Nowoczesne języki programowania obiektowego

Typy ogólne

Aleksander Lamża
ZKSB · Instytut Informatyki
Uniwersytet Śląski w Katowicach

aleksander.lamza@us.edu.pl

Zawartość

- Czym są typy ogólne?
- Kilka kluczowych terminów
- Jak to działa?
- Klasy ogólne
- Metody ogólne
- Trochę bardziej szczegółowo o typach
- Typy wieloznaczne i ograniczanie typów

Czym są typy ogólne?

Typy ogólne (ang. *generics*) **–** inaczej nazywane typami uogólnionymi, parametryzowanymi, generycznymi **–** pozwalają na pisanie bezpieczniejszego i łatwiejszego w utrzymaniu kodu.

Cała sprawa polega na tym, że **parametryzujemy typy** i przerzucamy na kompilator odpowiedzialność za rzutowanie.

Dzięki temu błędy związane z zastosowaniem niewłaściwych typów zostają wykryte na etapie kompilacji, a nie uruchomienia.

<T>

Kilka kluczowych terminów

Klasa ogólna to taka klasa, która ma co najmniej jeden **parametr typu** (oczywiście to samo dotyczy też interfejsów).

Typ ogólny to klasa ogólna lub interfejs ogólny.

Każdy typ ogólny definiuje zbiór **typów parametryzowanych**. Składa się on z nazwy klasy (interfejsu) zakończonej listą **rzeczywistych parametrów typów** umieszczonych w nawiasach ostrych (trójkątnych).

Typ surowy to typ ogólny bez żadnego rzeczywistego parametru typu.

Kilka kluczowych terminów

Dla rozjaśnienia, rzućmy okiem na typowy przykład:

```
List<String>
```

Jest to typ parametryzowany reprezentujący listę, której elementy są typu `String`. Oznacza to, że rzeczywistym parametrem typu jest `String`.

Rzućmy okiem na definicję interfejsu `List`:

```
public interface List<E> extends Collection<E>
```

W omawianym przykładzie **formalnemu parametrowi** `E` odpowiada typ `String`.

Przykład

Żeby uzmysłowić sobie, dlaczego mechanizm typów uogólnionych jest przydatny, zastanówmy się nad problemem, z którym bardzo często możemy się spotkać.

Tworzymy kolekcję znaczków:

```
List stamps = new ArrayList();
```

i wstawiamy do niej znaczek:

```
stamps.add(new Stamp());
```

Świetnie. Teraz gdzieś indziej w kodzie wstawiamy do tej kolekcji monetę:

```
stamps.add(new Coin());
```

W czym problem?

Przykład

W czasie kompilacji nie pojawią się żadne problemy i program uda się uruchomić.

Gorzej, jeżeli postanowimy przeiterować po naszej kolekcji i np. wyświetlić informacje o wszystkich znaczkach:

```
for (Iterator it = stamps.iterator(); it.hasNext();) {
    Stamp s = (Stamp) it.next();
    s.printInfo();
}
```

Efekt będzie taki, że dostaniemy elegancki `ClassCastException`.
Jasne, że możemy tę pętlę otulić try-catchem, ale nie o to chodzi.

Rzecz w tym, żeby **potencjalne problemy z typami wykrywać jak najszybciej**, czyli najlepiej na etapie kompilacji.
Co więcej, w tej sytuacji błąd zostanie zgłoszony w miejscu próby odczytu elementu listy, a nie błędnego wstawienia do listy obiektu klasy `Coin` (zamiast `Stamp`), co utrudnia lokalizowanie przyczyn problemów.

Przykład

Jak ten przykład wyglądałby po zastosowaniu typów ogólnych?

Tworzymy kolekcję znaczków, podając rzeczywisty parametr typu elementów tej kolekcji jako `Stamp`:

```
List<Stamp> stamps = new ArrayList();
```

Przy próbie wstawienia do niej monety:

```
stamps.add(new Coin());
```

dostaniemy błąd kompilacji:

error: no suitable method found for add(Coin)

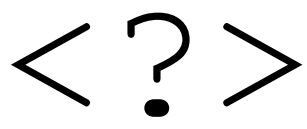
Świetnie! O to właśnie chodziło.
A jak będzie wyglądało iterowanie po takiej kolekcji?

Przykład

```
for (Iterator<Stamp> it = stamps.iterator(); it.hasNext();) {
    Stamp s = it.next();
    s.printInfo();
}
```

Rzutowanie nie jest już potrzebne, bo odpowiedzialność za to zrzuciliśmy na kompilator.

I tu pojawia się pytanie, jak ten mechanizm działa z perspektywy kompilatora.



Jak to działa?

Przede wszystkim: wirtualna maszyna Javy nie obsługuje typów ogólnych.

Każdemu typowi ogólnemu odpowiada typ surowy, czyli typ o tej samej nazwie, ale z **wymazanymi** (*ang.* erasure) parametrami typu. W ich miejsce są wstawiane typy graniczne (w przypadku ograniczeń – o czym później) lub typ `Object`. Dzięki temu w czasie wykonania instancje typów ogólnych współdzielą ten sam typ.

Zróbmy prosty test:

```
System.out.println(new ArrayList().getClass());
System.out.println(new ArrayList<String>().getClass());
System.out.println(new ArrayList<Integer>().getClass());
```

I co zobaczymy w konsoli?

Jak to działa?

```
class java.util.ArrayList
class java.util.ArrayList
class java.util.ArrayList
```

Stąd prosty wniosek – bez względu na parametr typu, wszystkie te obiekty są tak naprawdę instancjami surowego typu `ArrayList`.

No dobrze, ale w takim razie jak to działa?

Wcześniej była mowa o tym, że przerzucamy odpowiedzialność za rzutowanie na kompilator i tak właśnie się dzieje. Mamy taki kod:

```
List<String> names = new ArrayList();
…
String name = names.get(0);
```

Metoda `List.get()` – po wymazaniu typów – zwraca typ `Object`. Kompilator automatycznie wstawia w kodzie wynikowym dodatkową instrukcję maszyny wirtualnej rzutującą zwrócony typ `Object` na typ `String` (ten sam mechanizm działa dla pól ogólnych).

Klasy ogólne

Przejdźmy do konkretów. Doskonale wiemy, jak używać typów ogólnych jako użytkownicy klas, np. `ArrayList`. Pytanie, **jak tworzyć typy ogólne**?

Zacznijmy od **klas ogólnych**. Powiedzmy, że chcielibyśmy zdefiniować typ reprezentujący węzeł w jakieś strukturze danych (drzewie, grafie itp.), który może przechowywać dane dowolnego typu. Nazwijmy go `Node`:

```
public class Node<T> {

    T value;

    public Node(T value) { … }

    public T get() { … }

}
```

W definicji klasy posługujemy się symbolem `T` reprezentującym rzeczywisty typ przekazywany w momencie tworzenia instancji tej klasy, np.:

```
Node<String> node = new Node();
```

Metody ogólne

Czasami jest tak, że nie musimy parametryzować typu całej klasy, a jedynie metody (zwłaszcza w przypadku klas „narzędziowych”, które zawierają zbiór różnych wspomagających, najczęściej statycznych metod).

Przypuśćmy, że potrzebujemy metody zwracającej środkowy element tablicy dowolnego typu. Jej definicja może wyglądać tak:

```
public static <T> T getMiddle(T[] array) {
  return array[array.length / 2];
}
```

Przykładowe wywołanie metody wygląda następująco:

```
String[] a = {"raz", "dwa", "trzy"};
System.out.println(Generics.<String>getMiddle(a));
```

Parametr typu w większości sytuacji można pominąć, ponieważ kompilator ma wystarczająco dużo informacji o typie.

Uwaga na marginesie

Test na spostrzegawczość – jaką literą oznacza się parametry typów?

<?>

Uwaga na marginesie

Wszystko jest kwestią umowy i zwyczaju. Nic (poza zdrowym rozsądkiem) nie stoi na przeszkodzie, żeby napisać tak:

```
public class Node<TYPELEMENTUDRZEWABINARNEGO> { … }
```

W bibliotece Javy przyjęto następującą konwencję (której należy się trzymać):
– **`T`** oznacza dowolny typ,
– jeżeli chcemy sparametryzować kilka typów, poza `T` stosujemy **`U`** i **`S`**,
– **`E`** oznacza element kolekcji,
– **`K`** i **`V`** oznaczają klucz i wartość.

Trochę więcej szczegółów

Kluczowe aspekty typów ogólnych już omówiliśmy. Nie wyczerpuje to oczywiście tematu. Zastanówmy się nad następującą sytuacją:

Chcemy zaimplementować funkcję `min(a, b)` zwracającą mniejszą z dwóch wartości. Dla typu `int` mogłoby to wyglądać następująco:

```
public static int min(int a, int b) {
   return (a < b) ? a : b;
}
```

Chcemy zastosować typy ogólne, tak by funkcja działała nie tylko dla int-ów. Korzystamy więc z metody ogólnej:

```
public static <T> T min(T a, T b) {
   return (a < b) ? a : b;
}
```

Pojawia się problem:

error: bad operand types for binary operator '<'

Trochę więcej szczegółów

Problem polega na tym, że operator `<` można stosować jednie dla typów prostych. Jeżeli metoda ma działać na obiektach, musimy skorzystać z metody `compareTo()` zdefiniowanej w interfejsie `Comparable`:

```
public static <T> T min(T a, T b) {
   return (a.compareTo(b) < 0) ? a : b;
}
```

Metoda `compareTo()` zwraca wartość typu `int` (ujemną, zero albo dodatnią) określającą relację między porównywanymi wartościami.

(Uwaga: w rzeczywistej implementacji koniecznie należałoby się zabezpieczyć przed null-em w zmiennej `a`).

Spróbujmy to skompilować.

Trochę więcej szczegółów

Niestety przy próbie kompilacji dostajemy:

error: cannot find symbol

ponieważ nie można zagwarantować, że w obiekcie `a` ogólnego typu `T` będzie dostępna metoda `compareTo()`. Musimy więc zapewnić, że typ `T` implementuje interfejs `Comparable`. Wbrew pozorom nie robi się tego słowem `implements`, tylko `extends`:

```
public static <T extends Comparable> T min(T a, T b) { … }
```

Konstrukcja **`<T extends typ_graniczny>`** oznacza, że `T` musi być **podtypem** typu granicznego.

Gdybyśmy chcieli ograniczyć typy tylko do liczb, możemy zastosować zapis:

```
<T extends Number & Comparable>
```

No dobrze, wróćmy do przykładu. Kompilacja się powiodła, więc wszystko wygląda dobrze, ale...

Trochę więcej szczegółów

Podczas kompilacji pojawia się uwaga:

```
Note: Generics.java uses unchecked or unsafe operations.
```

O co tu chodzi? Rzućmy okiem na definicję interfejsu `Comparable`:

```
public interface Comparable<T>
```

I wszystko jasne – interfejs `Comparable` sam w sobie jest parametryzowanym typem. W zastosowanej przez nas konstrukcji brakuje wskazania rzeczywistego typu, więc wymuszamy użycie typu surowego, co – jak wiemy – nie jest wskazane.

Musimy przekazać typ. Tylko jaki? Narzuca się zapis:

```
public static <T extends Comparable<T>> T min(T a, T b) { … }
```

W wielu przypadkach to wystarczy, ale uniwersalnym rozwiązaniem sprawdzającym się przy bardziej skomplikowanych zależnościach między klasami byłoby zastosowanie **typów wieloznacznych**.

Trochę więcej szczegółów

Typy wieloznaczne jeszcze bardziej uelastyczniają system typów ogólnych. Mamy dwie konstrukcje. Pierwsza z nich ogranicza podtypy:

```
<? extends Foo>
```

Taki zapis reprezentuje dowolną **podklasę** klasy `Foo`.

Druga konstrukcja ogranicza nadtypy:

```
<? super Foo>
```

Jak się można spodziewać, zapis reprezentuje dowolną **nadklasę** klasy `Foo`.

Trochę więcej szczegółów

W rozpatrywanym przez nas przykładzie, zastosujemy drugą z konstrukcji:

```
public static <T extends Comparable<? super T>> T min(T a, T b)
```

Interfejs `Comparable` parametryzujemy typem `T` i wszystkimi jego nadtypami.

Aby w pełni skorzystać z zalet typów ogólnych, trzeba starać się eliminować wszystkie ostrzeżenia i uwagi związane z niebezpieczeństwami podstawienia niewłaściwych typów. Tylko wtedy mechanizmy Javy ustrzegą nas przed wyjątkami `ClassCastException`.

Istnieje jednak możliwość wyłączenia ostrzeżeń związanych z typami ogólnymi. Służy do tego adnotacja `@SupressWarnings` z parametrem `"unchecked"`. Można to zrobić tylko wtedy, gdy jesteśmy w 100% pewni, że kod jest bezpieczny dla typów. Powinniśmy też umieścić w kodzie komentarz tłumaczący, dlaczego konstrukcja jest bezpieczna.

Do poczytania

Java Generics FAQs
http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

Joshua Bloch: **Java. Efektywne programowanie**. Helion, Gliwice 2009.