1.8 Compléments sur les bornes et joker

On peut avoir des contraintes multiples, il y a de nombreux exemples dans l'API de Java.

```
class Paire<A extends Saisissable & Serializable,
                    B extends Saisissable & Serializable>
   {.....}
```

On peut écrire des contraintes récursives, comme le montre l'exemple suivant.

```
public interface Comparable<A>
   {int compareTo(T o);}

public class orderedSet<A extends Comparable<A>>
   {....}
```

Enfin quand la borne n'a pas besoin d'être donnée explicitement, on peut utiliser le caractère joker (wildcard en anglais), qui est représenté par un point d'interrogation.

Il procure tout d'abord un super-type à toutes les instanciations. `Paire<?,?>` est un super-type de `Paire<Integer, String>`. Ce peut être utilisé pour le typage d'une variable, mais comme le type n'est pas complètement précisé, on ne peut ensuite pas faire tout ce que l'on veut, en particulier, on ne peut pas écrire d'expressions dont la vérification nécessite de connaître le paramètre de type.

```
Paire<?,?> p3 = new Paire<Integer, String>();
p3.setFst(12); // NON : setFst dépend du paramètre de type
System.out.println(p3); // OK car l'appel de toString
                        // ne demande pas de connaître le type
```

On peut utiliser les jokers pour simplifier l'écriture de certaines opérations. Par exemple, on peut remarquer que A et B ne sont pas utilisés dans la vérification de l'écriture suivante :

```
  public static<X,Y> void affiche(Paire<X,Y> p)
    {
     System.out.println(p.getFst()+" "+p.getSnd());
    }
```

On peut donc les faire disparaître en introduisant le caractère joker :

```
    public static void affiche(Paire<?,?> p)
    {
     System.out.println(p.getFst()+" "+p.getSnd());
    }
```

1.9 Jeu de contraintes sur les paramètres des méthodes, contrainte avec mot-clef `super`

Nous étudions à présent l'effet des paramètres de types choisis pour les méthodes, et en particulier, nous cherchons à vous montrer comment avoir des méthodes dont le champ d'application ne serait pas artificiellement réduit. Les deux situations que nous allons présenter correspondent respectivement à l'utilisation d'une source de données, puis d'un puits de données.

Examinons une méthode comme la suivante, qui prend dans une liste passée en paramètre le premier élément pour affecter sa valeur à la paire receveur du message (la liste est utilisée comme source de données) :

```
public class Paire<A,B>{
     public void prendListFst(List<A> c)
      {this.setFst(c.get(0));}
....
}
```

Regardons son utilisation dans le contexte d'un programme :

```
Paire<Object,String> p6 = new Paire<Object,String>();
List<Object> lo = new LinkedList<Object>();
List<Integer> li = new LinkedList<Integer>();
lo.add(new Integer(6));
li.add(new Integer(6));
p6.prendListFst(lo);  // ok
p6.prendListFst(li);   // ne fonctionne pas
```

Analysons la raison pour laquelle la dernière instruction ne fonctionne pas. `li` est une `List<Integer>`, ce n'est pas une `List<Object>` et elle ne peut pas être passée en paramètre à `prendListFst(List<Object> c)`. Pourtant, quand on examine ce que réalise la méthode, il n'y a pas d'inconvénient à mettre un `Integer` dans une liste d'`Object`. Pour que cela fonctionne, il faut être capable de dire à la fonction `prendListFst` qu'elle peut admettre comme paramètre une liste d'objets de type A ou d'un sous-type de A.

On peut la réécrire ainsi :

```
public <X extends A> void prendListFst(List<X> c)
    {this.setFst(c.get(0));}
```

Mais comme le paramètre de type X ne sert à rien pour le compilateur (deuxième possibilité), on peut le faire disparaître au profit du joker :

```
public void prendListFst(List<? extends A> c)
    {this.setFst(c.get(0));}
```

Le problème peut exister en sens inverse. Par exemple, si on s'intéresse à une méthode qui écrit le premier membre d'une paire dans une collection (la collection est utilisée comme un puits de données) :

```
public class Paire<A,B>{
public void copieFstColl(Collection<A> c)
    {c.add(this.getFst());}
 ....
 }
```

Elle est trop stricte comme le montre le programme suivant :

```
Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
Collection<Object> co = new LinkedList<Object>();
p2.copieFstColl(co); // Non
```

La dernière instruction ne pourra pas compiler et pourtant mettre un `Integer` dans une collection d'`Object` devrait être possible. On écrit une nouvelle version qui va le permettre :

```
public void copieFstColl(Collection<? super A> c)
    {c.add(this.getFst());}

// dans un main ...
 Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
 Collection<Object> co = new LinkedList<Object>();
 p2.copieFstColl(co);  // A present accepte
```

On trouve un certain nombre d'exemples de ces contraintes complexes sur les paramètres dans l'API de Java, comme :

```
Class AbstractCollection<E>{
    ...
   public boolean addAll(Collection<? extends E> c){...}
}
```

```
Class LinkedBlockingQueue<E>{
   ...
   public int drainTo(Collection<? super E> c){...}
 }
```

1.10 Le principe de l'effacement de type et ses conséquences

En Java, la généricité est mise en œuvre avec la technique de l'effacement de type, qui revient à se ramener à la forme de représentation homogène dont nous avons parlé en début de cours. Cela concerne l'étape de compilation (génération du fichier de bytecode) lors de laquelle :

- toutes les informations de type placées entre chevrons sont effacées
  `class Paire {...}`
- Les variables de types restantes sont remplacées par la borne supérieure (`Object` en l'absence de contraintes)
  `class Paire{private Object fst; private Object snd;..}`
- Insertion de typecast si nécessaire (quand le code résultant n'est pas correctement typé)
  `Paire p = new Paire(9,''plus grand chiffre'');`
  `Integer i=(Integer)p.getFst();`

Cette implémentation a plusieurs conséquences :

— A l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les invocations `p2.getClass()==p5.getClass()` même si p2 est une `Paire<String,Integer>` et p5 une `Paire<Double,Etudiant>`.
— Les variables de type paramétrant une classe ne portent pas sur les méthodes et variables statiques (comme vu auparavant).
— Un attribut statique n'existe qu'en un exemplaire et pas en autant d'exemplaires que d'instanciations. Par exemple, on écrit :

```
public class Paire<A,B>{
        (...) static Integer nbInstances=0;
        public Paire(..){
                ...
                nbInstances++;
        }
        ...
}
Paire<Integer, String> p = new Paire...
Paire<String, String> p2 = new Paire...
```

en résultat, Paire.nbInstances vaut 2 !

— Généralement, on ne peut pas faire de typecast comme `(Paire<Integer,Integer>)p`.

Le type brut est le type paramétré sans ses paramètres, par exemple `ArrayList l;` ou `Paire p`. Il assure l'interopérabilité avec le code ancien (Java 1.4 et versions antérieures). Cependant il faut éviter d'utiliser des types bruts car le compilateur ne fait pratiquement pas de vérification, il vous l'indique d'ailleurs par un warning.

1.11 Synthèse

Nous avons présenté les principaux éléments de la généricité en Java qui est un puissant mécanisme de réutilisation orthogonal et combinable avec l'héritage :
— les interfaces et les classes génériques,
— la notation `<T>`,
— le niveau de paramétrage : attributs et méthodes non static,
— le paramétrage complémentaire des méthodes (static ou non),
— le paramétrage contraint (bornes avec `extends` et `super`),
— le joker ?,
— la finesse dans les paramètres des méthodes pour en élargir le champ d'utilisation,
— le principe de l'effacement de type.