Généricité
(Polymorphisme paramétrique)

Faculté Des Sciences - HAI401I
Modélisation et Programmation par Objets 2

Plan

Motivations

Imaginons que l'on développe :

- Une Pile d'entiers,
- Une Pile de Strings,
- Une Pile de Pieces, etc.

Comment ne pas écrire plusieurs fois des codes approchant, différant seulement par le traitement des types des valeurs empilées ?

Définition

Généricité

Le polymorphisme paramétrique (ou généricité) autorise la définition d'algorithmes et de types complexes (classes, interfaces) paramétrés par des types.

Entier serait un paramètre (réel) de Pile.

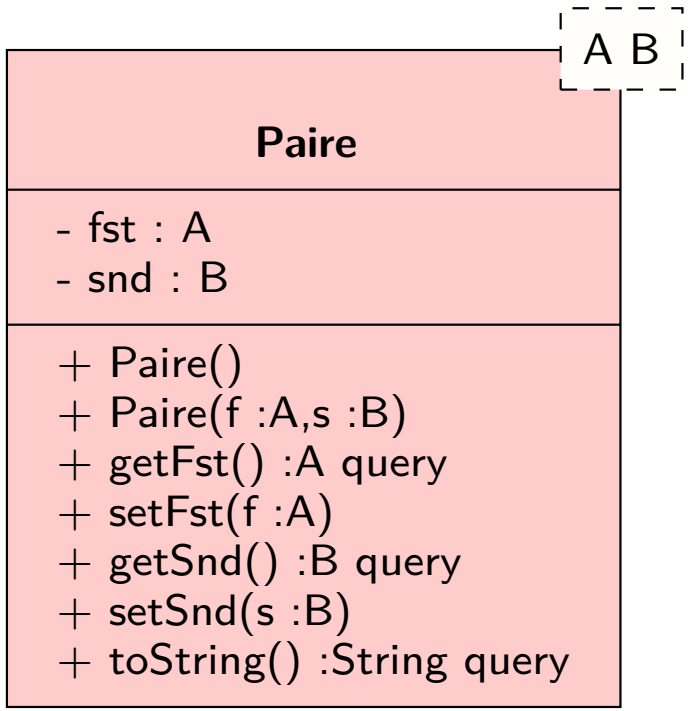
Présence dans les langages :

- de programmation : Java ($\geq$ 1.5), Eiffel, Ada, C++, Haskell, etc.
- de modélisation : UML

Représentation en UML

Un type Paire paramétré par :

- le type du premier élément (fst)
- le type du second élément (snd)



Modèles de Classes

Paire< A->Integer, B->String>

Classes

Motivations

Pour se convaincre : que ferait-on sans ?

- soit une unique copie du code utilisant un type universel, (Object, Object) en Java
  - traduction homogène
- soit une copie spécialisée du code pour chaque situation : (Int, Int), (Int, String), (Int, Piece), etc.
  - traduction hétérogène

Représentation homogène

```
1  public class Paire{
2          private Object fst, snd;
3          public Paire(Object f, Object s) {
4                  fst=f;
5                  snd=s;
6          }
7          public Object getFst() {
8                  return fst;
9          }
10         ...
11 }
```

L'utilisation demande de la coercition (*typecast*).

```
Paire p1 = new Paire("Paques",27);
String p1fst = (String)p1.getFst();
```

Représentation homogène

Inconvénients

- la coercition n'est vérifiée qu'à l'exécution :
  - on peut simplement vérifier par un instanceof ou récupérer l'exception ClassClassException

```
Paire p1 = new Paire("jour",27);
if (p1.getFst() instanceof String)
        String p1fst = (String)p1.getFst();
```

- le code est alourdi, plus difficile à comprendre et à mettre à jour
- la vérification est coûteuse à l'exécution

Représentation hétérogène

```
1  public class PaireStringString{
2          private String fst, snd;
3          public Paire(String f, String s) {
4                  fst=f;
5                  snd=s;
6          }
7          public String getFst()
8                  { return fst;}
9          ...
10 }
11 public class PaireIntString{
12         private Int fst;
13         private String snd;
14         public Paire(Int f, String s) {
15                 fst=f;
16                 snd=s;
17         }
18         public Int getFst()
19                 { return fst;}
20         ...
21 }
```

Représentation hétérogène

Inconvénients

- duplication excessive de code qui est source potentielle d'erreurs lors de l'écriture ou de la modification du programme.
- nécessité de prévoir toutes les combinaisons possibles de paramètres pour un programme donné.

Pour résumer

Objectifs du polymorphisme paramétrique

- éviter des duplications de code
- éviter des typecast et des contrôles dynamiques
- effectuer des contrôles à la compilation (statiques)
- faciliter l'écriture d'un code générique et réutilisable

Un peu d'histoire

Histoire de l'introduction en Java

1995 Naissance de Java

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

Propositions Pizza, GJ, NextGen, MixGen, Virtual Types, Parameterized Types, PolyJ

2004 Java 1.5 (Tiger) - JDK 5.0

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

Plan

Le paramétrage des classes

Le paramétrage des classes
(et des interfaces)

- Une classe générique admet des paramètres formels qui sont des types
- Ces paramètres portent sur les attributs et méthodes d'instance
- Ils ne portent pas sur les attributs et méthodes de classe (static)

La classe paramétrée Paire

```
1  public class Paire<A,B>
2  {
3          private A fst;
4          private B snd;
5          public Paire(){}
6          public Paire(A f, B s) {fst=f; snd=s;}
7          public A getFst() {return fst;}
8          public B getSnd() {return snd;}
9          public void setFst(A a) {fst=a;}
10         public void setSnd(B b) {snd=b;}
11         public String toString() {return getFst()+''-''+getSnd();}
12 }
```

Instanciation/Invocation

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

Mais pas de paramétrage par un type primitif, on ne peut écrire :

```
Paire<int,Piece> = new Paire<int,Piece>(9,new Piece(...));
```

En Java 1.7, syntaxe en losange (le compilateur déduit le type)

```
Paire<Integer,String> p = new Paire<>(9,''plus grand chiffre'');
```

Le paramétrage des classes

Paramétrage des méthodes d'instance

Cas standard

paramétrage par les paramètres de la classe

```
1  public class Paire<A,B>
2  {
3          private A fst;
4          ...
5          public A getFst() {
6                  return fst;
7          }
8          public void setFst(A a) {
9                  fst=a;
10         }
11         ...
12 }
```

Le paramétrage des classes

Paramétrage des méthodes d'instance

En cas de besoin

paramétrage par des paramètres supplémentaires

Comparaison des deux premières composantes de deux paires : la deuxième composante n'est pas forcément de même type.

```
1 public class Paire<A,B>
2 {
3         ...
4         public <C> boolean memeFst(Paire<A,C> p) {
5                 return p.getFst()==this.getFst();
6         }
7         ...
8 }
```

Le paramétrage des classes

Paramétrage des méthodes de classe

paramétrage obligatoire

```
1 public class Paire<A,B>
2 {
3         ...
4         public static<X,Y> void copieFstTab
5                                 (Paire<X,Y> p,  X[] tableau, int i) {
6                 tableau[i]=p.getFst();
7         }
8         ...
9 }
```

Le paramétrage des classes

Paramétrage des méthodes
(une utilisation)

```
1  Paire<Integer,String> p5 = new
2                     Paire<Integer,String>(9,"plus grand chiffre");
3
4  Integer[] tab=new Integer[2];
5
6  Paire.copieFstTab(p5,tab,0);
7
8  Paire<Integer,Integer> p2 = new
9                                    Paire<Integer,Integer>(9,10);
10
11 System.out.println(p5.memeFst(p2));
```

L'effacement de type

Ce que fait le compilateur de notre code :

- Lors de la compilation, 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() ;

L'effacement de type

Conséquences :

- À l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les instanciations

  p2.getClass()==p5.getClass()

- Les variables de type paramétrant une classe ne portent pas sur les méthodes et variables statiques

```
public class Paire<A,B> {
        ...
        public static void copieFstTab(Paire<A,B> p, A[] tableau, int i)
        {
                tableau[i]=p.getFst() ;
        }
        ...
}
```

L'effacement de type

Conséquences :

- Une variable statique n'existe qu'en un exemplaire (et pas en autant d'exemplaires que d'instanciations)

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

Paire.nbInstances vaut 2 !

- Pas d'utilisation dans le contexte de vérification de type instanceOf ou de coercition (typecast), pas de new A() ;

~~(Paire<Integer,Integer>)p~~

L'effacement de type

- Type brut (raw type) = le type paramétré sans ses paramètres
  Paire p7=new Paire() fonctionne !
- Assure l'interopérabilité avec le code ancien (Java 1.4 et versions antérieures)
- Attention le compilateur ne fait pratiquement pas de vérification en cas de type brut et l'indique par un warning

Combinaisons de dérivations et d'instanciations

- 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

- public interface Collection<E> extends Iterable<E>
- public class Vector<E> extends AbstractList<E>
- public class HashMap<K,V> extends AbstractMap<K,V>

  K - type des clefs (Keys)
  V - type des valeurs (Values)

Quelques exemples dans l'API des collections

```
1 public class Stack<E> extends Vector<E>
2 {
3         public Stack();
4         public E push(E item);
5         public E pop();
6         public E peek();
7         public boolean empty();
8         ...
9 }
```

Mariage Polymorphisme paramétrique / héritage

Sous-typage des classes pour un paramètre fixé

Stack<String>
est un sous-type de
Vector<String>

```
Vector<String> pi = new Stack<String>();
```

Mariage Polymorphisme paramétrique / héritage

Pas de sous-typage basé sur celui des paramètres

String sous type d'Object
Stack<String> n'est pas un sous type de Stack<Object>

Pourquoi ?

Certaines opérations admises sur une Pile<Object>, telles que `empile(Object o)`, ne sont pas correctes pour une `Pile<String>` (sauf si les types sont immuables), donc on n'autorise pas la partie barrée :

Stack<Object> pi = ~~new Stack<String>()~~ ;

Plan

Le paramétrage contraint (ou borné)

Pourquoi des contraintes sur les types passés en paramètres :

- lorsque ceux-ci doivent fournir certains services (méthodes, attributs) ;
- plus généralement, pour exprimer qu'ils correspondent à une certaine abstraction.

Le paramétrage contraint (ou borné)

Objectif : munir la classe Paire<A,B> d'une méthode de saisie

Contrainte : les types A et B doivent disposer d'une méthode de saisie également

La contrainte peut être une classe ou mieux une interface

```
1 public interface Saisissable {
2         public abstract void saisie(Scanner c);
3 }
```

Le paramétrage contraint

```
1  public class PaireSaisissable
2          <A extends Saisissable, B extends Saisissable>
3          implements Saisissable
4  {
5          private A fst;
6          private B snd;
7
8          public PaireSaisissable(A f, B s) {fst=f; snd=s; }
9
10         public A getFst() {return fst; }
11         public B getSnd() {return snd;}
12
13         public void setFst(A a) {fst=a;}
14         public void setSnd(B b) {snd=b;}
15
16         public String toString() {return getFst()+''-''+getSnd();}
17
18         public void saisie(Scanner c){
19                 System.out.print("Valeur first:");
20                 fst.saisie(c);
21                 System.out.print("Valeur second:");
22                 snd.saisie(c);
23         }
24 }
```

Le paramétrage contraint

Un type concret qui répond à la demande

```
1  public class StringSaisissable implements Saisissable {
2          private String s;
3          public StringSaisissable(String s) {
4                  this.s=s;
5          }
6          public void saisie(Scanner c) {
7                  s=c.next();
8          }
9          public String toString() {
10                 return s;
11         }
12 }
```

Le paramétrage contraint

Un programme

```
1  Scanner c = new Scanner(System.in);
2
3  StringSaisissable s1 = new StringSaisissable("");
4
5  StringSaisissable s2 = new StringSaisissable("");
6
7  PaireSaisissable<StringSaisissable,StringSaisissable> mp =
8          new PaireSaisissable<StringSaisissable,StringSaisissable>
9          (s1,s2);
10
11 mp.saisie(c);
```

Le paramétrage contraint

Contraintes multiples : Les éléments des paires sont saisissables et sérialisables

```
1 class Paire<A extends Saisissable & Serializable,
2                       B extends Saisissable & Serializable>
3 {...}
```

Le paramétrage contraint

Contraintes récursives

- un ensemble ordonné est paramétré par le type A
- A = les éléments qui sont comparables avec des éléments du même type A

```
1 public interface Comparable<A> {
2         public abstract boolean infStrict(A a);
3 }
4
5 public class orderedSet<A extends Comparable<A>>
6 {...}
```

Le paramétrage par des jokers (*wildcards*)

- Paire<Object,Object>n'est pas super-type de Paire<Integer,String>
  - mais il existe quand même un super-type à toutes les instanciations d'une classe paramétrée
- Le super-type de toutes les instanciations
  - Caractère joker ?
  - Paire < ?, ?> super-type de Paire<Integer, String>

Le paramétrage par des jokers

Utilisation pour le typage d'une variable. Mais tout n'est pas possible

```
Paire<?,?> p3 = new Paire<Integer,String>();
```

~~`p3.setFst(12);`~~

Ne peut être écrit

car appeler `setFst(12)` n'est possible que sur des paires dont le `fst` est `Integer`

cela dépend du paramètre de type et ne peut être contrôlé

```
System.out.println(p3);
```

Est correct

car appeler `toString` sur n'importe quel objet est possible

et ne dépend donc pas du paramètre de type

Le paramétrage par des jokers

Utilisation pour simplifier l'écriture du code

- A et B ne sont pas utilisés dans la vérification de :

```
1 public static<A,B> void affiche(Paire<A,B> p)
2 {
3         System.out.println(p.getFst()+" "+p.getSnd());
4 }
```

- On peut donc les faire disparaître :

```
1 public static void affiche(Paire<?,?> p)
2 {
3         System.out.println(p.getFst()+" "+p.getSnd());
4 }
```

Le paramétrage par des jokers

Utilisation pour élargir le champ d'application des méthodes

- Écrivons une méthode qui prend la valeur de la première composante d'une paire dans une liste (à la première position) :

```
1 public class Paire<A,B>{
2         public void prendListFst(List<A> c) {
3                 setFst(c.get(0));
4         }
5         ...
6 }
```

- Utilisation :

```
1 Paire<Object,String> p6 = new Paire<Object,String>();
2 List<Integer> li = new LinkedList<Integer>();
3 li.add(new Integer(6));
```

~~p6.prendListFst(li);~~

Pourtant il n'y a pas d'erreur sémantique :
un Integer est bien une sorte d'Object mais A=Object≠Integer

Le paramétrage par des jokers

- Pourtant il suffirait que le type des objets dans la liste c soit A ou un sous-type de A dans la méthode

```
1 public class Paire<A,B> {
2         public void prendListFst(List<A> c) {
3                 setFst(c.get(0));
4         }
5         ...
6 }
```

- On réécrit (possibilité 1)

```
1 public <X extends A> void prendListFst(List<X> c) {
2         setFst(c.get(0));
3 }
```

- Mais X ne sert à rien pour le compilateur (possibilité 2)

```
1 public void prendListFst(List<? extends A> c) {
2         setFst(c.get(0));
3 }
```

Le paramétrage par des jokers : contrainte `super`

- extends –> borne supérieure pour le type
- super –> borne inférieure
- Utilisation : puits de données
- Exemple copieFstColl, qui écrit le premier composant d'une paire dans une collection

Version Initiale (méthode de la classe `Paire`)

```
1 public void copieFstColl(Collection<A> c) {
2         c.add(getFst());
3 }
```

Le paramétrage par des jokers : contrainte super

Version Initiale trop stricte

```
1 public void copieFstColl(Collection<A> c) {
2         c.add(getFst());
3 }
```

```
1 Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
2
3 Collection<Object> co = new LinkedList<Object>();
```

~~p2.copieFstColl(co);~~

Pourtant mettre un Integer dans une Collection d'objets ne devrait pas poser de problème

Le paramétrage par des jokers : contrainte super

Nouvelle version : on peut mettre un A dans une collection de A ou d'un type supérieur à A

```
1 public void copieFstColl(Collection<? super A> c){
2         c.add(getFst());
3 }
```

```
1 Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
2
3 Collection<Object> co = new LinkedList<Object>();
4
5 p2.copieFstColl(co);
```

Plan

Synthèse

- Classes génériques
- notation <T>
- niveau de paramétrage : attributs et méthodes non static
- paramétrage complémentaire des méthodes (static ou non)
- paramétrage contraint (bornes avec `extends` et `super`)
- joker ?
- finesse dans les paramètres des méthodes pour en élargir le champ d'utilisation

```
1 Class AbstractCollection<E>{
2     ...
3     public boolean addAll(Collection<? extends E> c){...}
4 }
```

```
1 Class LinkedBlockingQueue<E>{
2     ...
3     public int drainTo(Collection<? super E> c){...}
4  }
```