Java Advanced - dodatkowe

v1.0.0

Typy generyczne

Typy generyczne

Typy generyczne - inaczej typy uogólnione (ang. generics) zostały wprowadzone w Javie 1.5 by umożliwić parametryzowanie metod, klas i interfejsów.

W praktyce oznacza to, że możemy podać typ (lub typy) argumentu przyjmowanego przez metody danej klasy czy interfejsu dopiero w momencie jej użycia w kodzie, dzięki czemu można uniknąć kłopotliwego rzutowania typów.

Typy generyczne

Zaletą **typów generycznych** jest to, że kompilator jest w stanie sprawdzić poprawność typów na etapie kompilacji.

Dzięki użyciu typów generycznych możemy wielokrotnie użyć pewnych części kodu niepowiązanych ściśle z konkretną implementacją.

Metodyka programowania oparta o korzystanie z typów i metod generycznych nazywana jest programowaniem generycznym lub uogólnionym.

Na typach generycznych opiera się między innymi implementacja poznanych już kolekcji.

Przykład

Aby przedstawić zalety generyków, rozważmy następujący przykład – mamy ma za zadanie napisać klasę **Box**, która będzie przechowywała elementy dowolnego typu.

Bez znajomości generyków moglibyśmy stworzyć:

```
public class Box {

    private Object element;

    public Object getElement() {
        return element;
    }

    public void setElement(Object element) {
        this.element = element;
    }

}
```

Przykład

Na pierwszy rzut oka takie rozwiązanie wydaje się jak najbardziej prawidłowe.

Mamy w klasie pole typy **Object**, które może przechowywać dowolny typ obiektu dziedziczący po **Object** (czyli wszystko poza typami prostymi).

Mamy możliwość utworzyć obiekt klasy **Box**, który będzie przechowywać obiekty typu **String**, **Integer** czy **Float** (lub dowolny inny).

Bez znajomości generyków moglibyśmy stworzyć:

```
public class Box {

    private Object element;

    public Object getElement() {
        return element;
    }

    public void setElement(Object element) {
        this.element = element;
    }

}
```

Przykład

W takim podejściu kryje się jednak zagrożenie.

Skoro pole element może przyjąć dowolny typ, to mogą pojawić się błędy spowodowane nieodpowiednim typem w konkretnej sytuacji jego użycia – np. próba wykonania na tym polu metody, która nie należy do jego typu.

Rozwiązaniem powyższego problemu jest stworzenie klasy, która umożliwi podanie typu polu element w momencie jego wykorzystania w kodzie.

Dzięki temu kompilator będzie w stanie zweryfikować czy pole to jest poprawnie używane.

Przykład

Klasa będzie mogła zwracać obiekt konkretnego typu, a nie Object przez co unikniemy rzutowania.

Tak właśnie działają generyki.

Klasa Box w wersji generycznej wyglądała by następująco :

```
public class Box <T> {

    private T element;

    public T getElement() {
        return element;
    }

    public void setElement(T element) {
        this.element = element;
    }

}
```

Przykład

Jak widać pojawiły się charakterystyczne nawiasy trójkątne, w których znajduje się litera T (standardowo jako akronim od słowa type).

Kompilator nie wymaga żeby była to litera T, jest to tylko powszechnie stosowana konwencja nazewnicza.

- **E** – Element (ang. Element),
- **T** – Typ (Type),
- **N** – Liczba (Number),
- **V** – Wartość (Value),
- **K** – Klucz (Key),
- **S,U,V** – kolejne typy.

Przykład

W naszym przykładzie T może reprezentować wszystko co nie jest typem prostym (prymitywem).

Klasa może przyjmować wiele parametrów, np.:

```
public class Box <T, S, V> {

    private T element1;
    private S element2;
    private V element3;

    ...

}
```

Użycie

Aby użyć klasy generycznej w kodzie należy skonkretyzować poszczególne parametry, tj. podać ich typ.

Nie jest to ściśle wymagane – to znaczy, że nie podanie typu parametru nie spowoduje błędu kompilacji, a wyświetlona zostanie jedynie ostrzeżenie.

Poniższa linia kodu tworzy instancję klasy generycznej Box, która jest parametryzowana typem String.

```
Box<String> box1 = new Box<String>();
```

Użycie

Możemy także tworzyć instancje:

```
Box<Integer> box2 = new Box<Integer>();
Box<Double> box3 = new Box<Double>();
```

Znanym nam przykładem generyków są kolekcje, np. ArrayList, czy Map:

```
List<String> list = new ArrayList<String>();
Map<Integer, String> map = new HashMap<>();
```

Ograniczenia typów

Czasami zachodzi potrzeba, by ograniczyć zakres typów, którymi będziemy mogli parametryzować klasy generyczne, w tym celu możemy użyć słowa kluczowego **extends**.

Użycie słowa extends ogranicza możliwość użycia klas do tych, które dziedziczą po określonej klasie lub implementują dany interfejs.

```
public class Box <T extends Number> {

    private T element;

    ...

}
```

W powyższym przykładzie klasa będzie mogła być sparametryzowana wyłącznie przez typy dziedziczące po **java.lang.Number**, czyli typy numeryczne, np. **Float**, **Double**, **Integer**.

Ograniczenia typów

Użycie słowa extends ogranicza możliwość użycia klas do tych, które dziedziczą po określonej klasie lub implementują dany interfejs.

Dzięki zastosowaniu ograniczeń w każdym miejscu klasy możemy korzystać z metod klasy bazowej (w naszym przykładzie - Number), np.:

```
public class Box <T extends Number> {

    private T element;
    //...
    public long getAsLong(){
        return element.longValue();
    }
}
```

W powyższym przykładzie klasa będzie mogła być sparametryzowana wyłącznie przez typy dziedziczące po **java.lang.Number**, czyli typy numeryczne, np. **Float**, **Double**, **Integer**.

Ograniczenia typów

Istnieje także możliwość zastosowania wielokrotnego ograniczenia, np. do typów implementujących kilka interfejsów, lub też klasę oraz interfejs.

```
public class Box <T extends Number & Comparable> {

    private T element;
    //...

}
```

Refleksja

Refleksja

Refleksja - polega na dynamicznym korzystaniu ze struktur programowania, które w momencie pisaniu kodu nie musiały być jeszcze znane.

Pozwala to na dynamiczne ładowanie klas i dostęp do ich publicznych pól i metod.

Paradygmat programowania obiektowego opartego na refleksji nazywamy programowaniem refleksyjnym.

Refleksja

Mechanizm refleksji pozwala na modyfikację funkcjonowania programu w trakcie jego wykonywania (bez konieczności dokonywania zmian w kodzie źródłowym).

Dzięki refleksji możemy zarządzać kodem w taki sposób, jakbyśmy zarządzali zwykłymi danymi.

Dzięki refleksji można również odczytać informacje o strukturze klas podczas działania programu, ich polach i dostępnych metodach, typach argumentów metod, czy informacji o klasach nadrzędnych danego obiektu.

Refleksja

Refleksja - polega na dynamicznym korzystaniu ze struktur programowania, które w momencie pisaniu kodu nie musiały być jeszcze znane.

Pozwala to na dynamiczne ładowanie klas i dostęp do ich publicznych pól i metod.

Paradygmat programowania obiektowego opartego na refleksji nazywamy programowaniem refleksyjnym.

Refleksja

Mechanizm refleksji pozwala na modyfikację funkcjonowania programu w trakcie jego wykonywania (bez konieczności dokonywania zmian w kodzie źródłowym).

Dzięki refleksji możemy zarządzać kodem w taki sposób, jakbyśmy zarządzali zwykłymi danymi.

Można również odczytać informacje o strukturze klas podczas działania programu, ich polach i dostępnych metodach, typach argumentów metod, czy informacji o klasach nadrzędnych danego obiektu.

Refleksja

Klasy obsługujące mechanizm refleksji w Javie zostały zgrupowane w pakietach **java.lang** i **java.lang.reflect**, m. in.

- **java.lang.reflect.Array**– dostarcza metod statycznych do dynamicznego tworzenia i dostępu do tablic javy
- **java.lang.reflect.Field**– reprezentuje atrybuty klasy
- **java.lang.reflect.Constructor**– reprezentuje konstruktory klasy
- **java.lang.reflect.Method**– reprezentuje metody klasy

Jak to działa?

Jak już wiemy z wcześniejszych wykładów, klasą nadrzędną dla każdej klasie w Javie jest **java.lang.Object**.

Każdy typ natomiast (obiektowy, prosty, tablicowy, itp.) jest reprezentowany przez instancję klasy **java.lang.Class** , którą można uzyskać poprzez wywołanie metody **`getClass()`** zaimplementowanej w klasie **Object**.

```
String sText = "text";
Class sClass = sText.getClass();
```

Jak to działa?

Jeśli nie mamy obiektu, możemy użyć atrybutu **class**.

```
Class c = String.class;
```

Tym sposobem możliwe jest uzyskanie instancji Class z typu prymitywnego (prostego)

```
Class c = boolean.class;
```

Instancję Class można również uzyskać znając jedynie nazwę klasy.

W tym celu można skorzystać ze statycznej metody **forName(className)** klasy **Class**.

```
Class c = Class.forName("pl.coderslab.MyClass");
```

Jak to działa?

Dzięki metodzie **`forName()`** możliwe jest uzyskanie obiektu reprezentującego klasę, która mogła nie być jeszcze zaimplementowana w trakcie pisanie programu.

Korzystając z tej metody musimy obsłużyć wyjątek **ClassNotFoundException**, który jest zwracany gdy klasa o podanej nazwie nie zostanie znaleziona.

Metody

Mając referencję do obiektu klasy Class, możemy skorzystać z szeregu przydatnych metod, m. in.:

- **getName()** - zwraca nazwę klasy
- **getPackage()** - zwraca obiekt Class reprezentujący klasę nadrzędną danej klasy
- **getInterfaces()** - zwraca tablicę obiektów java.lang.Class zawierającą interfejsy implementowane przez obiekt
- **getFields()** / **getDeclaredFields()** - zwraca tablicę obiektów java.lang.reflect.Field reprezentującą publicznie dostępne atrybuty klasy

Metody

- **getFields() / getDeclaredFields()** - zwraca tablicę obiektów java.lang.reflect.Field reprezentującą publicznie dostępne atrybuty klasy
- **getMethods() / getDeclaredMethods()** - zwraca tablicę obiektów java.lang.reflect.Method reprezentującą publicznie dostępne metody klasy
- **getConstructors() / getDeclaredConstructors()** - zwraca tablicę obiektów java.lang.reflect.Constructor reprezentującą publicznie dostępne kontrolery klasy

Metody zawierające w nazwie słowo **Declared** zwracają wszystkie składowe – wraz z prywatnymi (private) i chronionymi (protected)

Metody

Jedną z najistotniejszych metod klasy **Class** jest **`newInstance()`**.

Jest to jedyna metod dzięki której możemy tworzyć nowe obiekty w ramach mechanizmu refleksji (nie można w tym celu używać operatora new).

Przykład

W celu przedstawienia przykładu posłużymy się klasą:

```
public class MyClass {

    public void hello() {
        System.out.println("Hello world!!!");
    }
}
```

Bez refleksji:

```
MyClass myClass = new MyClass();
myClass.hello();
```

Z użyciem refleksji:

```
Class myClass = Class.forName("MyClass");
Method myMethod = myClass.getMethod("hello");
myMethod.invoke(myClass.newInstance());
```

Przykład

W obu fragmentach tworzony jest obiekt klasy **MyClass** i wykonywana jest metoda **`hello`** (w obu wypadkach na standardowym wyjściu zostanie wyświetlony napis „Hello world”).

Różnica polega na tym, że korzystając z mechanizmu refleksji nazwy klas i zmiennych możemy przenieść do zmiennych i ustalać ich wartość podczas wykonywania programu.

Przykład

Dzięki refleksji możemy także odczytać informacje na temat struktury klasy podczas pracy programu.

```
Field[] fields = myClass.getDeclaredFields();
System.out.println("Class " + myClass.getName() + " has " + fields.length + " fields");
for (Field field : fields) {
    System.out.println(field.getName());
}
```

```
Method[] methods = myClass.getDeclaredMethods();
System.out.println("Class " + myClass.getName() + " has " + methods.length + " methods");
for(Method method : methods){
    System.out.println(method.getName() + " returns " + method.getReturnType().getName());
}
```