Zaawansowane programowanie obiektowe i funkcyjne

Wykład 4: Klasy wewnętrzne i anonimowe

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.2
19 października 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Wstęp

- *Klasy wewnętrzne* powstały dla wygody programistów, którzy nie chcieli analizować kodu, rozproszonego po wielu plikach.
- Jednakże ich rozwój, stale motywowany wygodą programistów, doprowadził do powstania *klas anonimowych*, które zatrzęsły paradygmatami programowania w Javie.

Ponowne porządkowanie monet

- Określmy klasę monet

Plik `Coin`

```
1   public class Coin {
2       private int year;
3       private double value;
4       private String name;
5
6       public Coin(String name, int year, double value) {
7           this.year = year;
8           this.value = value;
9           this.name = name;
10      }
11   }
```

- Chcąc porządkować monety względem roku lub nazwy, musimy utworzyć dwa dodatkowe pliki

Plik `CoinDateComparator`

```
1   public class CoinDateComparator
        implements Comparator<Coin> {
2    @Override
3    public int compare(Coin o1, Coin o2) {
4     return o1.year-o2.year;
5    }
6   }
```

Plik `CoinNameComparator`

```
1   public class CoinNameComparator
        implements Comparator<Coin> {
2    @Override
3    public int compare(Coin o1, Coin o2) {
4     return o1.name.compareTo(o2.name);
5    }
6   }
```

Porządkowanie monet - klasy wewnętrzne

- Utrzymanie kodu byłoby znacznie wygodniejsze, gdybyśmy mogli przechowywać kod wszystkich klas w jednym pliku.

Plik `Coin`

```
1   public class Coin {
2
3       private int year;
4       private double value;
5       private String name;
6
7       public Coin(String name, int year, double value) {
8           this.year = year;
9           this.value = value;
10          this.name = name;
11      }
12
13      static class NameComparator implements Comparator<Coin> {
14          @Override
15          public int compare(Coin o1, Coin o2) {
16              return o1.name.compareTo(o2.name);
17          }
18      }
19      static class YearComparator implements Comparator<Coin> {
20          @Override
21          public int compare(Coin o1, Coin o2) {
22              return o1.year-o2.year;
23          }
24      }
25  }
```

- Umożliwiają to *klasy wewnętrzne*.

Klasy wewnętrzne

- Klasy wewnętrzne (*inner class*) definiujemy wewnątrz innej klasy.
- Jest to sensowne głównie w przypadku klas, które nie wchodzą w interakcję z innymi klasami niż klasa, w której są zagnieżdżone.
- Stosując klasy wewnętrzne zyskujemy
  - Dostęp do zmiennych prywatnych okalającej klasy.
  - Możliwość ukrycia klasy przed innymi klasami z pakietu.
  - Wygodniejszą pracę z logicznie pogrupowanym i przez to czytelniejszym kodem.

Rodzaje klas wewnętrznych

- Statyczne i dynamiczne klasy wewnętrzne
- Klasy lokalne (*local inner class*)
- Klasy anonimowe (*anonymous inner class*)

Wywoływanie statycznych klas wewnętrznych

- Dostęp do klasy wewnętrznej uzyskujemy odwołując się do nazw klasy zewnętrznej i wewnętrznej oddzielonych kropką `KlasaZewnetrzna.KlasaWewnetrzna`

Wywołanie klas wewnętrznych

```
1    myCollection.sort(new Coin.YearComparator());
2    myCollection.print();
3    myCollection.sort(new Coin.NameComparator());
4    myCollection.print();
```

- Wyniki sortowania:

`myCollection.sort(new Coin.YearComparator())`

1. Edward III florin (1343) - $6.8 million.
2. Brasher Doubloon (1787) - $7.4 million.
3. Flowing hair silver dollar (1794) - $10.0 million.
4. ...

`myCollection.sort(new Coin.NameComparator()))`

1. Brasher Doubloon (1787) - $7.4 million.
2. Double eagle (1933) - $7.4 million.
3. Edward III florin (1343) - $6.8 million.
4. ...

Enkapsulacja klas wewnętrznych

- Możemy zmodyfikować klasę `Coin`, aby odciąć dostęp do klas wewnętrznych

Fragment klasy `Coin`

```
1       private static class NameComparator implements Comparator<Coin> {
2           @Override
3           public int compare(Coin o1, Coin o2) {
4               return o1.name.compareTo(o2.name);
5           }
6       }
7
8       public static Comparator<Coin>  getNameComparator(){
9           return new NameComparator();
10      }
```

- Wykorzystanie klas wewnętrznych i ich istnienie jest ukryte za interfejsem klasy `Coin`:

Wywołanie metod klasy `Coin`

```
1   myCollection.sort(Coin.getNameComparator());
2   myCollection.print();
3   myCollection.sort(Coin.getYearComparator());
4   myCollection.print();
```

Pliki binarne klas wewnętrznych

- Kompilując klasy wewnętrzne, nadal tworzymy dla nich osobne pliki .class, tak jak w wypadku zwykłych klas.
- Nazwy plików tworzone są poprzez łączenie nazwy klasy zewnętrznej i wewnętrznej oddzielając je znakiem dolara.

Implementacja w oddzielnych źródłach

- Coin.class
- CoinDateComparator.class
- CoinNameComparator.class

Implementacja jako klasy wewnętrzne

- Coin.class
- Coin$DateComparator.class
- Coin$NameComparator.class

Statyczne a dynamiczne klasy wewnętrzne

- Omawiane przykłady to klasy statyczne, ale możemy też tworzyć klasy dynamiczne

Statyczne klasy wewnętrzne

- Mogą tworzyć instancje bez tworzenia instancji klasy zewnętrznej.
- Mają dostęp do statycznych elementów klasy zewnętrznej.

Dynamiczne klasy wewnętrzne

- Instancje niestatycznych klas wewnętrznych mogą istnieć tylko z instancją klasy zewnętrznej.
- Nie mogą posiadać elementów statycznych.
- Mają dostęp do wszystkich elementów klasy zewnętrznej.

- Dostęp uzyskiwany do elementów klas zewnętrznych nie jest dziedziczeniem!

Dynamiczne klasy wewnętrzne

- Zdefiniujmy klasę `OrderListener` wewnątrz klasy `TorpedoTube`

Klasa TorpedoTube

```
1   public class TorpedoTube {
2
3       private boolean flooded;
4
5       private void fireTorpedo(){
6           System.out.println("Torpedo fired!");
7       }
8
9       public class OrderListener implements ActionListener {
10
11          @Override
12          public void actionPerformed(ActionEvent e) {
13              if(flooded) fireTorpedo();
14          }
15      }
16  }
```

Instancja dynamicznej klasy wewnętrznej

- Możemy utworzyć instancję klasy wewnętrznej korzystając z `KlasaZew.KlasaWew obiektWewn = obiektZew.new KlasaWew()`

Utworzenie instancji

```
1   TorpedoTube.OrderListener orderListener = torpedoTube.new OrderListener();
```

Przesłanianie

- Możemy mieć metody o tej samej nazwie w klasie wewnętrznej i zewnętrznej.

Fragment klasy TorpedoTube

```
1  public class TorpedoTube {
2
3      public void getReport (){
4          System.out.println(flooded?"Ready to fire!":"Ready to reload");
5      }
6
7      public class OrderListener implements ActionListener {
8
9          public void getReport (){
10             System.out.println("Waiting for orders");
11         }
12
13         public void question(){
14             this.getReport();
15             TorpedoTube.this.getReport();
16         }
17     }
18 }
```

- Jak zachowa się wywołanie metody `question`?

```
Waiting for orders
Ready to reload
```

Operator `this`

- Operator `this` zawsze odnosi się do bieżącej klasy, czyli stosowany w klasie wewnętrznej odnosi się do klasy wewnętrznej.
- Chcąc odwołać się do klasy zewnętrznej należy zastosować konstrukcję `KlasaZewnetrzna.this`.

Motywacja

- Klasa wewnętrzna potrafi korzystać ze zmiennych klasy zewnętrznej, ale nader często oprócz atrybutów klasy wykorzystujemy zmienne lokalne.
- Zmienna lokalna istnieje tylko wewnątrz danego bloku (najczęściej metody).
- W sposób analogiczny do zmiennych lokalnych powstały *klasy lokalne*.

Klasa lokalna implementująca interfejs

- Załóżmy, że chcemy mieć metodę, która zwraca obiekt implementujący dany interfejs.
- Możemy stworzyć osobny plik z klasą, ale możemy też wykorzystać klasę lokalną.

Klasa `Painter`

```
1  public class Painter {
2      public Drawable createCircle(double r){
3          class Circle implements Drawable{
4              public void draw(){
5                  System.out.println("Drawing a circle, radius:"+r);
6              }
7          }
8          return new Circle();
9      }
10
11 }
```

- Klasa `Circle` nie będzie rozpoznawalna na zewnątrz metody `createCircle`, ale utworzone obiekty będą istniały jako implementacja interfejsu `Drawable`.

Wykorzystanie Klasy lokalnej

- Klasa `Circle` nie ma zdefiniowanego atrybutu `r`, ale mimo to używa go.

```
1   class Circle implements Drawable{
2    public void draw(){
3     System.out.println("Drawing a circle, radius:"+r);
4    }
5   }
```

- Spróbujmy narysować kilka kółek.

```
1   Drawable[] circles = {new Painter().createCircle(5),
2                         new Painter().createCircle(10)};
3
4   for (Drawable drawable : circles) {
5       drawable.draw();
6   }
```

- W wyniku otrzymamy różne koła o promieniu przekazanym lokalnie do metody `createCircle`.

```
Drawing a circle, radius:5.0
Drawing a circle, radius:10.0
```

Klasy lokalne

- Definiowane wewnątrz bloku programu.
- Mają dostęp do wszystkich elementów klasy zewnętrznej i do zmiennych lokalnych bloku.
- Widoczne tylko wewnątrz procedury, w której są zdefiniowane.
- Jeżeli blok jest statyczny, to klasa lokalna też jest statyczna.
- Nie mogą posiadać elementów statycznych i modyfikatorów.
- Kod binarny przechowywany jest w pliku postaci `KlasaZewnetrzna$N$KlasaWewnetrzna`,
  - gdzie `N` jest unikalnym znacznikiem klasy lokalnej wewnątrz klasy zewnętrznej.

Motywacja

- W omawianym przypadku, klasa lokalna służyła nam tylko jako chwilowy kontener do stworzenia implementacji interfejsu.
- Poza wnętrzem jednej metody nikt nigdy nie będzie wiedział, jak nazywała się ta klasa.
- A gdyby pójść krok dalej i utworzyć *klasy anonimowe* nieposiadające nazwy?

Tworzenie klasy anonimowej

- W zasadzie nie interesuje nas nazwa tworzonej klasy.
- Ważne jest tylko, jaki implementuje interfejs

```
1  new NazwaInterfejsu(){
2       /**
3          Implementacja metod
4       **/
5  }
```

- lub jaką klasę nadpisuje

```
1  new NazwaKlasyBazowej(arg1,...,argn){
2        /**
3           Definicja klasy
4        **/
5  }
```

Implementacja interfejsu przez klasę anonimową

- Rozszerzmy możliwości klasy `Painter` o rysowanie czerwonych kółek.

Fragment klasy `Painter`

```
1   public class Painter {
2        public Drawable createRedCircle(int r){
3
4            return new Drawable() {
5
6                public void draw() {
7                    System.out.println("Drawing a red circle, radius: " +r);
8                }
9            };
10       }
11  }
```

- Wykorzystujemy klasę anonimową w metodzie `createRedCircle`, aby uzyskać implementację interfejsu `Drawable`.
- Klasa anonimowa zachowuje możliwości klasy lokalnej co do odczytu zmiennych lokalnych.

Nadpisanie klasy przez klasę anonimową

- Nadpisanie klasy przez klasę anonimową może być czasami trudne do zauważenia.

Utworzenie instancji klasy

```
1   Person journalist = new Person("Clark Kent");
```

Utworzenie instancji klasy podrzędnej

```
1   Person superhero = new Person("Clark Kent"){
2    public String getNick(){
3     return "Superman";
4    }
5   };
```

Przesłanianie

- Niech klasa `Person` ma jeden atrybut `name`, przekazywany przez konstruktor i metodę `getName` go zwracającą.
- Utwórzmy instancję klasy i użyjmy klasy anonimowej.

Utworzenie instancji i instancji klasy podrzędnej

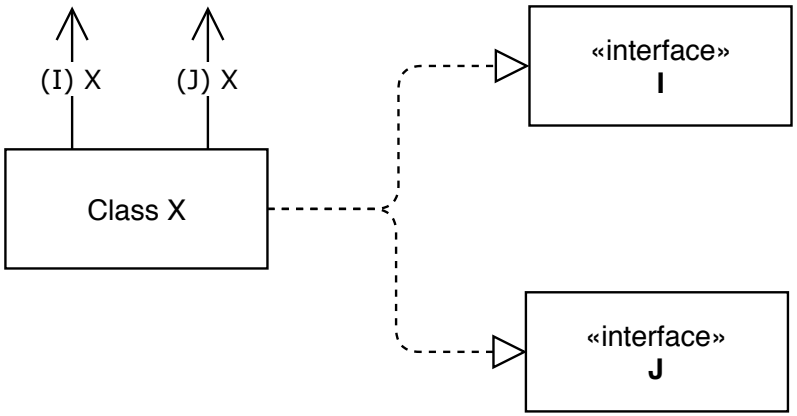
```
1
2   Person journalist = new Person("Clark Kent");
3   Person superhero = new Person("Clark Kent"){
4     public String getNick(){
5       return "Superman";
6     }
7     public String getName(){
8         return getNick();
9     }
10  };
```

- Co się stanie, gdy wywołamy metodę `getName` dla obu instancji?

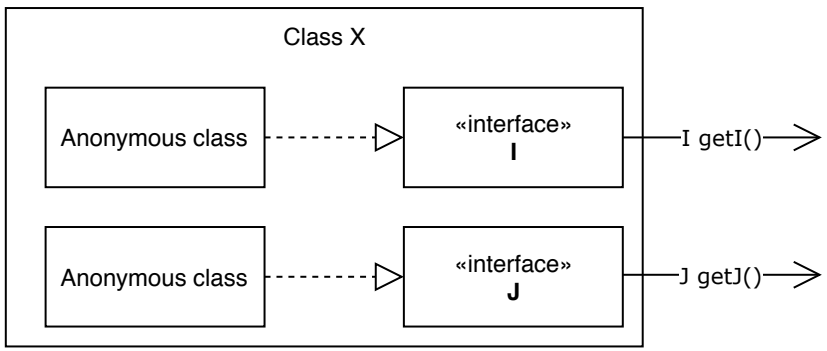
```
Clark Kent
Superman
```

- Praca domowa: Jak ustalić prawdziwą tożsamość Supermana?

Konsekwencje istnienia klas anonimowych - interfejsy



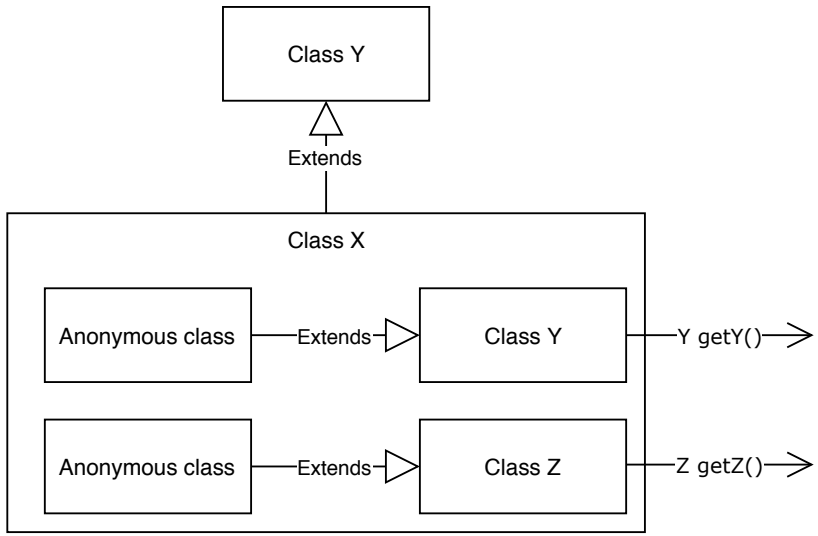
Rysunek 1: *Implementacja interfejsów przez klasę X*



Rysunek 2: *Implementacja interfejsów przez klasy anonimowe*

- Klasa `X` może implementować interfejsy `I` i `J`.
- Możemy też utworzyć metody zwracające anonimową implementację interfejsów.
- Ze względu na dostęp klasy wewnętrznej do zmiennych klasy zewnętrznej `(I)x` i `x.getI()` są równoważne.

Konsekwencje istnienia klas anonimowych - klasy



Rysunek 3: *Rozszerzanie klas przez klasę* `X`

- Klasa `X` może nadpisać tylko jedną z klas `Y` i `Z`.
- Możemy jednak utworzyć metody zwracające anonimowe klasy nadpisujące klasy `Y` i `Z`.
- Wykorzystując dostęp do danych prywatnych klasy zewnętrznej potrafimy symulować dziedziczenie z wielu klas.

Podsumowanie

- *Klasy wewnętrzne* umożliwiają szybsze i wygodniejsze pisanie kodu.
- Obecnie często zastępuje się je wyrażeniami lambda, które są jeszcze wygodniejsze i szybsze w użyciu
- Należy pamiętać, że zastosowanie klas anonimowych pozwala obiektowi przedstawiać się jako potomek wielu klas.

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.