Typy sparametryzowane

Streszczenie

Celem wykładu jest zaprezentowanie typów sparametryzowanych.
Czas wykładu – 90 minut.

Istnieją algorytmy, których zasada działania nie zależy od typu danych wejściowych. Możemy na przykład sortować obiekty różnych typów stosując te same mechanizmy. Ważne tylko, aby zadane było pewne kryterium porównujące obiekty między sobą.

Zanim zajmiemy się algorytmami, zobaczmy w jaki sposób przygotowuje się sparametryzowane struktury danych.

Parametr (bądź parametry) przekazuje się poprzez dowolny identyfikator (bądź identyfikatory) umieszczone w nawiasach <> po nazwie klasy:

```
class A<T>
{
    T t;
    void wypisz() { System.out.println(t);  }
}
```

Powyższy przykład definiuje klasę `A` z parametrem `T`. Wykorzystanie jej wymaga podania konkretnego typu, który będzie podstawiony zamiast parametru. W poniższym przykładzie generowane są dwa obiekty: jeden (wskazywany przez referencję `a`) typu `A<Integer>`, drugi (wskazywany przez `b`) typu `A<String>`. Typem `a` i `b` nie jest klasa `A`, ale klasa `A` z odpowiednim argumentem.

```
class typy
{
    public static void main(String[] args)
    {
        A<Integer> a = new A<Integer>();
        a.t = new Integer(-3);
        a.wypisz();
        A<String> b = new A<String>();
        b.t = "Ala ma kota";
        b.wypisz();
    }
}
```

Typem podstawianym za parametry klasy muszą być klasy. Typy skalarne są niedopuszczalne. Poniższy przykład jest zatem niepoprawny.

```
class typy
{
    public static void main(String[] args)
    {
        A<int> c; // błąd
    }
}
```

Wykorzystanie parametrów w klasach

Parametr klasy może być wykorzystany jako typ pola, typ argumentów funkcji oraz typ wyniku funkcji:

```
class A<T>
{
    T t; // typ pola
    A(T a) { t = a; } // typ argumentu funkcji
    T g() { return t; } // typ wyniku funkcji
    void wypisz() { System.out.println(g()); }
}
```

Istnieją jednak pewne ograniczenia na stosowanie parametru klasy.

- Pól, których typem jest parametr, nie można inicjalizować w momencie deklaracji.
- Nie można generować tablic o typie bazowym, który jest parametrem (związane z powyższym).
- Funkcja o typie wyniku będącym parametrem nie może być statyczna.
- Klasy wyjątków nie mogą być sparametryzowane.

Proszę odkomentować kolejne linie przykładu i zaobserwować komentarze kompilatora.

```
class A<T>
{
    // T t = new T();    // inicjalizacja pola
    // T[] x = new T[4]; // tworzenie tablicy
    // static T f() {  } // funkcja statyczna
}
```

Nie są to jednak ograniczenia, które uniemożliwiają wygodne stosowanie klas sparamatryzowanych. Inicjalizacje należy przenieść, np. do konstruktorów:

```
class A<T>
{
    T[] x;
    A(T[] y) { x = y; }
}
```

Kolejne ograniczenie związane z klasami z parametrem jest obserwowalne przy klasach z wieloma parametrami. Przeanalizujmy przykład klasy z dwoma parametrami:

```
class A<T1,T2>
{
    void f(T1 t) { }
    void f(T2 t) { }
}
```

Na pierwszy rzut oka wydaje się, że przykład jest poprawny. Przyjrzyjmy się jednak funkcjom `f(T1 t)` oraz `f(T2 t)`. W momencie definiowania klasy nie mamy żadnej gwarancji, że w momencie użycia klasy parametry `T1` oraz `T2` będą konkretyzowane różnymi typami. Co się stanie, gdy za `T1` i `T2` podstawiony zostanie ten sam typ, np. `Integer`? W konsekwencji powstaną dwie funkcje `f(Integer t)`. Z własności mechanizmu przeciążania funkcji wiemy, że taka sytuacja jest niedopuszczalna.

Klasy z ograniczonym parametrem

Rozważmy program obliczający średnią arytmetyczną wyrazów tablicy. Jaki jest algorytm? Należy zsumować wszystkie elementy tablicy i tę sumę podzielić przez ilość elementów. Proszę zauważyć, że nic nie mówimy o typie elementów tablicy. Może nim być typ całkowity, rzeczywisty, lub inny typ liczbowy w językach które dopuszczają inne typy liczbowe. Ale nawet w Javie nic ma potrzeby o mówieniu, który typ całkowity czy rzeczywisty wybieramy. Istotne jest tylko, aby elementy danego typu miały wartość liczbową, którą można będzie sumować. Poniższy przykład ilustruje implementację programu. Zwróćmy uwagę na instrukcję `suma += a.doubleValue();`. Pobierana jest wartość liczbowa kolejnego elementu tablicy `tab` o typie bazowym `T` będącym parametrem klasy. Ale żeby to było możliwe elementy typu `T` muszą taką wartość posiadać. Muszą być zatem potomkami klasy `Number`, klasy posiadającej funkcję `doubleValue();`

```
class A<T extends Number>
{
    T[] tab;
    A(T[] x) { tab = x; }
    double srednia()
    {
        double suma = 0.0;
        for (T a: tab) suma += a.doubleValue();
        return suma / tab.length;
    }
    void wypisz() { System.out.println(srednia()); }
}
```

Wykorzystajmy tę klasę do obliczenia średniej arytmetycznej tablicy liczb całkowitych oraz tablicy liczb rzeczywistych:

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

```
        Integer[] i = { 4,6,8,10 };
        A<Integer> ai = new A<Integer>(i);
        ai.wypisz();

        Double[] d = { 2.0, 4.0, 6.0 };
        A<Double> ad = new A<Double>(d);
        ad.wypisz();

        // String[] s1 = { "aa" };
        // A<String> as1 = new A<String>(s1);
        // as1.wypisz();
    }
}
```

Proszę z powyższym kodem zrobić dwa eksperymenty: a) wykomentować część dotyczącą napisów oraz b) skasować tekst `<T extends Number>` i zaobserwować komentarze kompilatora.

Wieloznaczne argumenty funkcji

Rozszerzmy program o funkcję porównującą wartości średnie dwóch różnych tablic. Wiemy, że mając jedną funkcję `srednia()` możemy obliczać średnie tablic o różnych typach bazowych. Chcielibyśmy zatem, aby nowo dodana funkcja pozwalała na porównywanie średnich tablic o różnych typach bazowych. Java umożliwia to przez zastosowanie tzw. *argumentów wieloznacznych*. Uzyskuje się je stosując `?` (znak zapytania) zamiast identyfikatora typu parametru.

```
class A<T extends Number>
{
    T[] tab;
    A(T[] x) { tab = x; }
    double srednia()
    {
        double suma = 0.0;
        for (T a: tab) suma += a.doubleValue();
        return suma / tab.length;
    }
    boolean porownaj_srednie(A<?> o)
    {
        return srednia() == o.srednia();
    }
}
```

Zwróćmy uwagę, że gdyby funkcja porównująca średnie korzystała z typu `T` będącego parametrem całej klasy, tj. miała sygnaturę `boolean porownaj_srednie(A<T> o)`, umożliwiałaby porównywanie średnich tablic o jednakowych typach bazowych.

Oto przykład wykorzystujący omawianą klasę do obliczania i porównywania średnich tablic liczb całkowitych oraz rzeczywistych:

```
class typy
{
    public static void main(String[] args)
    {
        Integer[] i = { 2,4,6,1 };
        A<Integer> ai = new A<Integer>(i);

        Integer[] j = { 2,4,6 };
        A<Integer> aj = new A<Integer>(j);

        Double[] d = { 2.0, 4.0, 6.0, 1.0 };
        A<Double> ad = new A<Double>(d);

        System.out.println(ai.porownaj_srednie(aj));

        System.out.println(ai.porownaj_srednie(ad));
    }
}
```

Argumenty wieloznaczne z ograniczeniem

Kolejnym rozszerzeniem języka są tzw. *argumenty wieloznaczne z ograniczeniem*. Podobnie jak w przypadku parametrów klas możemy ograniczyć typy argumentów funkcji, które mogą być podstawiane za argumenty wieloznaczne. Po znaku zapytania oznaczającego argument wieloznaczny podaje się słowo `extends` i identyfikator typu. Za taki argument podstawiony może być obiekt dowolnego typu, ale koniecznie potomka typu zadeklarowanego jako ograniczenie bądź typu ograniczenia. Ponadto, stosując słowo `super` i identyfikator typu ogranicza się typy argumentów, które mogą być podstawione w danym miejscu do typu ograniczenia lub jego przodka.

Przeanalizujmy przykład. Zdefiniujmy ciąg klas od siebie zależnych (ich zawartość nie jest istotna dla przykładu):

```
class K1 {}
class K2 extends K1 { }
class K3 extends K2 { }
class K4 extends K3 { }
class K5 extends K4 { }
class K6 extends K5 { }
```

oraz klasę

```
class A<T extends K2>
{
    void f1 (A<T> o) { System.out.println("f1"); }
    void f2 (A<?> o) { System.out.println("f2"); }
    void f3 (A<? extends K4> o) { System.out.println("f3"); }
    void f4 (A<? super K4> o) { System.out.println("f4"); }
}
```

Pytanie: które z klas K1 do K6 mogą być podstawiane za typ T będący parametrem klasy A oraz które z tych klas mogą być typami argumentów funkcji od f1 do f4?

```
class typy
{
    public static void main(String[] args)
    {
        // A<K1> k1 = new A<K1>();
        A<K2> k2 = new A<K2>();
        A<K3> k3 = new A<K3>();
        A<K4> k4 = new A<K4>();
        A<K5> k5 = new A<K5>();
        A<K6> k6 = new A<K6>();

        // przetestować poniższe kombinacje
        // k6.f4(k6);
        // k2.f1(k3);
        // k2.f2(k3);
        // k4.f3(k6);
        // k2.f4(k4);
    }
}
```

Dziedziczenie klas z parametrami

Klasy sparametryzowane mogą być elementami drzewa typów. Mogą dziedziczyć po innych klasach. Mogą być dziedziczone. Mogą mieć większą ilość parametrów niż klasa przodka. Mogą mieć mniejszą ilość parametrów niż klasa przodka.

```
class K1 {}
class K2<T> extends K1 {}
class K3<T,V extends T> extends K2<Integer> {}
class K4 extends K3<Object,String> {}
```

Klasa K1 jest potomkiem klasy Object. Nie posiada żadnych parametrów.

Klasa K2 dziedziczy po bezparametrowej klasie K1, ale sama posiada parametr.

Klasa K3 jest potomkiem sparametryzowanej klasy K2, gdzie ten parametr ukonkretnia typem Integer. Sama zaś posiada dwa parametry, gdzie drugi musi rozszerzać pierwszy.

Klasa K4 nie ma parametrów, ale dziedziczy po klasie K3 ukonkretniając jej parametry typami Object oraz String.

Przykład wykorzystania klas od K1 do K4:

```
class typy
{
    public static void main(String[] args)
    {
        K1 k1 = new K1();
        K2<Double> k2 = new K2<Double>();
        K3<Object,String> k3 = new K3<Object,String>();
```

```
        K4 k4 = new K4();
    }
}
```

Funkcje ze sparametryzowanymi typami argumantów

Jeśli typ z parametrem potrzebny jest nie w obrębie całej klasy, ale tylko w jednej funkcji (lub kilku funkcjach, ale niezwiązanych ze sobą), można sparametryzować tylko wybrane typy argumentów wybranych funkcji. Parametry typów wprowadza się w nawiasach <> przed typem wyniku. Klasa

```
class A
{
    static <T, V extends T> void f(T t, V v)
    {
        System.out.println("" + t + "  " + v);
    }
}
```

definiuje jedną dwuargumentową funkcję `f`, której typ drugiego argumentu ma rozszerzać się do typu argumentu pierwszego. (Funkcję `f` zdefiniowano jako statyczną, ale tylko na potrzeby poniższego jej użycia. W ogólnym przypadku funkcje z argumentami sparametryzowanymi nie muszą być statyczne.)

```
class typy
{
    public static void main(String[] args)
    {
        A.f(new Object(), new Integer(3));
        // A.f(new String(), new Integer(1));
    }
}
```

Dlaczego drugie wywołanie funkcji `f` jest wykomentowane?

Interfejsy z parametrami

Sparametryzowane mogą być również interfejsy.

```
interface I<T,V extends Number>
{
    void f(T t,V v);
}
```

Klasy implementujące takie interfejsy muszą w swojej deklaracji zawrzeć typy konkretyzujące parametry implementowanego interfejsu. Może to być parametr klasy (`T` w przykładzie), lub konkretny typ (`Integer` w przykładzie):

```
class A<T> implements I<T,Integer>
{
    public void f(T t,Integer v)
    {
        System.out.println("Jestem w f:  " + t + "  " + v);
    }
}

class typy
{
    public static void main(String[] args)
    {
        A<Double> a = new A<Double>();
        a . f ( new Double(3) , new Integer(8) );
        a . f (3.0 , 8);
    }
}
```

Podsumowanie

- Klasy z parametrami umożliwiają programowanie algorytmów niezależnych od typów.
- Nazwy parametrów podaje się w <> po nazwie klasy.
- Typy parametrów klas mogą być ograniczane.
- Klasy z parametrami mogą dziedziczyć inne klasy.
- Klasy z parametrami mogą być dziedziczone przez inne klasy.
- Parametry mogą posiadać tylko wybrane funkcje klas (niekoniecznie całe klasy).
- Interfejsy mogą mieć parametry.
- Funkcje mogą mieć parametry wieloznaczne i wieloznaczne z ograniczeniami.