Licence MIASHS

INF F5 — Généricité

UGA UFR SHS Université Grenoble Alpes

Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Introduction
- Classe paramétrée
- Méthode générique
- Limites pour les types paramètres
- Effacement
- Généricité et héritage

Introduction

Généricité

- La généricité est un mécanisme des langages de programmation qui permet la définition de programmes paramétrés par des types.
- En Java, depuis la version 1.5, la généricité peut être utilisée dans la définition d'interfaces, de classes et de méthodes.

Motivation – Exemple 1

```
public class Paire {
    private int premier, second;
    public Paire(int a, int b) {
        premier = a; second = b;
    }
    public int getPremier() {
        return premier;
    }
    public int getSecond() {
        return second;
    }
}
```

- Et si on veut pouvoir manipuler des paires de `String`, de `Point`, *etc* ?

Motivation – Exemple 2

```
public class Paire {
    private Object premier, second;
    public Paire(Object a, Object b) {
        premier = a; second = b;
    }
    public Object getPremier() {
        return premier;
    }
    public Object getSecond() {
        return second;
    }
}
```

- Qu'en est-il de l'utilisation de la classe Paire ?

Motivation – Exemple 2

- Avantage : on peut créer des paires pour tout
```
Paire p1 = new Paire("un", "deux");
Paire p2 = new Paire(1, 2);
```
- Inconvénient 1 : trans-typage obligatoire pour récupérer une valeur
```
String s = (String)p1.getPremier();
int i = (Integer)p2.getSecond();
```
- Inconvénient 2 : erreurs de trans-typage détectés à l'exécution et non à la compilation
```
String s = (String)p2.getPremier();
```

Classe paramétrée

Classe paramétrée

```
public class Paire<T> {
    private T premier, second;
    public Paire(T a, T b) {
        premier = a; second = b;
    }
    public T getPremier() {
        return premier;
    }
    public T getSecond() {
        return second;
    }
}
```

- T est un type paramètre qui sera précisé lors de l'instanciation.

Classe paramétrée

- Avantage conservé : on peut créer des paires pour tout

```
Paire<String> p1 = new Paire<String>("un",
                                     "deux");
Paire<Integer> p2 = new Paire<Integer>(1, 2);
```

- Inconvénient 1 éliminé : plus besoin de trans-typage

```
String s = p1.getPremier();
int i = p2.getSecond();
```

- Inconvénient 2 éliminé : erreurs de typage détectés à la compilation

```
String s = p2.getPremier();
```

Classe paramétrée

- Et si on veut faire des paires de types hétérogènes ?
-
```
public class Paire<T, U> {
    private T premier;
    private U second;
    public Paire(T a, U b) {
        premier = a; second = b;
    }
    public T getPremier() {
        return premier;
    }
    public U getSecond() {
        return second;
    }
}
```

Méthode générique

Méthode générique

- On peut aussi utiliser des types paramètres pour la définition de méthodes
- ```
  public class X {
     public <T>  void affiche(Paire<T> p){
        System.out.println(p);
     }
    public static <T> T choix(T a, T b){
        return (int)(Math.random()*2) == 1 ? a : b;
     }
  }
  ```
- Le type réel des paramètres effectifs détermine T à chaque appel de méthode.
  ```

Méthode générique

- … et dans un programme :
- ```
  Paire<String> ps = new  Paire<String>("un",
                                        "deux");
  Paire<Integer> pi = new Paire<Integer>(1, 2);
  X x = new X();
  x.affiche(ps);
  x.affiche(pi);
  Number n = X.choix(new Integer(2),
                     new Double(3.14159));
  ```
- Le type `Number` est déterminé dans la dernière ligne par une inférence de type, c'est le type le plus précis qui est à la fois super-type de `Integer` et de `Double`.
  ```

Limites pour les types paramètres

Limites pour les types paramètres

- On ajoute une méthode à la classe Paire :
- ```
  public class Paire<T> {
      // ...
    public T min(){
        if (premier.compareTo(second) <= 0)
            return premier;
        else
            return second;
     }
  }
  ```
- Erreur signalée par le compilateur : méthode compareTo non définie pour le type T.
  ```

Limites pour les types paramètres

- Il faut donc restreindre les types effectifs possibles pour T :
-
```
public class Paire<T extends Comparable> {
   // ...
  public T min(){
     if (premier.compareTo(second) <= 0)
        return premier;
     else
        return second;
   }
}
```

Limites pour les types paramètres

- Le type limitant peut être une classe ou une interface :

```
public class Paire<T extends Number> {
   // ...
}
```

- Il peut y avoir plusieurs types limitant :

```
public class Paire<T extends A & Comparable> {
   // ...
}
```

Effacement

Effacement

- Les classes paramétrées sont compilées en un type « brut » qui est le seul existant à l'exécution des programmes. Les paramètres de type sont « effacés ». En conséquence :
- ```
  Paire<String> p = new Paire<String>("a", "b");
  boolean b ;
  b = p instanceof Paire; // OK
  b = p instanceof Paire<String>; // Erreur !
  ```
- Le message d'erreur du compilateur indique qu'il faut utiliser la forme brute du type et non la forme paramétrée.
  ```

Effacement

- Lors de l'effacement la classe Paire<T> donne le type brut suivant :
-
```
public class Paire {
    private Object premier, second;
    public Paire(Object a, Object b) {
        premier = a; second = b;
    }
    public Object getPremier() {
        return premier;
    }
    public Object getSecond() {
        return second;
    }
}
```

Effacement

- Lors de l'effacement la classe Paire<T **extends** A> donne le type brut suivant :
-
```
public class Paire {
    private A premier, second;
    public Paire(A a, A b) {
        premier = a; second = b;
    }
    public A getPremier() {
        return premier;
    }
    public A getSecond() {
        return second;
    }
}
```

Effacement

- Un autre exemple de programme

- ```
  Paire<String> p1 = new Paire<String>("un", "deux");
  Paire p2 = new Paire("trois", "quatre");
  p1 = p2;
  p2 = p1;
  ```

- Pas d'erreur mais des avertissements à la deuxième et à la troisième ligne (sécurité de type et référence à un type brut).
  ```

Généricité et héritage

Généricité et héritage

- Soient 2 classes héritant l'une de l'autre, considérons par exemple la classe Point3D qui hérite de la classe Point.
- ```
  Paire<Point3D> pp3 = new Paire<Point3D>(
                                  new Point3D(), new Point3D());
  Paire<Point> pp = new Paire<Point>(
                                      new Point(), new Point());
  pp = pp3; // ERREUR !!!!
  ```
- Il n'y a pas de relations d'héritage entre Paire<Point3D> et Paire<Point>.
- Si cela était possible, nous pourrions écrire
  `pp.setPremier(new Point());`
  et pp3 ne référencerait alors plus une paire de Point3D.
  ```

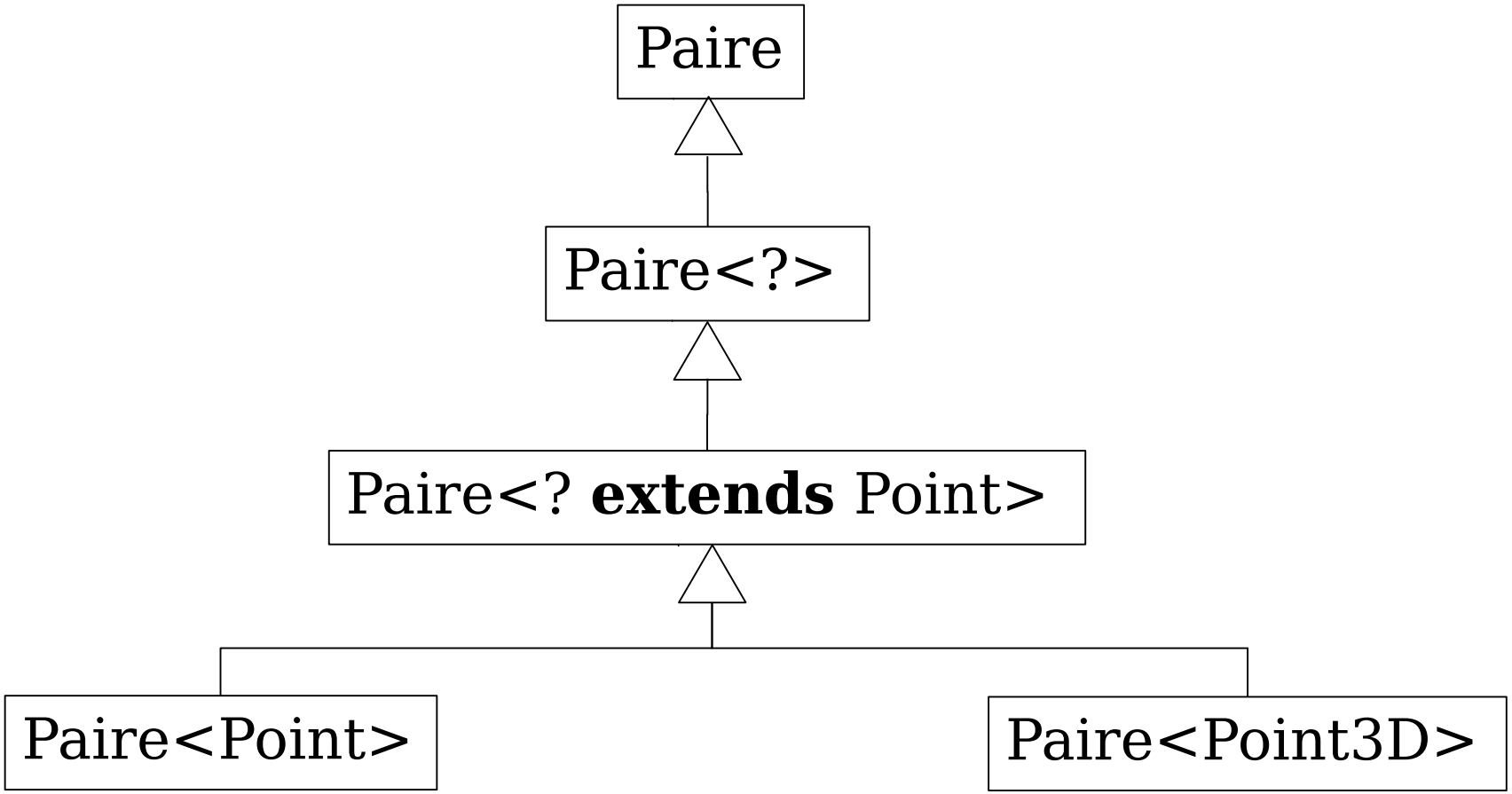
Joker

- Considérons la méthode suivante, ajoutée à la classe X, par exemple.
- ```
  public static void affiche2(Paire<Point> p) {
      System.out.println(p);
  }
  ```
- ```
  Paire<Point> pp = new Paire<Point>( ... );
  Paire<Point3D> pp3 = new Paire<Point3D>( ... );
  X.affiche2(pp); // OK
  X.affiche2(pp3); // NON : pp3 n'est pas Paire<Point>
  ```
- On aurait pu utiliser un joker pour pouvoir activer la méthode avec pp3 :
  ```
  public static void affiche2(Paire<?> p) {
      System.out.println(p);
  }
  ```

Joker

- **public static void** affiche2(Paire<?> p) {
  System.out.println(p);
  }
- Avec ?, la méthode peut-être appliquée à une Paire dont les éléments ont n'importe quel type. Si on veut se limiter à une Paire dont les éléments sont au moins des Point, on peut écrire :
- **public static void** affiche2(Paire<? **extends** Point> p) {
  System.out.println(p);
  }
- On peut limiter le joker par le haut de la hiérarchie d'héritage avec **extends** ou par le bas avec **super**.

Généricité et héritage



Généricité et héritage