

Paradygmaty programowania - ćwiczenia Lista 8

Na wykładzie zostały przypomniane podstawowe informacje, dotyczące programowania obiektowego w języku Java. Poniżej jako ilustracja omówionych mechanizmów zostanie zdefiniowana klasa uogólniona (ang. generic class) BT dla binarnych drzew niemutowalnych. Pozwala ona na wykorzystywanie drzew binarnych w stylu programowania funkcyjnego (oczywiście bez mechanizmu dopasowania do wzorca, którego w Javie nie ma). Definicja klasy jest oparta na przykładzie z poniższej książki:

P.-Y.Saumont, *Java. Programowanie funkcyjne*. Helion 2017

<https://helion.pl/ksiazki/java-programowanie-funkcyjne-pierre-yves-saumont.javapf.htm>

```
package immutablestructures;
```

```
public abstract class BT<A> {  
    public abstract A elem();  
    public abstract BT<A> left();  
    public abstract BT<A> right();  
  
    public static final BT EMPTY = new Empty();  
  
    private static class Empty<A> extends BT<A> {  
        @Override public A elem() {  
            throw new IllegalStateException("elem() called on empty tree");  
        }  
  
        @Override public BT<A> left() {  
            throw new IllegalStateException("left() called on empty tree");  
        }  
  
        @Override public BT<A> right() {  
            throw new IllegalStateException("right() called on empty tree");  
        }  
    }  
  
    private static class Node<A> extends BT<A> {  
        private final A elem;  
        private final BT<A> left;  
        private final BT<A> right;  
  
        private Node(A elem, BT<A> left, BT<A> right) {  
            this.elem = elem;  
            this.left = left;  
            this.right = right;  
        }  
    }  
}
```

```

@Override
public A elem() {
    return elem;
}

@Override
public BT<A> left() {
    return left;
}

@Override
public BT<A> right() {
    return right;
}
}

@SuppressWarnings("unchecked")
public static <A> BT<A> empty() {
    return EMPTY;
}

public static <A> BT<A> node(A elem, BT<A> left, BT<A> right) {
    return new Node<>(elem, left, right);
}
}

```

Wykorzystywane są statyczne podklasy zagnieżdżone. Zagnieżdżenie ogranicza widoczność ich składowych i pozwala uniknąć zaśmiecania pakietu ogólnymi nazwami, takimi jak `Empty` czy `Node`. Każdy obiekt wewnętrznej klasy posiada referencję do obiektu klasy zewnętrznej, natomiast statyczna klasa zagnieżdżona nie ma takiej referencji (tak jak statyczna metoda nie ma referencji `this`). Podklasy są też prywatne, więc użytkownicy nie mają dostępu do reprezentacji drzew ani do konstruktorów. Drzewa można tworzyć wyłącznie za pomocą statycznych metod fabrycznych (ang. *factory methods*) `empty` i `node`.

Zwróć uwagę na sposób zdefiniowania pustego drzewa:

```
public static final BT EMPTY = new Empty();
```

Typ `BT` nie jest sparametryzowany. Jest to tzw. typ surowy (ang. *raw type*), dzięki czemu singleton `EMPTY` może reprezentować puste drzewo z elementami dowolnego typu. Na wykładzie 4 (str. 22) w *Scali* został w tym celu wykorzystany typ `BT[Nothing]`, ale w Javie nie ma typu `Nothing`. Typów surowych należy unikać (będzie jeszcze o tym mowa), ale tutaj jego użycie (lokalnie!) jest uzasadnione, w przeciwnym razie trzeba by tworzyć inne puste drzewo dla każdego typu. Użycie typów surowych powoduje jednak utratę bezpieczeństwa, co kompilator sygnalizuje za pomocą ostrzeżenia. Pominięcie adnotacji `@SuppressWarnings("unchecked")` przed definicją metody fabrycznej `empty()` spowoduje wyprowadzenie ostrzeżenia:

```
BT.java:53: warning: [unchecked] unchecked conversion
```

```
    return EMPTY;
```

```
    ^
```

```
required: BT<A>
```

```
found:    BT
```

```
where A is a type-variable:
```

```
  A extends Object declared in method <A>empty()
```

Na ostatniej stronie wykładu 8 pokazany jest rekomendowany sposób implementacji singletonów za pomocą typu wyliczeniowego (enumeracji). Podejścia tego nie można jednak zastosować, jeśli singleton musi rozszerzać klasę nadrzędną inną niż `Enum`.

Poniżej znajduje się program, wykorzystujący klasę BT. Została tam zdefiniowana funkcja printBT, która w czytelny sposób wyświetla zadane drzewo binarne (porównaj ją z przykładową funkcją w OCamlu z listy 6). Została ona wywołana dla prostego drzewa binarnego.

Statyczne metody fabryczne można zaimportować statycznie:

```
import static immutablestructures.BT.*;
```

Dzięki temu można uniknąć notacji kropkowej i tworzyć drzewa w analogiczny sposób, jak to robiliśmy na wykładzie 4 w Scali (str. 23) i w OCamlu (str. 20).

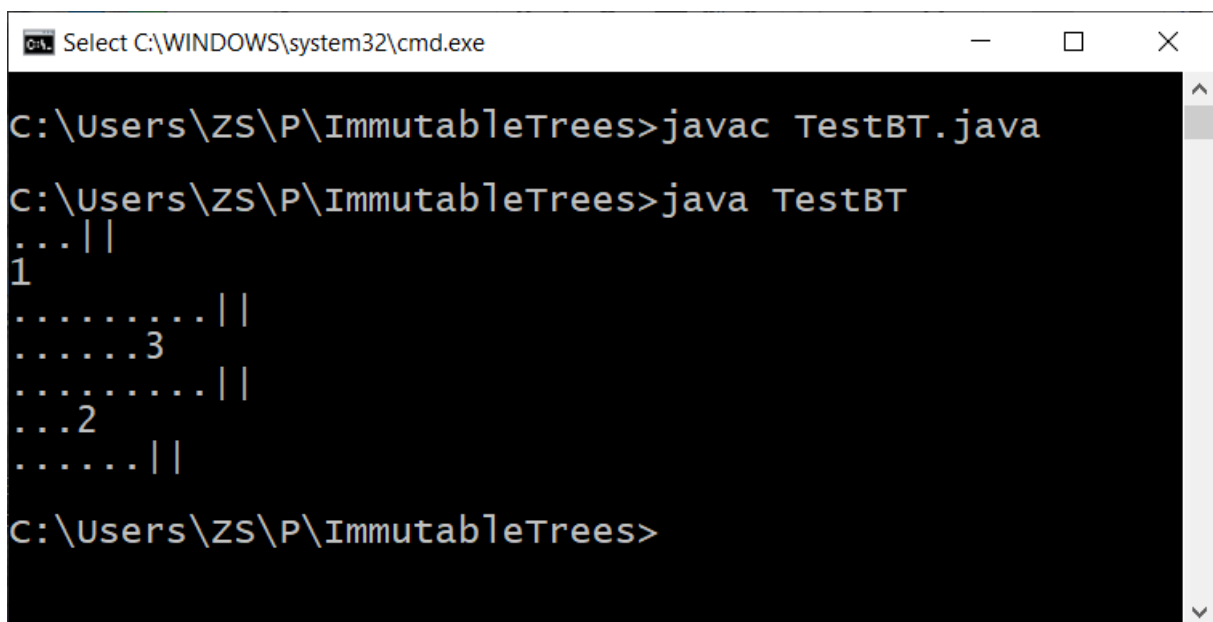
```
import immutablestructures.BT;
```

```
import static immutablestructures.BT.*;
```

```
public class TestBT {
    public static void main(String[] args){
        BT<Integer> tree = node(1, node(2, empty(), node(3, empty(), empty())), empty());
        printBT(tree);
    }

    private static void aux(BT<Integer> t, int height) {
        if (t == EMPTY) {
            for(int i=0; i<height; i++) System.out.print("...");
            System.out.println("||");
        }
        else {
            aux(t.right(), height+1);
            for(int i=0; i<height; i++) System.out.print("...");
            System.out.println(t.elem());
            aux(t.left(), height+1);
        }
    }

    public static void printBT(BT<Integer> tree) {
        aux(tree, 0);
    }
}
```



```
Select C:\WINDOWS\system32\cmd.exe

C:\Users\ZS\P\ImmutableTrees>javac TestBT.java

C:\Users\ZS\P\ImmutableTrees>java TestBT
... ||
1
..... ||
..... 3
..... ||
... 2
..... ||

C:\Users\ZS\P\ImmutableTrees>
```

Paradygmaty programowania - ćwiczenia

Lista 8

1. (Java) Dany jest następujący interfejs dla kolejek .

```
public interface MyQueue<E> {  
    public void enqueue( E x ) throws FullException;  
    public void dequeue( );  
    public E first( ) throws EmptyException;  
    public boolean isEmpty( );  
    public boolean isFull( );  
}
```

Napisz dwie klasy publiczne dla wyjątków FullException i EmptyException.

Napisz klasę generyczną, implementującą interfejs MyQueue, w której kolejka jest reprezentowana przez **tablicę** cykliczną (patrz zadanie 2 z listy 7). **Użyj kolekcji ArrayList**. Przeprowadź test na małej kolejce (np. o rozmiarze 3), którą całkowicie zapelnisz.

2 . Przeanalizuj następujący program w Javie. Czy ten program się skompiluje? Jeśli nie, to dlaczego i jak go poprawić (bez zmieniania argumentów metod)?

```
public class Test {  
    int zawartość = 0;  
    static void argNiemodyfikowalny(final Test zmienna) {  
        zmienna.zawartość = 1;  
        zmienna = null;  
    }  
    static void argModyfikowalny(Test zmienna) {  
        zmienna.zawartość = 1;  
        zmienna = null;  
    }  
    public static void main(String[] args) {  
        Test modyfikowalna = new Test();  
        final Test niemodyfikowalna = new Test();  
        // tutaj wstaw instrukcje  
    }  
}
```

Co i dlaczego zostanie wyświetlone, jeśli wiersz „// tutaj wstaw instrukcje” zastąpimy następującymi instrukcjami:

- a) argNiemodyfikowalny(modyfikowalna);
System.out.println(modyfikowalna.zawartość);
- b) argNiemodyfikowalny(niemodyfikowalna);
System.out.println(niemodyfikowalna.zawartość);
- c) argModyfikowalny(modyfikowalna);
System.out.println(modyfikowalna.zawartość);
- d) argModyfikowalny(niemodyfikowalna);
System.out.println(niemodyfikowalna.zawartość);

Działanie programu należy wyjaśniać, rysując jego „obraz pamięci”, tzn. rysując referencje w postaci strzałek, komórki pamięci i ich zawartości jako prostokąty. Należy pokazać, co będzie umieszczone na stosie, a co na sterwie programu (patrz wykład 2, str. 15-16).