Programowanie Obiektowe

Laboratorium 4 – modyfikacja przykładowej aplikacji

Tomasz Bieliński

2017-03-21

Przygotowanie do zajęć w domu

Celem tego laboratorium będzie rozwinięcie prostej aplikacji napisanej w języku Java. Na początku zostaną omówione niektóre elementy wspomnianej aplikacji.

Zakresy widoczności

W języku Java są 4 zakresy widoczności:

- Publiczny oznaczany słowem kluczowym ***public*** – pole, metoda lub klasa oznaczone tym modyfikatorem jest dostępne **dla każdej innej klasy z każdego pakietu**,
- Pakietu oznaczany przez brak modyfikatora zakresu widoczności – pole, metoda lub klasa tego typu jest widoczne **dla wszystkich klas z tego samego pakietu**,
- Chroniony oznaczany słowem kluczowym **protected** – pole, metoda lub klasa oznaczone tym modyfikatorem jest dostępne **tylko dla klasy w, której jest ono/ona zdefiniowane lub dla klas pochodnych**,
- Prywatny oznaczany słowem kluczowym **private** – pole, metoda lub klasa oznaczone tym modyfikatorem jest dostępne **tylko dla klasy w której jest ono/ona zdefiniowane**.

Typ wyliczeniowy

W języku Java również jest dostępny typ wyliczeniowy, jednak różni się on od analogicznego typu w języku C++. W języku C++ typ wyliczeniowy jest traktowany jak typ prosty, natomiast w języku Java jest to **typ referencyjny**. Przykład przedstawia Listing 1.

Listing 1 Przykład typu wyliczeniowego

```
package pl.edu.pg.eti.ksg.po.lab2.biegpolesie;

/**
 * Rodzaje terenu na jakie mogą natknąć się uczestnicy biegu po lesie
 * @author TB
 */
public enum RodzajTerenu {
    /**
     * Teren po którym łatwo jest się porsuzać
     */
    DROGA,

    /**
     * Nieco mnie przebieżna niż {@link
pl.edu.pg.eti.ksg.po.lab2.biegpolesie.RodzajTerenu#DROGA}
     */
    SCIEZKA,

    /**
     * Teren w którym trzeba się ostrożnie pruszać
     */
    WYSOKI_LAS,
```

```
    /**
     * Teren trudny
     */
    NISKI_LAS,

    /**
     * Teren niebezpieczny, prawie niemożliwy do przebycia
     */
    BAGNO;
}
```

Kolekcje

W bibliotece standardowej języka Java znajduje się dużo gotowych klas implementujących kolekcję. Kolekcje implementują całą hierarchię interfejsów, które standaryzują operacje na kolekcja niezależnie od wybranej implementacji. Listing 2 przestawia zastosowanie klasy java.util.HashSet poprzez interfejs java.util.Set.

Listing 2 Zastosowanie klasy java.util.HashSet rzutowanej na interfejs java.util.Set

```
package pl.edu.pg.eti.ksg.po.lab2;

/*
 * ......
 */
public class Javalab2 {

    public static void main(String[] args) {

        Set<Uczestnik> uczestnicy = new HashSet<>();
        uczestnicy.add(new BagiennyBiegacz("Krzysztof", "Kowalski",
                                        Czlowiek.Plec.MEZCZYZNA));
        uczestnicy.add(new RobotMobilny(1));
        uczestnicy.add(new Terminator(3912));
        uczestnicy.add(new Student("Alfred", "Archiwista",
                                Czlowiek.Plec.MEZCZYZNA, "Historia"));
        uczestnicy.add(new StudentWETIOrazLesnik("Apoloniusz",
                    "Gałązka",Czlowiek.Plec.MEZCZYZNA, "Informatyka"));


       /*
        * ......
        */

        for(Uczestnik u : uczestnicy)
        {
            bieg.dodajUczestnika(u);
        }

        bieg.przeprowadzImpreze();
    }
}
```

Warto zwrócić uwagę na linię

```
Set<Uczestnik> uczestnicy = new HashSet<Uczestnik>();
```

Interfejs java.util.Set oraz klasa java.util.HashSet są typami generycznymi. Oznacza to, że jako parametr przyjmują one typ. Dzięki temu można powiedzieć, że obiekt na który wskazuje referencja uczestnicy zawiera elementy implementujące interfejs Uczestnik. Bez funkcjonalności typów generycznych konieczne by było rzutowanie obiektów na klasę java.lang.Object, a podczas odczytywania obiektów z kolekcji trzeba by było rzutować je z powrotem właściwą klasę lub interfejs (np. Uczestnik).

W Java 7 wprowadzono tzw. „Diamond operator", który pozwala na skracanie zapisu tworzenia niektórych obiektów generycznych.

```
Set<Uczestnik> uczestnicy = new HashSet<>();
```

W pakiecie java.util istnieje cała hierarchia interfejsów dotyczących kolekcji[1]. Dobrym przykładem elementu tej hierarchii jest interfejs java.lang.Iterable, który pozwala iterować po kolekcjach w następujący sposób:

```
for(Uczestnik u : uczestnicy)
{
    bieg.dodajUczestnika(u);
}
```

Klasy wewnętrzne, anonimowe, lambda wyrażenia, interfejsy funkcjonalne

Załóżmy istnienie następującego interfejsu (Listing 3).

Listing 3 Interfejs SluchaczZdarzen

```
package pl.edu.pg.eti.ksg.po.lab2.sup;
/*
 * Adnotacja @FunctionalInterface została wprowadzona w Javie 8.
 * Służy ona do oznaczania interfejsów z jedną metodą. Na takie
 * interfejsy  mogą zostać przekonwertowane lambda wyrażenia oraz metody
 * mające taką samą listę argumentów oraz typ zwracany.
 */
@FunctionalInterface
public interface SluchaczZdarzen {
    /*
     * Metoda interfejsu funkcjonalnego może mieć dowolną listę
     * parametrów oraz typ zwracany.
     */
    public void wystapiloZdarzenie(int priorytet, String nazwa);
}
```

[1] https://docs.oracle.com/javase/8/docs/api/java/util/package-tree.html, sekcja „Interface Hierarchy"

Interfejs ten zastosowano aby zaprezentować w jaki sposób zwykła klasa może subskrybować zdarzenia (Listing 4 oraz Listing 5).

Listing 4 ZwyklaKlasa która obsługuje zdarzenia poprzez interfejs SluchaczZdarzen na 6 różnych sposobów

```
package pl.edu.pg.eti.ksg.po.lab2.sup;
public class ZwyklaKlasa {

    private static int liczbaInstancji = 0;

    private int numerInstancji;

    /**
     * Klasa wewnętrzna ma dostęp do wszystkich pól i metod obiektu oraz
     * klasy. Nie musi implementować interfesju. Może mieć dowolny zasięg
     * widoczności.
     */
    private class KlasaWewnetrzna implements SluchaczZdarzen
    {
        private int poleKlasyWewnentrznej;

        public KlasaWewnetrzna(int poleKlasyWewnentrznej) {
            this.poleKlasyWewnentrznej = poleKlasyWewnentrznej;
        }

        @Override
        public void wystapiloZdarzenie(int pryiorytet, String nazwa) {
            System.out.println("KlasaWewnetrzna obsługuje zdarzenie "
                    +nazwa+" o priorytecie "+priorytet);
            System.out.println("Wartosci pól:");
            System.out.println("liczbaInstancji: "+ liczbaInstancji);
            System.out.println("numerInstancji: "+ numerInstancji);
            System.out.println("poleKlasyWewnentrznej: "+
                               poleKlasyWewnentrznej);
        }
    }

    /**
     * Statyczna klasa wewnętrzna ma tylko dostęp
     * do wszystkich statycznych pól i metod klasy.
     * Nie musi implementować interfesju.
     * Może mieć dowolny zasięg widoczności.
     */
    private static class StatycznaKlasaWewnetrzna
                                implements SluchaczZdarzen
    {
        private int poleStatycznejKlasyWewnentrznej;

        public StatycznaKlasaWewnetrzna(int p) {
            this.poleStatycznejKlasyWewnentrznej = p;
        }
```

```
        @Override
        public void wystapiloZdarzenie(int priorytet, String nazwa) {
            System.out.println("StatycznaKlasaWewnetrzna"+
              "obsługuje zdarzenie "+nazwa+
              " o priorytecie "+priorytet);
            System.out.println("Wartosci pól:");
            System.out.println("liczbaInstancji: "+ liczbaInstancji);
//Brak dostępu do pola numerInstancji - klasa wewnętrzna jest statycza
            //System.out.println("numerInstancji: "+ numerInstancji);
            System.out.println("poleStatycznejKlasyWewnentrznej: "+
                                    poleStatycznejKlasyWewnentrznej);
        }

    }
    public ZwyklaKlasa() {
        liczbaInstancji++;
        numerInstancji = liczbaInstancji;
    }
    public SluchaczZdarzen klasaWewnetrzna()
    {
        return new KlasaWewnetrzna(2*numerInstancji);
    }
    public SluchaczZdarzen statycznaKlasaWewnetrzna()
    {
        return new StatycznaKlasaWewnetrzna(3*liczbaInstancji);
    }
    public SluchaczZdarzen klasaAnonimowa()
    {
        /**
         * Obiekt klasy anonimowej tworzony w metodzie obiektu
         * nadrzędnego  ma dostęp do wszystkich pól i metod analogicznie
         * do klasy wewnętrznej. Klasy anonimowe mogą być utworzone na
         * bazie każdej klasy po której można dziedziczyć, bądź
         * interfejsu. Najczęściej jednak klasy te są
         * tworzone na bazie klas abstrakcyjnych i interfejsów.
         */
        return new SluchaczZdarzen() {

            int poleKlasyAnonimowej = 5*numerInstancji;

            @Override
            public void wystapiloZdarzenie(int priorytet, String nazwa)
            {
                System.out.println("KlasaAnonimowa obsługuje zdarzenie "
                 +nazwa+" o priorytecie "+priorytet);
                System.out.println("Wartosci pól:");
                System.out.println("liczbaInstancji: "+ liczbaInstancji);
                System.out.println("numerInstancji: "+ numerInstancji);
                System.out.println("poleKlasyAnonimowej: "+
                          poleKlasyAnonimowej);
            }
        };
    }
```

```
    public SluchaczZdarzen wyrazenieLambda()
    {
        /**
         * Wyrażenia lambda wprowadzono w Javie 8. Można przyjąć,
         * że jest to skrócona wersja klasy anonimowej
         * dla interfejsu oznaczonego andotacją
         * @FunctionalInterface
         */
        return (int p, String n) -> {
               System.out.println("Wyrażenie Lambda obsługuje zdarzenie "
                +n+" o priorytecie "+p);
                System.out.println("Wartosci pól:");
                System.out.println("liczbaInstancji: "+ liczbaInstancji);
                System.out.println("numerInstancji: "+ numerInstancji);
        };
    }

    public void zwyklaMetoda(int priorytet, String nazwa)
    {
        System.out.println("zwyklaMetoda obsługuje zdarzenie "+nazwa
                    +" o priorytecie "+priorytet);
        System.out.println("Wartosci pól:");
        System.out.println("liczbaInstancji: "+ liczbaInstancji);
        System.out.println("numerInstancji: "+ numerInstancji);
    }

    public static void zwyklaStatycznaMetoda(int priorytet,
                                             String nazwa)
    {
        System.out.println("zwyklaStatycznaMetoda obsługuje zdarzenie "
         +nazwa+" o priorytecie "+priorytet);
        System.out.println("Wartosci pól:");
        System.out.println("liczbaInstancji: "+ liczbaInstancji);
    }
}
```

Listing 5 Przykład subskrypcji zdarzenia oraz jego wywołania

```
package pl.edu.pg.eti.ksg.po.lab2.sup;

import java.util.LinkedList;
import java.util.List;

public class Javalab2_sup {
    public static void main(String[] args) {
        ZwyklaKlasa instancjaA = new ZwyklaKlasa();
        ZwyklaKlasa instancjaB = new ZwyklaKlasa();

        List<SluchaczZdarzen> listaSluchaczy = new LinkedList<>();

        listaSluchaczy.add(instancjaA.klasaWewnetrzna());
        listaSluchaczy.add(instancjaA.statycznaKlasaWewnetrzna());
        listaSluchaczy.add(instancjaA.klasaAnonimowa());
```

```
        listaSluchaczy.add(instancjaB.wyrazenieLambda());

        /**
         * Wprowadzone w Javie 8. Pozwala przekonwertować metodę obiektu
         * na klasę implementującą interfejs funkcjonalny.
         */
        listaSluchaczy.add(instancjaB::zwyklaMetoda);

        /**
         * Wprowadzone w Javie 8. Pozwala przekonwertować metodę
         * statyczną na klasę implementującą interfejs funkcjonalny
         */
        listaSluchaczy.add(ZwyklaKlasa::zwyklaStatycznaMetoda);

        String nazwa = "Test Interfejsów";
        int priorytet = 8;
        for(SluchaczZdarzen sl : listaSluchaczy)
        {
            sl.wystapiloZdarzenie(priorytet, nazwa);
        }
    }
}
```

Zdarzenia są jedną z zaawansowanych technik programowania obiektowego. Przykład rejestracji zdarzeń, który uwidacznia Listing 5, jest nieco uproszczony. Zazwyczaj klasa która generuje zdarzenia posiada prywatną kolekcję obiektów obsługujących zdarzenie, zaś dostęp do niej jest poprzez metodę publiczną. W języku Java zdarzenia nie są wspierane specjalną konstrukcją językową – trzeba do tego celu wykorzystać interfejsy. W języku C# do obsługi służy specjalne pole klasy oznaczane słowem kluczowym ***event***, które ukrywa całą implementacje obsługi zdarzeń.

Zadania na laboratorium

Zadania laboratoryjne będą polegać na modyfikacji istniejącej aplikacji. Aplikacją tą jest prosta symulacja o nazwie „Bieg po lesie”. Aplikacja ta symuluje wyścigi plenerowe po trasie złożonej z różnych form terenu. Celem uczestników jest jak najszybsze pokonanie trasy i dotarcie na metę. Zmagania uczestników są komentowane przez komentatora. Sami uczestnicy także wypowiadają się na temat elementów trasy i swoich zmagań.

Uwaga! Zadania przedstawione poniżej mogą być modyfikowane przez prowadzącego laboratorium.

Zadanie 0

Przeanalizuj kod aplikacji. Odnajdź tam klasę o zasięgu pakietowym. Zastanów się jakie to rodzi konsekwencje, oraz z jakiego powodu klasa ta ma ustawiony taki zasięg widoczności.

Zadanie 1 (2 punkty)

Dodaj do pakietu pl.edu.pg.eti.ksg.po.lab2.biegpolesie typ enum ***DziedzinaZadania***, który będzie zawierał następujące dziedziny:

- Matematyka,
- Fizyka,
- Informatyka,
- Sztuka,
- Nauki Leśne

Można dodać także inne dziedziny nauk (np. medycyna).

Stwórz następujące klasy i metody:

- Klasę implementująca interfejs ***ElementTrasy*** odpowiadająca za umieszczenie zadania na trasie, analogicznie do klasy ***Teren***,
- Metodę ***rozwiazZadanie*** w interfejsie ***Uczestnik.*** Metoda ta ma zwracać wartość boolean informującą, czy uczestnik rozwiązał zadanie z odpowiedniej dziedziny – analogicznie do metody ***predkoscPoruszaniaSie***. Zaimplementuj tą metodę we wszystkich klasach implementujących interfejs uczestnik lub dziedziczących po niej. Warto do tego zastosować generator pseudolosowy. Dla człowieka przyjmij szansę 10% na rozwiązanie zadania, zaś dla robota mobilnego 5% (robot mobilny nie umie czytać i zawsze losuje rozwiązanie).
- Dodaj w metodzie ***zmaganiaZElementemTrasy*** klasy ***BiegPoLesie*** obsługę zadań.
- Dodaj w klasie ***Komentator*** metodę ***relacjonuj*** dedykowaną dla zadań. Dodaj jej wywołanie w odpowiednim miejscu klasy ***BiegPoLesie*.**

Zadanie 2 (1 punkt)

Zmień dostęp do konstruktorów klas ***Teren*** oraz ***Zadanie*** na pakietowy. Utwórz klasę ***FabrykaElementowTrasy***, która będzie miała metody konstruujące obiekty implementujące interfejs ***ElementTrasy***. Metody te powinny przyjmować obiekt typu enum ***RodzajTerenu*** lub ***DziedzinaZadania***. Ponadto powinna istnieć trzecia metoda, która przyjmuje jako parametr obiekt typu String. Metoda ta powinna wyrzucać własny wyjątek, jeśli przetworzenie napisu na element trasy nie powiedzie się. Metody w klasie ***FabrykaElementówTrasy*** mogą być statyczne (choć nie muszą).

Popraw implementacje metod tworzących różne trasy, tak aby wykorzystywały stworzoną fabrykę. Dodaj w menu głównym 4 opcję polegająca na stworzeniu biegu przez użytkownika z wykorzystaniem metody konstruującej ElementTrasy na podstawie napisu. Pamiętaj o obsłudze wyjątków.

Zadanie 3 (1 punkt)

Wymagane zadanie 1!

Dodaj do klas implementujących uczestników 4 z poniżej wymienionych klas. Pamiętaj aby w metodach ***predkoscPoruszaniaSie*** oraz ***rozwiazZadanie*** znalazł się stosowny komentarz uczestnika do rodzaju terenu lub dziedziny zadania ;). Zaprezentuj ich działanie w programie.

Klasy dziedziczące po klasie **Człowiek**:

- **Student** – Dziedziczy po klasie **Czlowiek**. ze względu na ogólne oczytanie ma dwa razy większą szansę na rozwiązanie wszelkich zadań niż zwykły człowiek. Ma dodatkową właściwość – kierunek studiów.
- **StudentPolitechniki** – dziedziczy po klasie **Student**. W dziedzinach związanych z naukami ścisłymi ma większe szanse powodzenia. 50% z fizyki, oraz 70% z matematyki oraz informatyki. Lubi imprezy.
- **StudentWETI** – dziedziczy po klasie **StudentPolitechniki**. Ma zwiększona szansę na rozwiązanie zadań z informatyki (niemal 100%).
- **StudentWArch** – dziedziczy po klasie **StudentPolitechniki.** Wykazuje podobną sprawność w rozwiązywaniu zadań z Sztuki jak i z Matematyki i Informatyki.
- **StudentWETIOrazLesnik** – Student WETI absolwent Technikum Leśnego. Dziedziczy po klasie **StudentWETI**. Wykazują większa wiedzę w naukach leśnych oraz posiada większą sprawność w poruszaniu się w niskim lesie.
- **LesnyBiegacz** – dziedziczy po klasie **Człowiek**. Posiada większa sprawność w poruszaniu się w wysokim oraz niskim lesie, analogicznie jak zwykły człowiek po drodze oraz ścieżce. Poza tym szybciej niż zwykły człowiek przemierza drogę oraz ścieżkę.

Klasy dziedziczące po klasie **Robot**:

- **Terminator –** dziedziczy po klasie **Robot**. Zwyczajny robot z serii T-101. Posiada nieco większą sprawność w poruszaniu się w terenie niż zwykły człowiek. Rozwiązuje z 100% dokładnością zadania z nauk ścisłych. Nie jest w stanie sobie poradzić jednak ze sztuką oraz naukami leśnymi – losuje rozwiązanie.
- **TerminatorL –** dziedziczy po klasie **Terminator**. Niewielka seria T-101L zaprojektowana aby wspomagać leśniczych w ich pracach. Lepiej poruszają się w niskim lesie niż T-101. T-101L umieją też rozwiązywać zadania z dziedziny nauki leśnie z prawdopodobieństwem 60%.

Można też zastosować inne pomysły, takie jak LesnyBiegaczAbsolwentWETI – prędkość poruszania się jak u leśnego biegacza oraz znajomość informatyki oraz innych nauk ścisłych. Przy dodaniu innych dziedzin zadań można także stworzyć takie postaci jak BiegajacaFarmaceutka – ma zwiększoną prędkość poruszania się, gdy jest taka potrzeba przepłynie rzekę wpław, zna się na medycynie lub SpiewajacaFarmaceutka – zna się na medycynie i sztuce, śpiewa zamiast mówić. Kolejną możliwością jest Historyk – zna się na historii, w wysokim lesie porusza się wolniej gdyż poszukuje artefaktów i grodzisk oraz innych śladów człowieka z dawnych czasów.

Zadanie 4 (1 punkt)

W klasie **BiegPoLesie** dodaj obsługę zdarzenia wejścia na metę. Zarejestruj obiekt klasy komentator jako subskrybenta tego zdarzenia tak, aby je komentował. Zaprezentuj w metodzie **main** subskrypcję tego zdarzenia poprzez metodę publiczną. Wykonaj to dowolną metodą przedstawioną w sekcji *Klasy wewnętrzne, anonimowe, lambda wyrażenia, interfejsy funkcjonalne*.