Typy uogólnione.

Robert A. Kłopotek
r.klopotek@uksw.edu.pl

Wydział Matematyczno-Przyrodniczy. Szkoła Nauk Ścisłych, UKSW

23.03.2017

Typy uogólnione (1/2)

- **Typy uogólnione** (generyczne) są to typy parametryzowane typem
- Ich własności w języku Java w znaczący sposób odbiegają od koncepcji szablonów znanej z C++, pomimo podobieństwa zapisu.
- Są dostępne od Java 1.5
- Na typach generycznych opiera się implementacja biblioteki kolekcji (kontenerów) języka Java, biblioteka `java.util`
- W Java 8 wprowadzono następujące modyfikacje:
  - Optymalizacja współbieżności kolekcji
  - Otwarcie kolekcji na wyrażenia lambda

Typy uogólnione (2/2)

- Typy generyczne nie są zgodne z klasowo-obiektowym paradygmatem programowania
- Stanowią więc rozszerzenie pierwotnych możliwości języka Java
- Wspierają one paradygmat uogólniony nazywamy paradygmatem generycznym
- Zalety:
  - kontrola typów na poziomie kompilacji, a nie dopiero w trakcie działania programu
  - brak potrzeby rzutowania
  - konstruowanie ogólnych algorytmów

Koncepcja typów uogólnionych w Java

- Dość wysoki stopień uogólniania kodu był i jest w Javie dostępny poprzez:
  - zagwarantowane dziedziczenie klasy Object
  - implementację interfejsów
  - mechanizmy refleksji (czyli np. dynamiczne, w fazie wykonania programu, odwołania do pól i metod klas)
- wprowadzenie generics (typów uogólnionych) dodaje ułatwienia w postaci
  - unikania konwersji zawężających
  - tworzenia bardziej czytelnego kodu
  - wykrywania błędów w fazie kompilacji i unikania wyjątku ClassCastException

Zalety typów uogólnionych w kolekcjach (1/2)

- bezpieczeństwo typów - możemy przetrzymywać obiekty tylko jednego typu
- rzutowanie typów nie jest konieczne
  - bez typów generycznych

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

  - z typami uogólnionymi

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

Zalety typów uogólnionych w kolekcjach (2/2)

- sprawdzanie w momencie kompilacji - problemy z przetrzymywanie obiektów niewłaściwych typów nie pojawią się podczas działania

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Błąd kompilacji!
```

Typy parametrów - konwencja oznaczeń

- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

Ogólna klasa reprezentującą dowolne pary (1/2)

```
class ParaObj {
  Object first;
  Object last;

  public ParaObj(Object f, Object l) {
    first = f;
    last = l;
  }

  public Object getFirst() { return first; }
  public Object getLast()   { return last; }

  public void setFirst(Object f) { first = f; }
  public void setLast(Object l) { last = l; }
}
```

Ogólna klasa reprezentującą dowolne pary (2/2)

```
ParaObj po = new ParaObj("Ala", new Integer(3));
System.out.println(po.getFirst() + "␣" + po.getLast());

// Problem 1
// konieczne konwersje zawężające
String name = (String) po.getFirst();
int nr = (Integer) po.getLast();
po.setFirst(name + "␣Kowalska");
po.setLast( new Integer(nr + 1));
System.out.println(po.getFirst() + "␣" + po.getLast());

// Problem 2
// możliwe błędy
po.setLast("kot");
System.out.println(po.getFirst() + "␣" + po.getLast());

// Błąd może być wykryty w fazie wykonania
// późno, czasem w innym module
Integer n = (Integer) po.getLast();    // ClassCastException
```

Para - typ uogólniony (1/3)

```
class Para<S, T> {
  S first;
  T last;

  public Para(S f, T l) {
    first = f;
    last = l;
  }

  public S getFirst() { return first; }
  public T getLast()   { return last; }

  public void setFirst(S f) { first = f; }
  public void setLast(T l) { last = l; }
}
```

Para - typ uogólniony (2/3)

```
Para<String, String> p1
  = new Para<String, String> ("Jan", "Kowalski");
Para<String, Data> p2
  = new Para<String, Data> ("Jan␣Kowalski", new Date());
Para<Integer, Integer> p
  = new Para<Integer, Integer>(1,2);  // autoboxing działa;

//autoboxing
Para<String, Integer> pg = new Para<String, Integer>("Ala", 3);
System.out.println(pg.getFirst() + "␣" + pg.getLast());
String nam = pg.getFirst();          // bez konwersji!
int m = pg.getLast();                // bez konwersji!
pg.setFirst(name + "␣Kowalska");
pg.setLast(m+1);                     // autoboxing
System.out.println(pg.getFirst() + "␣" + pg.getLast());
```

Para - typ uogólniony (3/3)

```
Para<String, Integer> p1
  = new Para<String, Integer>("Ala", 3);
Para<String, String> p2
  = new Para<String, String>("Ala", "Kowalska");

// "Raw Type"
Class p1Class = p1.getClass();
System.out.println(p1Class); //class Para
Class p2Class = p2.getClass();
System.out.println(p2Class); //class Para

// zwraca tablicę metod deklarwoanych w klasie
Method[] mets = p1Class.getDeclaredMethods();
for (Method m : mets) System.out.println(m);
//public java.lang.Object Para.getFirst()
//public java.lang.Object Para.getLast()
//public void Para.setFirst(java.lang.Object)
//public void Para.setLast(java.lang.Object)
```

Para - podsumowanie

- jest tylko jedna klasa Para dla wszystkich instancji klasy sparametryzowanej Para<S, T>; typ wyznaczany przez tę klasę nazywa się typem surowym ("raw type"),
- z definicji klasy Para zniknęły wszystkie parametry typu i zostały zastąpione przez Object; ten mechanizm nazywa się czyszczeniem typów ("type erasure") ,
- ponieważ jest tylko jedna klasa Para - np. zmienne reprezentowane przez pola statyczne są wspólne dla wszystkich instancji typu sparametryzowanego

Możemy (w definicjach uogólnionych klas i metod)

- podawać je jako typy pól i zmiennych lokalnych
- podawać je jako typy parametrów i wyników metod
- dokonywać jawnych konwersji do typów oznaczanych przez nie (ale to będzie tylko ważne na etapie kompilacji, po to by uniknąć błędów niezgodności typów, natomiast nie uzyskamy w fazie wykonania faktycznych konwersji np. zawężających)
- wywoływać na rzecz zmiennych oznaczanych typami sparametryzowanymi metody klasy Object (i ew. właściwe dla klas i interfejsów, które stanowią tzw. górne ograniczenia danego parametru typu)

Nie możemy (w definicjach uogólnionych klas i metod) (1/2)

- tworzyć obiektów typów sparametryzowanych (new T() jest niedozwolone, no bo na poziomie definicji generics nie wiadomo co to konkretnie jest T)
- używać operatora instanceOf (z powodu j.w.)
- używać ich w statycznych kontekstach (bo statyczny kontekst jest jeden dla wszystkich różnych instancji typu sparametryzowanego)
- używać ich w literałach klasowych
- wywoływać metod z konkretnych klas i interfejsów, które nie są zaznaczone jako górne ograniczenia parametru typu (w najprostszym przypadku tą górną granicą jest Object, wtedy możemy używać tylko metod klasy Object)

Nie możemy (w definicjach uogólnionych klas i metod) (2/2)

- używać typów sparametryzowanych przy tworzeniu tablic (podając je jako typ elementu tablicy)
- w obsłudze wyjątków (bo jest to mechanizm fazy wykonania)
- w literałach klasowych (bo oznaczają typy fazy wykonania)

Ograniczenia parametrów typu

- istnieje możliwość ograniczenia zakresu klas/interfejsów jaki dany typ uogólniony będzie przyjmował
- służy do tego słowo extends

```
public class Box<T> {
  private T t;
  public void set(T t) {
       this.t = t;
  }
  public <U extends Number> void inspect(U u){
       System.out.println("T: " + t.getClass().getName());
       System.out.println("U: " + u.getClass().getName());
  }
  public static void main(String[] args) {
       Box<Integer> integerBox = new Box<Integer>();
       integerBox.set(new Integer(10));
       integerBox.inspect("some text"); // Błąd kompilacji!
  }
}
```

Wielokrotne ograniczenie parametrów typu

- zapisujemy jako: `<T extends B1 & B2 & B3>`
- jesli na liście ograniczeń jest klasa to musi być wymieniona jako pierwsza (może być tylko jedna klasa a reszta interfejsy), np:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

- jeśli A nie będzie wymieniona jako pierwsza dostaniemy błąd kompilacji

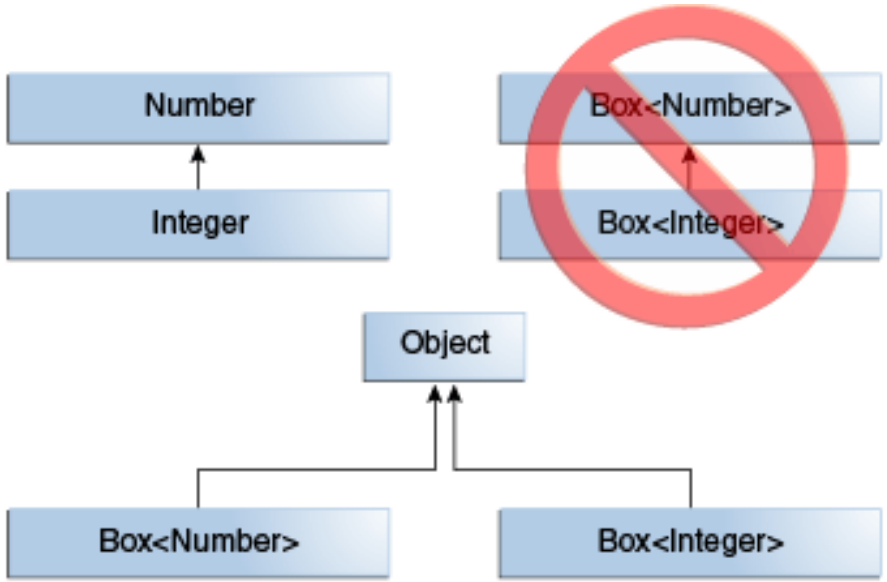
```
class D <T extends B & A & C> { /* ... */ }
// Błąd kompilacji!
```

Generyki - dziedziczenie i podtypy

- w języku Java jeśli mamy zależność klasa Integer dziedziczy po Number to możemy na nią rzutować bez problemów, np:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));    // OK
box.add(new Double(10.1));   // OK
```

- problem pojawia się wtedy, jeśli chcielibyśmy rzutować na siebie typy generyczne, np. Box<Number> i Box<Integer>



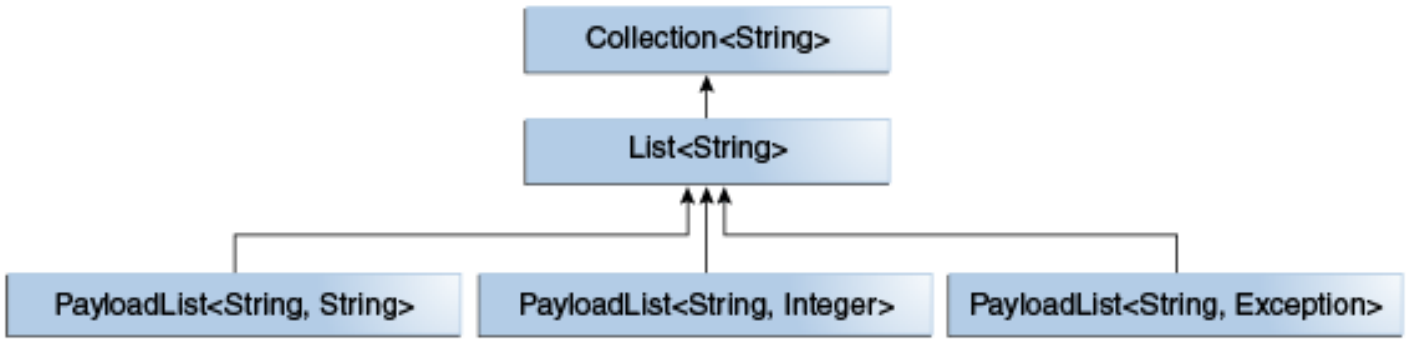
Podtypy - jak to zrobić? (1/2)

- możemy tworzyć podtypy oraz hierarchię typów za pomocą rozszerzania klas i interfejsów generycznych
- jednym z przykładów jest klasa
  `ArrayList<E> implements List<E>` oraz
  `List<E> extends Collection<E>`
- możemy wtedy zdefiniować własne klasy lub interfejsy generyczne je rozszerzające, np:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

Podtypy - jak to zrobić? (2/2)

- parametryzacje typu `PayloadList<E,P> extends List<E>`, takie jak
  - `PayloadList<String,String>`
  - `PayloadList<String,Integer>`
  - `PayloadList<String,Exception>`
- są podtypem `List<String>`



Parametry uniwersalne (wildcards)

- Jeśli `ArrayList<Integer>` i `ArrayList<String>` nie są podtypami `ArrayList<Object>` - to jak stworzyć metodę wypisującą zawartość dowolnej listy `ArrayList` ?
- Do tego służą parametry uniwersalne (wildcards) - oznaczenie "?"
- Są trzy typy takich parametrów:
  - ograniczone z góry <? extends X> - oznacza "wszystkie podtypy X"
  - ograniczone z dołu <? super X> - oznacza "wszystkie nadtypy X"
  - nieograniczone <?> - oznacza "wszystkie typy"

Wariancja typów uogólnionych

- Typ sparametryzowany C<T> jest **kowariantny** względem parametru T, jeśli dla dowolnych typów A i B, takich, że B jest podtypem A, typ sparametryzowany C<B> jest podtypem C<A> (kowariancja - bo kierunek dziedziczenia typów sparametryzowanych jest zgodny z kierunkiem dziedziczenia parametrów typu)
- Kowariancję uzyskujemy za pomocą symbolu <? extends X>
- Przykład: `List<? extends Number>` jest nadtypem wszystkich typów sparametryzowanych, gdzie parametrem typu jest Number albo typ pochodny od Number.

Kowariancja typów uogólnionych

- Typ sparametryzowany C<T> jest **kontrawariantny** względem parametru T, jeżeli dla dowolnych typów A i B, takich że B jest podtypem A, typ sparametryzowany C<A> jest podtypem typu sparametryzowanego C<B> (kontra - bo kierunek dziedziczenia jest przeciwny).
- Kontrawariancję uzyskujemy za pomocą symbolu <? super X>.
- Przykład: Integer jest podtypem Number, a `List<Number>` jest podtypem `List<? super Integer>`, wobec czego możemy podstawiać:

```
List<? super Integer> list = new ArrayList<Number>;
```

Biwariancja typów uogólnionych

- **Biwariancja** oznacza równoczesną kowariancję i kontrawariancję typu sparametryzowanego
- Przykład: `ArrayList<?>` jest nadtypem `ArrayList<? extends Integer>` i nadtypem dla `ArrayList<? super Integer>`

Metody sparametryzowane i konkludowanie typów

- Parametryzacji mogą podlegać nie tylko klasy czy interfejsy, ale również metody.
- Definicja metody sparametryzowanej ma postać:

```
specyfikatorDostępu [static] <ParametryTypu>
        typWyniku nazwa( lista parametrów) {
        // ...
        }
```

- Argumenty typów są określane na podstawie faktycznych typów użytych przy wywołaniu metody
- Proces wyznaczania aktualnych argumentów typów nazywa się konkludowaniem typów (ang. type inferring)

Konkludowanie typów - przykład

```
public class Metoda {
 public static <T extends Comparable<T>> T max(T[] arr) {
  T max = arr[0];
  for (int i=1; i<arr.length; i++)
    if (arr[i].compareTo(max) > 0) max = arr[i];
  return max;
 }
  public static void main(String[] args) {
    Integer[] ia = { 1, 2, 77 };
    int imax = max(ia);// w wyniku konkluzji T staje się Integer

    Double[] da = {1.5, 231.7 };
    double dmax = max(da);// w wyniku konkluzji T staje się Double
    System.out.println(imax + "␣" + dmax);
  }
}
```

Konflikty typów uogólnionych - przykład (1/4)

Mamy strukturę: `class Person`,
`class VipPerson extends Person` oraz
`class SuperPerson extends VipPerson`

```
static void addPersonToList(List<Person> list) {
  list.add(new Person("Arek"));   // ok
  list.add(new VipPerson("Arek", 1)); // ok
}

static void addAnyToList(List list) {
  list.add(new Person("Arek"));   // ok
  list.add(new VipPerson("Arek", 1)); // ok
  list.add(new Integer(1));   // ok
}

static void printList(List list) { // dowolna lista
  System.out.println(list.toString());
}
```

Konflikty typów uogólnionych - przykład (2/4)

```
  void printFromList(List<? extends Person> list) {
    Person person = list.get(0);
    System.out.println(person);
  }
  void addPersonToList2(List<? extends Person> list) {
    list.add(new Person("Arek")); // ->
    // błąd bo nie wiadomo czy to lista
    // VipPerson, Person, czy jakaś inna klasa dziedzicząca po Pers
    list.add(new VipPerson("Arek", 1));  // też błąd
  }
void addPersonToList3(List<? super VipPerson> list) {
    list.add(new Person("Arek")); // ->
    // błąd. Co jeśli jest to List<VipPerson>?
    // Wtedy nie można dodać Person
    list.add(new VipPerson("Arek", 1));  // ok
    list.add(new SuperPerson("Arek"));  // ok
    list.add(new Object());  // błąd
    Object object = list.get(0);
    // typ Object. Nie wiemy jakiego typu jest
    // obiekt w liście. Może VipPerson, może
    // Person... kto wie!?
}
```

Konflikty typów uogólnionych - przykład (3/4)

```
void addPersonToList4(List<? super Person> list) {
  list.add(new Person("Arek"));  // ok
  list.add(new VipPerson("Arek", 1));  // ok
  Object object = list.get(0);
  // tylko object, bo w sumie nie wiadomo jakiego jest typu
}
public static void copy(List<? extends VipPerson> src,
                        List<? super VipPerson> dest) {
  for (int i = 0; i < src.size(); i++)
    dest.add(src.get(i));
}
```

Konflikty typów uogólnionych - przykład (4/4)

```
public static void main(String[] args) {
  List<Person> list = new ArrayList<>();
  addPersonToList(list);
  // sposoby deklaracji
  List<Person> peopleList = new ArrayList<>();  // ok
  List<Person> list3 = new ArrayList<Person>();  // ok
  List<Person> list2 = new ArrayList<VipPerson>();  // błąd
  addAnyToList(peopleList); // ok
  List<VipPerson> vipList = new ArrayList<>();  // ok
  addPersonToList(vipList); // ->
  // błąd (nie każdy Person jest VipPerson)
  List<SuperPerson> superList = new ArrayList<>();  // ok
  copy(superList, peopleList);// ok
}
```

Pytania?