ITI 1521. Introduction à l'informatique II

Polymorphisme **paramétré**

by

Marcel Turcotte

Version du 2 février 2020

Préambule

Préambule

Aperçu

Aperçu

Polymorphisme paramétré

Nous verrons que les types génériques permettent la conception de structures de données pouvant sauvegarder des objets de classes diverses sans compromettre l'analyse statique du type des expressions. Ces concepts serviront à la conception de type abstrait de données robustes.

Objectif général :

- Cette semaine, vous serez en mesure de décrire les mécanismes par lesquels on peut concevoir des structures de données génériques en Java sans compromettre l'analyse statique des types.

Une vidéo d'introduction :

- `https://www.youtube.com/watch?v=Ll6s0a1XMuI`

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Utiliser** un type paramétré.
- **Définir** un nouveau type générique.
- **Concevoir** une méthode de classe générique.
- **Reconnaitre** les situations où les types génériques sont avantageux.
- **Expliquer** en vos propres mots en quoi les types génériques donnent lieu à des applications robustes.

Lectures :

- `https://docs.oracle.com/javase/tutorial/java/generics/index.html`

Préambule

Plan du module

Plan

Motivation

Structures de données génériques

Considérons l'exemple de la classe **Pair** à nouveau. Cette fois-ci, nous souhaitons sauvegarder les références de deux objets de type **Time**.

- Quelles sont les **variables d'instance** ?
- Donnez la signature des **méthodes d'instance** ?

Time Pair

```
public class Pair {
  private Time first;
  private Time second;
  public Pair(Time first, Time second) {
    this.first = first;
    this.second = second;
  }
  public Time getFirst() {
      return first;
  }
  public Time getSecond() {
      return second;
  }
}
```

⇒ `new Pair(new Time(14,30), new Time(16,0));`

Shape Pair

```
public class Pair {
  private Shape first;
  private Shape second;
  public Pair(Shape first, Shape second) {
    this.first = first;
    this.second = second;
  }
  public Shape getFirst() {
      return first;
  }
  public Shape getSecond() {
      return second;
  }
}
```

⇒ `new Pair(new Circle(0,0,0), new Rectangle(1,1,1,1);`

Discussion

- **Problème :** on souhaite concevoir une classe **Pair** que l'on pourra utiliser afin de sauvegarder des objets de **divers types**.
- Quel sera le type des **variables** et **paramètres** ?

Pair

```
public class Pair {
  private _______ first;
  private _______ second;
  public Pair(_______ first, _______ second) {
    this.first = first;
    this.second = second;
  }
  public _______ getFirst() {
      return first;
  }
  public  _______ getSecond() {
      return second;
  }
}
```

Caveat : une solution temporaire

- Nous explorons tout d'abord une **première avenue** à l'aide des **concepts vus en classe** jusqu'à maintenant.
- Cette solution était en fait la **seule possible 2004**.
- Cette solution n'offre **aucune sécurité des types**.

Trouvez le type de la variable first !

- On doit pouvoir sauvegarder la référence de **tout objet** dans la variable **first**.
  - Quel est son type ?
- Quel est le type le plus général en Java ?
- Quelle est la classe la plus générale en Java ?

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(_____ first, _____ second) {
    this.first = first;
    this.second = second;
  }
  public _____ getFirst() {
      return first;
  }
  public  _____ getSecond() {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(Object first, Object second) {
    this.first = first;
    this.second = second;
  }
  public _____ getFirst() { // type of the return value?
      return first;
  }
  public _____ getSecond() {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(Object first, Object second) {
    this.first = first;
    this.second = second;
  }
  public Object getFirst() {
      return first;
  }
  public Object getSecond() {
      return second;
  }
}
```

Pair

- Comment utilise-t-on cette classe ?

```
Pair p;

String a;
a = "King";
String b;
b = "Edward";

p = new Pair(a, b);

a = p.getFirst();
b = p.getSecond();
```

- Voyez-vous un problème avec ces décclarations ?

Pair

```
Pair p;

String a;
a = "King";
String b;
b = "Edward";

p = new Pair(a, b);

a = (String) p.getFirst();
b = (String) p.getFirst();
```

- La classe **Object** est plus générale que la classe **String** ! Chaque fois qu'un élément est retiré d'une structure de données, il faut forcer le type de la valeur de retour.
- Il y aura une erreur lors de l'exécution si l'objet n'est pas de type **String**.
- Avec ce **forçage de type**, on se **prive** de l'aide que le **compilateur** pourrait nous apporter.

Types génériques

Types génériques

Java 1.5 a été une mise à jour majeure *. Elle a introduit le concept de **types génériques** («***generics***»).

```
public class Pair<T> {
    ...
}
```

La déclaration de la classe comporte un **paramètre (formel) de type**. C'est le type des objets qui seront sauvegardés par les objets de la classe **Pair**.

*C'était en 2004.

Utilisation

La valeur de **T** doit être spécifiée lors de la déclaration d'une variable.

```
Pair<String> name;
Pair<Integer> range;
```

ainsi qu'au moment de créer un objet :

```
name = new Pair<String>("Hillary", "Clinton");
```

```
Integer min;
min = new Integer(0);

Integer max;
max = new Integer(100);

range = new Pair<Integer>(min, max);
```

Types génériques : une solution gagnante

- Des victoires sur deux fronts :
  - Une **seule** implémentation de la classe **Pair** que l'on réutilise dans de multiples contextes (on y sauve les références d'objets de types variés).
  - Tout ça, sans sacrifier la **validation des types** à l'étape de la **compilation**.

Types génériques

- Un **type générique** est un type possédant un **paramètre formel de type**.
- Un **type paramétré** est une instanciation d'un type générique avec un **paramètre actuel de type**.

Type génériques

Définir un type générique.

- Un **type générique** est un **type référence** ayant un ou plusieurs paramètres de type.
- Un **type générique** c'est une **classe** ayant un ou plusieurs paramètres de type.

```java
public class Pair<T> {

    private T first;
    private T second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
    }
    public void setFirst(T value) {
        first = value;
    }
    public void setSecond(T value) {
        second = value;
    }
}
```

Types génériques

- Qu'avons-nous **obtenu** ?
  - Des types «génériques» et «robustes».

```
Pair<Integer> range;
range = new Pair<Integer>(new Integer(0),new Integer(10));

Integer i;
i = range.getFirst();
```

Des erreurs de compilation !

L'objet déclaré de type **Pair<Integer>**, ne peut être utilisé que pour sauvegarder la référence d'un objet de type **Integer**.

```
range.setFirst("Voila");
```

Produira une **erreur de compilation**, et c'est ce qu'on souhaite !

```
Test.java:20: setFirst(java.lang.Integer)
in Pair<java.lang.Integer> cannot be applied to
(java.lang.String)
        range.setFirst("Voila");
              ^
1 error
```

Encore des erreurs de compilation !

De même, une variable référence ayant le type de compilation suivant **Pair<Integer>** ne peut désigner un objet dont le type paramétré est **Pair<String>**.
L'énoncé

```
range = new Pair<String>("Hillary", "Clinton");
```

produira une erreur de compilation,

```
Test.java:22: incompatible types
found   : Pair<java.lang.String>
required: Pair<java.lang.Integer>
        range = new Pair<String>("Hillary", "Clinton");
                ^
1 error
```

```java
public class Pair<X,Y> {

    private X first;
    private Y second;

    public Pair(X first, Y second) {
        this.first = first;
        this.second = second;
    }
    public X getFirst() {
        return first;
    }
    public Y getSecond() {
        return second;
    }
    public void setFirst(X value) {
        first = value;
    }
    public void setSecond(Y value) {
        second = value;
    }
}
```

Type paramétré

Lorsqu'on utilise un **type générique**, ici **Pair**, on doit fournir des **valeurs de type** pour chacun **paramètre formel de type**.

```
public class Test {
    public static void main(String[] args) {

        Pair<String,Integer> p;

        String attribute;
        attribute = new String("height");

        Integer value;
        value = new Integer(100);

        p = new Pair<String,Integer>(attribute, value);
    }
}
```

Quiz : est-ce que ces énoncés sont valides ?

```
Pair<String,Integer> p;
```

```
p = new Pair<String,Integer>();
```

```
p.setFirst("session");
```

```
p.setSecond(12345);
```

Quiz : est-ce que ces énoncés sont valides ?

```java
public class T1 {
    public static void main(String[] args) {

        Pair<String,Integer> p;

        p = new Pair<Integer,String>();

    }
}
```

```
// > javac T1.java
// T1.java:6: incompatible types
// found   : Pair<java.lang.Integer,java.lang.String>
// required: Pair<java.lang.String,java.lang.Integer>
//       p = new Pair<Integer,String>();
//               ^
// 1 error
```

Quiz : est-ce que ces énoncés sont valides ?

```java
public class T2 {
    public static void main(String[] args) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>();
        p.setFirst(12345);
        p.setSecond("session");
    }
}
```

```
T2.java:5: setFirst(java.lang.String) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (int)
        p.setFirst(12345);
         ^
T2.java:6: setSecond(java.lang.Integer) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (java.lang.String)
        p.setSecond("session");
         ^
2 errors
```

Quiz : est-ce que ces énoncés sont valides ?

```java
public class T3 {
    public static void main(String[] args) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>("session", 12345);
        Integer s = p.getFirst();
    }
}
```

```
// > javac T3.java
// T3.java:8: incompatible types
// found   : java.lang.String
// required: java.lang.Integer
//       Integer s = p.getFirst();
//                                ^
// 1 error
```

Types génériques

Les génériques servent à définir des structures de données «générales» tout en détectant les erreurs de types au moment de la compilation !

Types génériques et les interfaces

- L'**interface** aussi nous permet de définir un **type référence**.
- L'**interface** aussi peut avoir un ou plusieurs paramètres de type.

Interface comme type générique : Comparable

```java
public interface Comparable {
    int compareTo(Comparable other);
}
```

```java
public interface Comparable<T> {
    int compareTo(T other);
}
```

Time : sans type paramétré

```
public class Time implements Comparable {

    private int timeInSeconds;

    public int compareTo(Comparable obj) {

        Time other = (Time) obj;
        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Time : avec un type paramétré

```
public class Time implements Comparable<Time> {

    private int timeInSeconds;

    public int compareTo(Time other) {

        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Discussion

```java
public interface Comparable {
    int compareTo(Comparable other);
}
```

```java
public interface Comparable<T> {
    int compareTo(T other);
}
```

- **Sans** type générique, le contrat que nous avions nous demandait d'implémenter une méthode **compareTo** dont le paramètre était de type **Comparable**. Il fallait donc ensuite forcer le type, ce qui pourrait causer une erreur d'exécution.
- **Avec** un type générique, le contrat spécifie qu'il faut implémenter une méthode **compareTo** dont le type est le même que celui de la classe en question, ici soit **Time** ou **Person**.

Person : sans type paramétré

```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {
        Person other = (Person) obj;
        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Person : avec type paramétré

```java
public class Person implements Comparable<Person> {

    private int id;
    private String name;

    public int compareTo(Person other) {

        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

```java
public class Pair<String> {

    private String first;
    private String second;

    public String getFirst() {
        return first;
    }
    public String getSecond() {
        return second;
    }
    public void setFirst(String value) {
        first = value;
    }
    public void setSecond(String value) {
        second = value;
    }
}
```

Méthodes génériques

Méthode de classe générique

```
public class Test {
    public static <E> void display(E[] xs) {
        for (int i=0; i<xs.length; i++) {
            System.out.println(xs[i]);
        }
    }
}
```

```
Random generator;
generator = new Random();

Integer[] xs;
xs = new Integer[5];

for (int i=0; i<5; i++) {
    xs[i] = generator.nextInt(100);
}

display(xs);
```

```
> java TestDisplay
78,95,53,21,7
```

```
String[] words;
words = new String[7];

words[0] = "alpha";
words[1] = "bravo";
words[2] = "charlie";
words[3] = "delta";
words[4] = "echo";
words[5] = "foxtrot";
words[6] = "golf";

display(words);
```

```
> java TestDisplay
alpha,bravo,charlie,delta,echo,foxtrot,golf
```

Utils.max

```
public class Utils {

    public static <T extends Comparable<T>> T max(T a, T b) {
        if (a.compareTo(b) > 0) {
            return a;
        } else {
            return b;
        }
    }

}
```

Appel à une méthode de classe générique

- L'**appel** à une méthode de classe générique **ne nécessite** (généralement) aucun élément de syntaxe supplémentaire.
- Le compilateur fait l'inférence du type automatiquement.

```
Integer i1, i2, iMax;

i1 = new Integer(1);
i2 = new Integer(10);

iMax = Utils.max(i1, i2);

System.out.println("iMax = " + iMax);
```

Ici, le compilateur infère que le type du paramètre doit être **Integer**.

Appel à une méthode de classe générique

```
String s1, s2, sMax;

s1 = new String("alpha");
s2 = new String("bravo");

sMax = Utils.max(s1,s2);

System.out.println("sMax = " + sMax);
```

Ici, le compilateur infère que le type du paramètre doit être **String**.

Spéficier la valeur du paramètre de type

On peut tout de même **spécifier** le type :

```
iMax = Utils.<Integer>max(i1, i2);
sMax = Utils.<String>max(s1, s2);
```

```java
public class SortAlgorithms {

    public static <T extends Comparable<T>>
                        void selectionSort(T[] xs) {

        for (int i = 0; i < xs.length; i++) {

            int min = i;

            for (int j = i+1; j < xs.length; j++) {
                if (xs[j].compareTo(xs[min]) < 0) {
                    min = j;
                }
            }

            T tmp = xs[min];
            xs[min] = xs[i];
            xs[i] = tmp;
        }
    }

}
```

Une nouvelle syntaxe pour les boucles

Java 1.5 avait aussi introduit une nouvelle syntaxe pour les boucles.

```
int[] xs;
xs = new int[] {1, 2, 3};

int sum = 0;

for (int x : xs) {
    sum += x;
}
```

Prologue

Résumé

- Un **type générique** est un **type référence** ayant un **paramètre formel de type**.
- Un **type paramétré** est une instanciation d'un type générique avec un **paramètre actuel de type**.
- Les types génériques permettent la création de structures de données «générales» ; générales en ce sens qu'on y sauvegarde la référence d'objets de divers types **sans compromettre la validation des types**.

Prochain module

- **Type abstrait de données (TAD)** : les piles

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa