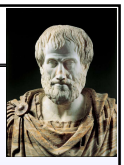


JTP: Typy Rodzajowe/Generyczne w Javie (Java Generics)

dr hab. Piotr Kosiuczenko
prof. WAT

1

Typy Generyczne

- Typy generyczne wywodzą swoją nazwę od genra, co jest liczbą mnogą łacińskiego genus, tj. rodzaj.
- Typy generyczne w Javie zapewniają bezpieczeństwo związane z tym, że określeniem typu wyrażeń i możliwością sprawdzenia typu przez parser.
- W czasie kompilacji jest możliwość sprawdzenia poprawności danej formuły pod względem typów:

```
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(10);

v.add("TEN");
```

2

Klasy Generyczne

- Niektóre typy danych i klasy mogą być zaimplementowane niezależnie od konkretnego typu swoich atrybutów i metod, tzn. mogą posiadać parametry będące typami (polimorfizm).
- `ArrayList<T>` - kiedy taki parametr zostaje zastąpiony konkretnym typem, taka klasa może być skompilowana.
- Klasa, która posiada parametry, których wartościami są typy, jest nazywana generyczną (parametryczną lub rodzajową).
- W Javie typy generyczne są zaimplementowane poprzez bezpieczne rzutowanie.

3

Klasy Generyczne

```
public class Pair<T> {
  private T first;
  private T second;
  public Pair(T f, T s) {
    first = f;
    second = s;
  }
  public Pair() {
    first = new T();
    second = new T();
  }
  public T getFirst() {
    return first;
  }
  public T getSecond() {
    return second;
  }
  public void setFirst(T f) {
    first = f;
  }
  public void setSecond(T s) {
    second = s;
  }
```

Parametrem musi być klasa - nie może być typ prymitywnym.

Niepoprawna deklaracja

metoda parametryczna

4

Klasy Generyczne

```
public String toString() {
  return "[" + first.toString() + ", " +
               second.toString() + "]";
}

public boolean equals(Object o) {
  if(!(this.getClass() == o.getClass())) return
                    false;
  Pair p = (Pair)o;
  return (this.first.equals(p.first) &&
   this.second.equals((p.second));
}
}
```

Wiemy, że o jest tej samej klasy, co this.

5

Dlaczego Klasy Generyczne? Impl. bez rodzajów

```
static void swapAll(Collection c) {
  for (Object el : c) {
    Object t = ((Pair)el).getFirst();
    ((Pair)el).setFirst(((Pair)el).getSecond());
    ((Pair)el).setSecond(t);
  }
}
```

musimy dopasować typ

6

Dlaczego Klasy Generyczne?

```
static <T> void swapAll(Collection<Pair<T>> c) {
  for (Pair<T> el : c) {
    T t = el.getFirst();
    el.setFirst(el.getSecond());
    el.setSecond(t);
  }
}

  Collection c = new ArrayList<Pair<Integer>>();
  c.add(new Object());
  c.swapAll;
```

implementacja z parametrami

spowoduje błąd w czasie kompilacji

Konstruktory

- Niektóre typy danych i klasy mogą być zaimplementowane niezależnie od konkretnego typu swoich atrybutów i metod, tzn. mogą posiadać parametry będące typami.

```
Pair<Object> p = new Pair<Object>(null, null);
```

- Konstruktory nie mogą być deklarowane dla parametrycznego typu `T`:

```
T o = new T();
Pair<T> p = new Pair<T>();
T[] a = new T[5];
```

Jeśli `T` jest parametrem, to deklaracje te spowodują błąd w czasie kompilacji.

- Możliwa jest za to automatyczna konwersja typów:

```
Pair<Integer> ip = new Pair<Integer>(3, 5)
```


Ograniczenie parametrów

- Czasem jest konieczne ograniczenie parametrów:

```
class RestrictedPair<T extends Color> {
      ...
}
```


Parametryczne klasy

- Czasem jest konieczne powiązanie parametrow:
- ~~`public class MyVector<S extends Comparable> extends Vector`~~
- ~~`public class MyVector<S extends Comparable> extends Vector<T>`~~
- `public class MyVector<S extends Comparable<?>> extends Vector<S>`

Niepoprawne, mimo że daje się skompilować.

użycie białej karty

Konstruktory

- Deklaracja metody generycznej z nowym parametrem `T`:

```
public static <T> T genericM(T[] a)
```

- Wywołanie metody z aktualnym parametrem:

```
Object s = ArrayList.<String>genericM(a);
```

konkretny typ

Kowariancja i Kontrawariancja

- Zalozmy, że klasa `Dog` rozszerza klasę `Animal`.
- Czy następujący kod jest poprawny?

```
ArrayList<Dog> dogs = new ArrayList<Dog>();
ArrayList<Animal> animals = dogs;
```

problem z wkładaniem obiektow klasy `Animal` do wektora zawierającego tylko psy.

- `ArrayList<Dog>` nie może być traktowane jako podklasa `ArrayList<Animal>` (tj. nie ma tzw. kowariancji.)

Typy Generyczne: Literatura Dodatkowa

- Java Tutorial: https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Literatura Dodatkowa

- Podstawowe informacji: https://www.w3schools.com/java/java_lambda.asp
- Szersze omówienie: https://www.geeksforgeeks.org/lambda-expressions-java-8/
- Oracle: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

JTP: Wyrażenia Lambda

dr hab. Piotr Kosiuczenko
profesor nadzw. WAT

Wyrażenia Lambda

- **Ogólna postać**
  (parameters) -> { lambda body }
- Przykład dla zerowej liczby parametrów:
  () -> System.out.println("Hello, world.")
- Przykład dla 1 parametru, identyczność
  a -> a
- 1 parametr, inkrementacja
  a -> ++a
- 2 parametry, mnożenie
  *(a, b) -> a * b*

Wyrażenia Lambda

- Wyrażenia lambda w językach programowania są funkcjami anonimowymi, tj. funkcjami, które nie posiadają nazwy
- W sensie implementacyjnym są zastosowaniem operacji do argumentów
- Za pomocą wyrażeń lambda można definiować funkcje posiadające nazwy
- Jeśli jednak nie ma potrzeby używania takiej funkcji wielokrotnie, to wyrażenia lambda wystarczają

Wyrażenia Lambda

- 2 parametry z informacją o typach
  (String name, double x) -> name + „ = „ + x
- Blok kodu
  (x, y) -> { return x * y; }

Wyrażenia Lambda: funkcje nazwane

```
public class Calculator {
    interface IntegerMathU {
      int operation(int x);
    }
    interface IntegerMathB {
        int operation(int x, int y);

    }
    public int operateBinary(int a, int b, IntegerMathB op) {
        return op.operation(a, b);
    }

    public int operateUnary(int a, IntegerMathU op) {
        return op.operation(a);
    }
```


Wyrażenia Lambda: działanie na listach

```
    LinkedList<String> l = new LinkedList<String>();
    l.add("Hello1"); l.add("Hello2"); l.add("Hello3");
    LinkedList<String> l1 = new LinkedList<String>();
    l.forEach(x -> l1.add(exclaim.modifyString(x)));
    System.out.println(l1.toString());
  }
}
```


Wyrażenia Lambda: funkcje nazwane

```
    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMathU increment = (x) -> x + 1;
        IntegerMathB addition = (a, b) -> a + b;
        IntegerMathB subtraction = (a, b) -> a - b;
        System.out.println("20++ = " +
                myApp.operateUnary(20, increment));
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));

    }
}
```


JTP: Zbiórka Śmieci

dr hab. Piotr Kosiuczenko
prof. WAT

Wyrażenia Lambda: działanie na listach

```
import java.util.LinkedList;
interface StringProcessingFunction {
  String modifyString(String str);
}

public class Main {
    public static void printFormatted(String str,
  StringProcessingFunction format) {
    String result = format.modifyString(str);
    System.out.println(result);
    }

 public static void main(String[] args) {
    StringProcessingFunction exclaim = (s) -> s + "!";
    StringProcessingFunction ask = (s) -> s + "?";
    printFormatted("Hello", exclaim);
    printFormatted("Hello", ask);
```


Maszyna wirtualne: zbiórka śmieci

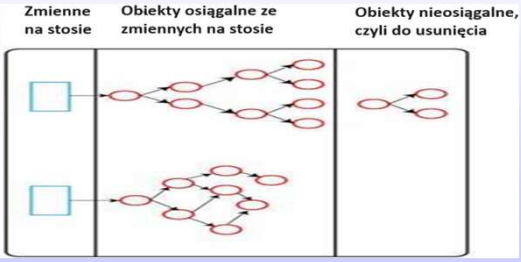
1. W trakcie wykonania programu tworzone są różne obiekty, nieraz w dużej ilości tak, że skład zostaje zapełniony.
2. Potrzebne jest usuwanie niepotrzebnych już obiektów.
3. W Javie służy temu mechanizm zbiórki śmieci usuwający „niepotrzebne” już obiekty.
4. Co znaczy niepotrzebne?
5. JVM tego oczywiście „nie wie”.

Maszyna wirtualne: zbiórka śmieci, cd.

1. Łatwiej na początek zada pytanie: które obiekty mogą być jeszcze potrzebne?
2. Za potrzebne można uznać obiekty, do których odnośniki są wartościami zmiennych programowych znajdujących się na stosie.

Maszyna wirtualne: zbiórka śmieci cd.

1. Jakie jeszcze obiekty mogą ewentualnie być potrzebne?
2. Takie obiekty, do których jest ścieżka dostępu wychodząc z pewnych obiektów powyżej wspomnianych.



Maszyna wirtualne: zbiórka śmieci cd.

1. Jakie algorytmy mogą być użyte do znalezienia wszystkich takich obiektów?
2. Algorytmy przeszukiwania grafów skierowanych, jak np. przeszukiwanie wgłąb i przeszukiwanie wszerz.
3. Idea jest taka, żeby usunąć obiekty, do których nie ma dostępy wychodząc ze zmiennych znajdujących się na stosie.

Maszyna wirtualne: zbiórka śmieci cd.

1. W praktyce stosuje się jednak inne algorytmy, bo użycie wspomnianych byłoby zbyt kosztowne czasowo przy dużych grafach znajdujących się na składzie.
2. Algorytmy i sposoby zbiórki śmieci stanowią odrębną tematykę.
3. Zbiórka śmieci jest prowadzona automatycznie przez JVM.
4. Można ją jednak uruchomić za pomocą metody `System.gc();`

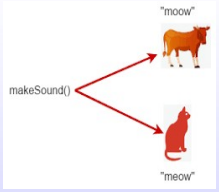
Maszyna wirtualne: zbiórka śmieci cd.

```
public class GCTest {
  public static void main(String[] args) {
    GCTest x1=new GCTest();
    GCTest x2=new GCTest();
    System.out.println(x1);
    System.out.println(x2);
    x1=null;
    System.out.println("Start Garbage Collection");
     // metoda gc() uruchomia zbiórkę śmieci
    System.gc();
    System.out.println(x1);
    System.out.println(x2);
  }
}
```


JTP: Polimorfizm

dr hab. Piotr Kosiuczenko
prof. WAT

Polimorfizm



31

Polimorfizm

1. Polimorfizm, z greckiego, znaczy "wiele form" i występuje, gdy mamy wiele klas, które są ze sobą powiązane poprzez dziedziczenie.
2. Jak wspomnieliśmy w poprzednim rozdziale, dziedziczenie pozwala nam dziedziczyć atrybuty i metody z innej klasy.
3. Polimorfizm wykorzystuje te metody do wykonywania różnych zadań. Pozwala nam to wykonywać jedną czynność na różne sposoby.

32

Polimorfizm: rodzaje

Polimorfizm w Javie dzieli się głównie na dwa typy:

1. Polimorfizm czasu kompilacji, zwany też polimorfizmem statycznym. Ten rodzaj polimorfizmu osiąga się poprzez przeciążanie nazw metod.
2. Polimorfizm czasu wykonywania. Tutaj stosujemy wiązanie dynamiczne – omówiliśmy to zagadnienie wcześniej.

33

Polimorfizm: przeciążenie

1. Polimorfizm czasu kompilacji osiąga się poprzez przeciążanie/przeładowanie nazwy metody.
2. Gdy istnieje parę metod o tej samej nazwie, ale różnych parametrach, mówi się, że te ich nazwy są przeciążone.
3. Metody mogą być przeciążone z powodu różnej liczby lub nieporównywalnych typów parametrów, np.
   `m()`, `m(int x)`, `m(String x)`.
   W takim przypadku wyklucza to nadpisanie jednej metody przez drugą.

34

Polimorfizm: przeciążenie nazwy metody

```
public class Sum {
    public static int sum(int x, int y) {
    return (x + y);
    }
    // przeciążenie, pierwszej nazwy, bo są tu 3 parametry
    public static int sum(int x, int y, int z)  {
        return (x + y + z);
    }
    // przeciążenie, pierwszej nazwy, bo param. typu double
    public static double sum(double x, double y)  {
        return (x + y);
    }
    public static void main(String args[])  {
        System.out.println(sum(10, 20));
        System.out.println(sum(10, 20, 30));
        System.out.println(sum(10.5, 20.5));
    }
```


35

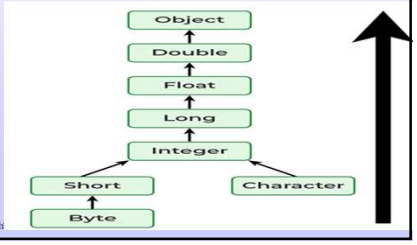
Polimorfizm: ustalanie typu metody

Ustalanie typu wiązanej metody jest na podstawie typów parametrów

Podklasa ma wyższy priorytet niż nadklasa

Ustalając priorytet, kompilator wykonuje następujące kroki:

- Konwersja danego typu do wyższego typu (pod względem zakresu) w tej samej rodzinie (jeśli np. nie ma dostępnego typu danych Long dla typu danych Integer, to wyszuka typ danych Float).
- Konwersja typu do następnej wyższej rodziny.

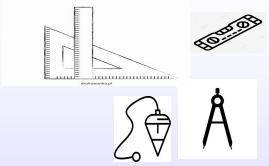


36

Polimorfizm czasu wykonania

1. Polimorfizm czasu wykonania osiąga się w Javie za pomocą algorytmu look up, o tym mówiliśmy wcześniej.
2. Najpierw jednak należy rozwiązać problem przeciążenia nazwy, jeśli występuje – o tym mówiliśmy ostatnio. Kiedy nazwa, w szczególności sygnatura, są ustalone stosujemy algorytm look up.

JTP: Pryncypia języka Java

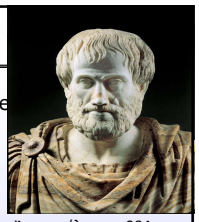


dr hab. Piotr Kosiuczenko
prof. WAT

Pryncypia języka Java

- Abstrakcja
  - Obiekty
  - Asocjacja
  - Zakapslowanie
  - Interfejsy
- Polimorfizm
  - Dziedziczenie
  - Przeciążenie
  - Typy rodzajowe (Java generics)
- Compile once run everywhere - kompilowalność na JVM
- Automatyczna zbiórka śmieci

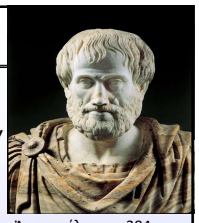
Pryncypia języka Java: Abstrakcja



Ἀριστοτέλης, ur. 384 p.n.Ch., zm. 322 p.n.Ch

- Pojęcie abstrakcji (stgr. ἀφαίρεσις - oderwanie, odjęcie, ...) pochodzi od Arystotelesa, z jego Metafizyki.
- Abstrakcja polegający na pominięciu nieistotnych własności obiektów.
- Abstrakcja bywa często wielopoziomowa.

Abstrakcja w sensie informatycznym



Ἀριστοτέλης, ur. 384 p.n.Ch., zm. 322 p.n.Ch

- Abstrakcja polega na ograniczeniu zakresu własności manipulowanych obiektów do tych, które są istotne dla interakcji między nimi, przetwarzania danych, etc., w szczególności algorytmów.
- Ukrywa implementację i złożoność danych.
- W Javie, prezentuje tylko sygnaturę nie ujawniając wewnętrznej funkcjonalności.
- Pomaga uniknąć powtarzającego się kodu.
- Daje programistom elastyczność w zakresie zmian implementacji abstrakcyjnego zachowania.

Pryncypia języka Java: Abstrakcja typów

- Zakapslowanie i podział funkcjonalności na metody służą abstrakcji.
- Częściową abstrakcję można osiągnąć za pomocą klas abstrakcyjnych.
- Wyższy stopień abstrakcji można osiągnąć za pomocą interfejsów.
- Podział na klasy, dziedziczenie i delegacja realizuje zasadę programowania DRY - Don't Repeat Yourself – nie powtarzaj się, tj. nie powtarzaj kodu.

Pryncypia języka Java: Zakapslowanie

- Zakapslowanie pomaga w zabezpieczeniu danych, umożliwiając ochronę danych przechowywanych w klasie przed dostępem całego systemu.
- Jak sama nazwa wskazuje, chroni ona wewnętrzną zawartość klasy jak kapsuła.
- Konkretnie, zakapslowanie ogranicza bezpośredni dostęp do elementów atrybutów klasy.
- Atrybuty są ustawione jako prywatne.
- Dostęp do nich jest poprzez metody `get` i ewentualnie `set`.

Pryncypia języka Java: Dziedziczenie

- Klasa (klasa podrzędna) może rozszerzyć inną klasę (klasę nadrzędną), dziedzicząc jej cechy.
- Dziedziczenie umożliwia utworzenie klasy podrzędnej.
- Klasa podrzędna dziedziczy pola i metody klasy nadrzędnej.
- Klasa podrzędna może nadpisywać wartości i metody klasy nadrzędnej.
- Może również dodawać do swojego rodzica nowe dane/atrybuty i nowe metody.
- Poprawia możliwość ponownego wykorzystania kodu.

Pryncypia języka Java: Polimorfizm

- Ta sama nazwa metody jest używana w różnych kontekstach.
- Statyczny polimorfizm w Javie jest realizowany przez przeciążanie metod.
- Dynamiczny polimorfizm w Javie jest realizowany przez nadpisywanie metod.

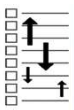
Pryncypia języka Java: Polimorfizm

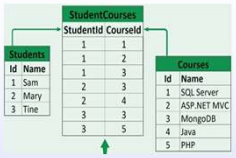
- Ta sama nazwa metody jest używana kilka razy
- Różne metody o tej samej nazwie mogą być wywoływane z obiektu.
- Dynamiczny polimorfizm w Javie jest realizowany przez nadpisywanie metod.
- Statyczny polimorfizm w Javie jest realizowany przez przeciążanie metod.

Programowanie obiektowe versus proceduralne i funkcjonalne

proceduralne

W programowaniu imperatywnym/proceduralnym koncentrujemy się na poleceniach i procedurach, a operujemy na zmiennych.





relacyjne

Programowanie relacyjne ma dwa zasadnicze składniki: relacje i kwerendy.

programowanie w logice

$P(t_1,...,t_n)$

A :- B1, ..., Bn

funkcjonalne

λ

W programowaniu funkcjonalnym podstawową jednostką jest funkcja.