Plan

Généricité bornée
(Effacement de type, bornes, joker)

Université de Montpellier - Faculté des sciences - HMIN215

L'effacement de type

Ce que fait le compilateur de notre code :

```
1 public class Paire<A,B>{
2         private A fst;
3         private B snd;
4 (...)
5         public Paire(A f, B s)  {fst=f; snd=s;}
6         public A getFst() {return fst;}
7 (...)
8 }
```

- Lors de la compilation, toutes les informations de type placées entre chevrons sont effacées

  class Paire {...}

- Les variables de type 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

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         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<>(s1,s2);
9
10 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 dans une liste (**source de données**) une valeur de la première composante d'une paire :

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

- Utilisation :

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

~~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                 if (!c.empty()) setFst(c.get(0));
4         }
5         ...
6 }
```

- On réécrit (possibilité 1)

```
1 public <X extends A> void prendListFst(List<X> c) {
2         if (!c.empty()) 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         if (!c.empty()) 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 (**puits de données**)

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<>(9,10);
2
3 Collection<Object> co = new LinkedList<Object>();
4
5 p2.copieFstColl(co); // erreur !
```

~~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);
```

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  }
```