Programowanie obiektowe

Wykład 6: Elementy programowania generycznego

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.2
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Przykład wprowadzający

- Rozważmy stworzenie metody, która będzie liczyła modę dla podanego zbioru.

Wyliczanie mody

```
1  public double calculateTendency() {
2    double modeValue, maxCount;
3    maxCount = 0;
4    modeValue = Double.NaN;
5    double[] unique = uniqueValues(timeSeries);
6    for (double value: unique) {
7      int count = 0;
8      for (double occurrence: timeSeries) {
9        if (value == occurrence) count++;
10     }
11     if (count > maxCount) {
12       maxCount = count;
13       modeValue = value;
14     }
15   }
16   return modeValue;
17 }
```

- Jakie dostrzegamy ograniczenia przedstawionego przykładu?.
- Obliczenia są ograniczone do typu `double`.
- Rozwiązaniem jest *programowanie generyczne*.

Programowanie generyczne

- Programowanie generyczne pozwala pisać kod działający na różnych typach danych.
- Pozwala także ograniczać zakres dopuszczalnych typów.
- Eliminuje rzutowanie co ogranicza niebezpieczną konwersję zwężającą.
- W rezultacie otrzymujemy uniwersalny kod z kontrolą typów.

Notacja

- Możemy tworzyć generyczne klasy, interfejsy i metody.
- Używamy nawiasów <> do oznaczania elementów generycznych.
- Wewnątrz nawiasów umieszczamy oznaczenie generowanego elementu zgodnie z konwencją.

E	element zbioru,
T	typ,
K,V	klucz, wartość,
N	liczba,
S,U,V	kolejne typy.

Generyczne wyliczanie mody

- Przedefiniujmy wyliczanie mody.

Wyliczanie mody

```
1   public static <T> T calculateMode(ArrayList<T> timeSeries) {
2     T modeValue;
3     int maxCount;
4     maxCount = 0;
5     modeValue = null; //null zamiast Double.NaN
6       for (T value : timeSeries) { //Typ ArrayList zamiast Array
7         int count = 0;
8         for (T occurrence : timeSeries) {
9           if (value.equals(occurrence)) count++; //equals
                zamiast ==
10        }
11        if (count > maxCount) {
12          maxCount = count;
13          modeValue = value;
14        }
15      }
16    return modeValue;
17  }
```

- Wyliczanie działa dla dowolnego typu <T>.
- Wymuszone zostały pewne zmiany w kodzie (linie 5, 6 i 9).
- Co się stanie gdy zamiast mody będziemy liczyć średnią?

Ograniczenie zmiennych typowych

- Czasami konieczne jest nałożenie pewnych ograniczeń na zmienne typowe klas i metod.
- Dzieje się tak wtedy, gdy chcemy uzyskać dostęp do specyficznego interfejsu, wspólnego dla obiektów na których działa algorytm.
- Lub gdy chcemy ograniczyć działanie do pewnego zbioru klas.
- Wówczas możemy ograniczyć typ w wyrażeniu generującym.
  - Wyrażenie `<U extends Number>` stwierdza, że argumenty muszą być obiektami rozszerzającymi klasę `Number`.
  - Możemy wymusić zarówno rozszerzanie klasy jak i implementację interfejsów.
  - Wyrażenie `<T extends C1 & I2 & I3>` akceptuje tylko obiekty rozszerzające klasę `C1` i implementujące interfejsy `I2` i `I3`.

Generyczne wyliczanie średniej

- Utworzymy kod generyczny wyliczający średnią przy następujących założeniach.
  - Średnia będzie wartością typu `double`
  - Średnią da się wyliczyć tylko dla danych numerycznych

Wyliczanie średniej

```
1  public <T extends Number> double calculateMean(ArrayList<T>
       timeSeries) {
2    double meanValue = 0;
3
4    for (T value: timeSeries) {
5      meanValue+= value.doubleValue(); //Allowed by  limitation
         to Number
6    }
7
8    return meanValue/timeSeries.size();
9  }
```

- Ze względu na konieczność przeprowadzenia operacji + i / oraz konwersji do typu `double` kod nie mógłby zostać stworzony bez ograniczenia typu.

Klasy generyczne

- Klasy generyczne są zazwyczaj tworzone jako kontenery dla innych obiektów.
- Pozwalają przechowywać obiekty jednego typu, ale działają tak samo dla dowolnego typu

IntegerBox

```
public class IntegerBox {
 private Integer integer;
  public void set(Integer
        integer) {
  this.integer = integer;
 }
 public Integer get() {
  return integer; }
 }
```

Klasa może przechowywać tylko typ `Integer`.

Box

```
public class Box {
 private Object object;
 public void set(Object
       object) {
  this.object = object;
 }
 public Object get() {
  return object; }
 }
```

Typ przechowywany w instancji klasy może się zmieniać.

Box<T>

```
public class Box<T> {
 private T t;
 public void set(T t) {
  this.t = t;
  }
 public T get() {
  return t;
 }
}
```

Przechowywany typ `T` jest określany przy tworzeniu instancji.

Unikanie rzutowania

- Przyjrzyjmy się kontenerowi `ArrayList`, który może być definiowany generycznie.
- Utworzenie kontenera dla klasy `Object` powoduje, że nie wiemy jakie obiekty znajdują się wewnątrz i nie mamy dostępu do poszczególnych interfejsów

`ArrayList<Object>`

```
List list = new ArrayList(); list.add("hello");
((String)list.get(0)).toUpperCase();
```

- Określenie typu `String` pozwala na kontrolowanie typu przechowywanych obiektów i bezpiecznie korzystanie z interfejsów.

`ArrayList<String>`

```
List<String> list = new ArrayList<>(); list.add("hello");
list.get(0).toUpperCase();
```

Interfejsy generyczne

- Oprócz klas można tworzyć interfejsy generyczne

ArrayList<Object>

```
public interface Pair<K, V> {
  public K getKey();
  public V getValue();
}
```

- Klasy generyczne mogą implementować te interfejsy

ArrayList<Object>

```
public class OrderedPair<K, V> implements Pair<K, V> {
 private K key;
 private V value;
 public OrderedPair(K key, V value) {
  this.key = key; this.value = value; }
 public K getKey() { return key; }
 public V getValue() { return value; }
}
```

Tworzenie instancji

- Klasy generyczne inicjujemy podając typy i wartości

Tworzenie instancji klas generycznych

```
Box<Integer> integerBox = new Box<>();
OrderedPair<String, Integer> p1;
p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String> p2;
p2 = new OrderedPair<>("hello", "world");
OrderedPair<String, Box<Integer>> p;
p= new OrderedPair<>("primes", new Box<Integer>(...));
```

Metody generyczne

- Metody generyczne mogą być definiowane nie tylko w klasach generycznych.
- Metody definiują swoje własne parametry

Definicja metody generycznej

```
public class Util {
  public static <K, V> boolean compare(Pair<K, V> p1,
      Pair<K, V> p2) {
   return p1.getKey().equals(p2.getKey()) &&
       p1.getValue().equals(p2.getValue());
   }
}
```

- Wywołanie metody wymaga określenia typów parametrów

Wywołanie metody generycznej

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Implementacja interfejsu generycznego

- Uprzednio stworzyliśmy klasę implementującą interfejs stosując niebezpieczne rzutowanie.

```
1  public class Employee extends Person implements
       Comparable {
2      private double salary;
3      @Override
4      public int compareTo(Object o) {
5          return (int)(salary-((Employee)o).salary);
6      }
7  }
```

- Teraz możemy zastąpić ją bezpieczną formą implementującą generyczny interfejs Comparable<T>

```
1  public class Employee extends Person implements
       Comparable<Employee> {
2      private double salary;
3      @Override
4    public int compareTo(Employee e) {
5          return (int)(salary-e.salary);
6      }
7  }
```

Automatyczne określanie typu

- Rozważmy metodę generyczną z parametrem <String>

```
ArrayAlg.<String>getMiddle("Jan", "Chryzostom",
    "Pasek");
```

- Możemy wywołać ją bez parametru, z określeniem typu na podstawie argumentów

```
ArrayAlg.getMiddle("Jan", "Chryzostom", "Pasek");
```

- Jednakże metoda będzie powodować błędy, jeżeli typ nie będzie jednoznacznie określony

```
ArrayAlg.getMiddle(2.72, 3, 3.14);
```

- Argumenty są typów `Double` i `Integer` przez co mogą być zrzutowane zarówno do `Number` lub `Comparable`.

Ograniczenia typów generycznych

- Typy generyczne mają pewne ograniczenia użycia.
  - Nie można używać typów prostych jako parametrów typowych.
    - Nie można stworzyć `Pair<int,double>` da się tylko `Pair<Integer,Double>`
  - Ograniczone sprawdzanie typów w czasie działania programu.
    - Dla obiektów `Pair<Double,Double> doublePair` i `Pair<Integer,Integer> integerPair` wyrażenie `doublePair.getClass()== integerPair.getClass()` zwróci `true`.
  - Nie można tworzyć tablic generycznych.
    - Nie zadziała `PairPair<Integer,Double>[] table = new Pair<Integer,Double>[10];`
- Więcej ograniczeń w [Horstmann, 2016].


PROGRAMOWANIE GENERYCZNE
POWRÓCI W PRZYSZŁYM SEMESTRZE

Bibliografia

[Horstmann, 2016] Horstmann, C. S. (2016).
Java. Podstawy.
Helion.