Zaawansowane programowanie w Javie

Studia zaoczne - Wykład 1

dr hab. Andrzej Zbrzezny, profesor UJD

Katedra Matematyki i Informatyki
Uniwersytet Jana Długosza w Częstochowie

16 marca 2024

Zalecana Literatura

Literatura podstawowa

- Cay Horstmann.
  Java. Przewodnik doświadczonego programisty. Wydanie III.
  Wydawnictwo Helion. Gliwice, październik 2023.
- Joshua Bloch.
  Java. Efektywne programowanie. Wydanie III.
  Wydawnictwo Helion. Gliwice, sierpień 2018.
- Cay Horstmann.
  Java. Podstawy. Wydanie XII.
  Wydawnictwo Helion. Gliwice, grudzień 2022.
- https://dev.java/

Interfejsy i wyrażenia lambda

Java 8. Przewodnik doświadczonego programisty – rozdział 3

- http://pdf.helion.pl/jav8pd/jav8pd.pdf

Spis treści rozdziału *Interfejsy i wyrażenia lambda*

Interfejsy i wyrażenia lambda

Wprowadzenie

- Java została zaprojektowana jako obiektowy język programowania w latach 90. ubiegłego wieku.
- W tym czasie programowanie obiektowe było najważniejszym paradygmatem w tworzeniu oprogramowania.
- Interfejsy są kluczową funkcjonalnością w programowaniu obiektowym.
- Pozwalają na określenie, co ma zostać wykonane bez konieczności tworzenia implementacji.

Interfejsy i wyrażenia lambda

Wprowadzenie

- Długo przed pojawieniem się programowania obiektowego istniały funkcyjne języki programowania (takie jak Lisp).
- W językach tych to funkcje, a nie obiekty, były najważniejszym mechanizmem tworzącym strukturę programu.
- Ostatnio programowanie funkcyjne jest coraz ważniejsze, ponieważ dobrze sprawdza się w przypadku programowania równoległego i zdarzeniowego („reaktywnego”).
- Java (począwszy od wersji 8) wspiera wyrażenia funkcyjne, które stanowią wygodne połączenie pomiędzy programowaniem obiektowym a funkcyjnym.

1. Interfejsy

- Interfejs to mechanizm pozwalający na zapisanie kontraktu pomiędzy dwoma stronami: dostawcą usług i klasami, które chcą, by ich obiekty mogły być wykorzystywane z usługą.
- Popatrzmy na usługę, która operuje na ciągu liczb całkowitych, dając informację o średniej z pierwszych n wartości:

```
public static double average(IntSequence seq, int n)
```

Takie sekwencje mogą przyjmować wiele form:

- ciąg liczb całkowitych wpisany przez użytkownika,
- ciąg wylosowanych liczb całkowitych,
- ciąg liczb pierwszych,
- ciąg elementów w tablicy zmiennych typu całkowitego,
- ciąg kodów znaków w postaci ciągu znaków (ang. string),
- ciąg cyfr w liczbie.

1. Interfejsy

- Chcemy zaimplementować jeden mechanizm obsługujący wszystkie powyższe rodzaje danych.
- Najpierw zobaczmy, jakie wspólne cechy mają ciągi liczb całkowitych. Aby móc obsłużyć taki ciąg, potrzebne są co najmniej dwie metody:
  - sprawdzająca, czy istnieje kolejny element,
  - pobierająca kolejny element.
- Aby zadeklarować interfejs, dostarczamy nagłówki metod w taki sposób:

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

Interfejsy i wyrażenia lambda

Interfejsy

- Nie musimy implementować tych metod, ale jeżeli chcemy, możemy dopisać domyślną implementację.
- Jeśli nie ma domyślnej implementacji, mówimy, że metoda jest abstrakcyjna.
- Wszystkie metody interfejsu automatycznie stają się publiczne.
- Dzięki temu nie trzeba dopisywać przy metodach `hasNext` oraz `next` modyfikatora **`public`**.
- Niektórzy programiści dopisują to dla zwiększenia przejrzystości kodu.

Interfejsy i wyrażenia lambda

Interfejsy

- Klasy `SquareSequence` oraz `DigitSequence` implementują wszystkie metody interfejsu `IntSequence`.
- Jeśli klasa implementuje tylko niektóre z metod, musi być zadeklarowana z modyfikatorem **`abstract`**.
- Istnieje nieskończenie wiele liczb, które są kwadratem innej liczby całkowitej, a obiekt tej klasy może zwracać kolejne takie liczby.
- Słowo kluczowe **`implements`** mówi o tym, że klasa będzie obsługiwała interfejs `IntSequence`.
- Klasa implementująca musi deklarować metody interfejsu jako publiczne. W przeciwnym wypadku będą one miały zasięg pakietu.
- Ponieważ interfejs wymaga, by metoda była publiczna, kompilator zgłosi błąd.

Interfejsy i wyrażenia lambda

Konwersja do typu interfejsu

- Rozważmy program `IntSequenceDemo.java`
- Popatrzmy na zmienną `digits`. Jej typ to `IntSequence`, nie `DigitSequence`.
- Zmienna typu `IntSequence` odwołuje się do obiektu dowolnej klasy implementującej interfejs `IntSequence`.
- Można zawsze przypisać do zmiennej obiekt, którego typ jest określony implementowanym interfejsem, lub przekazać go do metody oczekującej zmiennej z takim interfejsem.

Interfejsy i wyrażenia lambda

Konwersja do typu interfejsu

- A oto odrobina przydatnej terminologii. Typ `S` to typ nadrzędny typu `T` (podtypu), jeśli dowolna wartość podtypu może być przypisana do zmiennej typu nadrzędnego bez konwersji.
- Przykładowo, interfejs `IntSequence` jest typem nadrzędnym klasy `DigitSequence`.
- Choć można deklarować zmienne, używając interfejsu jako ich typu, nie jest możliwe utworzenie instancji obiektu, którego typem będzie interfejs. Wszystkie obiekty muszą być instancjami klas.

Interfejsy – rzutowanie i operator `instanceof`

- Czasem będziemy potrzebowali odwrotnej konwersji – z typu nadrzędnego do podtypu. Wtedy stosujemy rzutowanie.
- Na przykład jeśli zdarzy się, że obiekt wskazywany przez zmienną typu `IntSequence` jest w rzeczywistości typu `DigitSequence`, można wykonać konwersję typu w taki sposób:

```
IntSequence sequence = new DigitSequence(2017);
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

- W tej sytuacji rzutowanie było potrzebne, ponieważ `rest` to metoda klasy `DigitSequence`, ale nie ma jej w `IntSequence`.

Interfejsy – rzutowanie i operator `instanceof`

- Można wykonać rzutowanie obiektu jedynie do typu jego rzeczywistej klasy lub jednego z jego typów nadrzędnych.
- Jeżeli wykonamy nieprawidłowe rzutowanie, zostanie zgłoszony błąd kompilacji lub wyjątek rzutowania klasy:

```
IntSequence sequence = ...;
// Nie może zadziałać, ponieważ IntSequence
// nie jest typem nadrzędnym dla String
String digitString = (String) sequence;
// Może zadziałać, ale zwróci wyjątek
// class cast exception, jeśli się nie powiedzie
RandomSequence rs = (RandomSequence) sequence;
```

Interfejsy – rzutowanie i operator `instanceof`

- Aby uniknąć zgłoszenia wyjątku, można przed wykonaniem rzutowania sprawdzić za pomocą operatora **`instanceof`**, czy jest to możliwe.
- Wyrażenie

```
obiekt instanceof Typ
```

  zwraca `true`, jeśli obiekt jest instancją klasy, dla której `Typ` jest typem nadrzędnym.
- Warto to sprawdzać przed wykonaniem rzutowania.

```
IntSequence sequence = ...;
if (sequence instanceof RandomSequence) {
    RandomSequence rs = (RandomSequence) sequence;
}
```

Rozszerzanie interfejsów

- Interfejs może rozszerzać inny interfejs, dokładając dodatkowe metody do oryginalnych. Przykładowo, `Closeable` to interfejs z jedną metodą:

```
public interface Closeable {
    void close();
}
```

- Jak zobaczymy, jest to ważny interfejs wykorzystywany do zwalniania zasobów w sytuacji, gdy wystąpi wyjątek. Interfejs `Channel` rozszerza ten interfejs:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

Implementacja wielu interfejsów

- Klasa może implementować dowolną liczbę interfejsów.
- Na przykład klasa `FileSequence`, która wczytuje liczby całkowite z pliku, może implementować interfejsy `Closeable` oraz `IntSequence`:

```
public class FileSequence implements
    IntSequence, Closeable
{
    ...
}
```

- W takiej sytuacji klasa `FileSequence` ma dwa typy nadrzędne: `IntSequence` i `Closeable`.

Stałe

- Każda zmienna zdefiniowana w interfejsie automatycznie otrzymuje atrybuty **`public`**, **`static`** oraz **`final`**.
- Na przykład interfejs `SwingConstants` definiuje stałe opisujące kierunki na kompasie:

```
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    ...
}
```

- Można się do nich odwoływać się za pomocą pełnej nazwy `SwingConstants.NORTH`.

Interfejsy i wyrażenia lambda

Stałe

- Jeśli nasza klasa zechce implementować interfejs `SwingConstants`, można opuścić przedrostek `SwingConstants` i napisać jedynie `NORTH`.
- Nie jest to jednak często wykorzystywane.
- Dużo lepiej w przypadku zestawu stałych wykorzystać typ wyliczeniowy.
- Nie można umieścić w interfejsie zmiennych instancyjnych, ponieważ interfejs określa zachowanie a nie stan obiektu.

Metody statyczne i domyślne

- W starszych wersjach języka Java wszystkie metody interfejsu musiały być abstrakcyjne – to znaczy bez implementacji.
- Obecnie można dodawać metody z implementacją na dwa sposoby: jako metody statyczne i metody domyślne.
- Nigdy nie było technicznych przeszkód, aby interfejs mógł posiadać metody statyczne, ale nie pasowały one do roli interfejsów jako abstrakcyjnej specyfikacji.
- To podejście się zmieniło. Szczególnie metody wytwórcze pasują do interfejsów.

Metody statyczne i domyślne

- Na przykład interfejs `IntSequence` może mieć statyczną metodę `digitsOf` generującą ciąg cyfr z przekazanej liczby całkowitej:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

- Metoda zwraca instancję klasy implementującej interfejs `IntSequence`, ale przy wywoływaniu nie ma znaczenia, która to będzie klasa.

```
public interface IntSequence {
    ...
    public static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
```

Metody statyczne i domyślne

- Można dostarczyć domyślną implementację dowolnej metody interfejsu. Taką metodę trzeba oznaczyć modyfikatorem **`default`**.

```
public interface IntSequence {
    // Domyślnie sekwencje są nieskończone
    default boolean hasNext() {
        return true;
    }
    int next();
}
```

- Klasa implementująca ten interfejs może przesłonić metodę `hasNext` lub odziedziczyć domyślną implementację.

Metody statyczne i domyślne

- Wprowadzenie możliwości definiowania metod domyślnych czyni przestarzałym klasyczny wzorzec polegający na tworzeniu interfejsu i klasy, która implementuje większość lub wszystkie jego metody.
- Wzorzec ten zastosowany był w przypadku interfejsu `Collection` i klasy `AbstractCollection` oraz interfejsu `WindowListener` i klasy `WindowAdapter` w Java API.
- Obecnie należy po prostu implementować metody w interfejsie.
- Ważnym zastosowaniem domyślnych metod jest umożliwienie modyfikowania interfejsów.

Metody statyczne i domyślne

- Popatrzmy przykładowo na interfejs `Collection`, który jest częścią języka Java od wielu lat.
- Załóżmy, że jakiś czas temu utworzyliśmy klasę `Bag`:

```
public class Bag implements Collection
```

- Później, w Java 8, do interfejsu dodano metodę `stream`.
- Załóżmy, że metoda `stream` nie ma domyślnej implementacji. W takiej sytuacji klasa `Bag` nie skompiluje się, ponieważ nie implementuje nowej metody.
- Dodanie do interfejsu metody bez domyślnej implementacji powoduje, że nie zostanie zachowana kompatybilność źródeł (ang. source-compatible).

Metody statyczne i domyślne

- Załóżmy jednak, że nie rekompilujemy klasy i po prostu korzystamy ze starego pliku JAR zawierającego skompilowaną klasę.
- Klasa nadal będzie się ładować, nawet bez brakującej metody. Programy wciąż mogą tworzyć instancje klasy `Bag` i nic nie będzie się działo.
- Oznacza to, że dodanie metody do interfejsu zachowuje kompatybilność binariów (ang. binary-compatible).
- Jeśli jednak program wywoła metodę `stream` na instancji klasy `Bag`, wystąpi błąd `AbstractMethodError`.

Metody statyczne i domyślne

- Uczynienie metody domyślną rozwiązuje oba problemy. Klasa `Bag` znowu będzie się kompilowała.
- A jeśli klasa będzie załadowana bez ponownej kompilacji i zostanie wywołana metoda `stream` na instancji klasy `Bag`, wykonany zostanie kod metody `Collection.stream`.

Rozstrzyganie konfliktów metod domyślnych

- Jeśli klasa implementuje dwa interfejsy, z których jeden ma domyślną metodę, a drugi metodę (domyślną lub nie) z taką samą nazwą i typami argumentów, to musimy rozstrzygnąć konflikt.
- Nie zdarza się to zbyt często i zazwyczaj dość łatwe jest rozwiązanie tej sytuacji.
- Popatrzmy na przykład. Załóżmy, że mamy interfejs `Person` z metodą `getId`:

```
public interface Person
{
    String getName();
    default int getId() { return 0; }
}
```

Rozstrzyganie konfliktów metod domyślnych

- Załóżmy też, że mamy interfejs `Identified`, również obejmujący taką metodę:

```
public interface Identified
{
    default int getId() {
        return Math.abs(hashCode());
    }
}
```

- Przypomnijmy, że metoda `hashCode` jest dziedziczona z klasy `Object` i zwraca dla danego obiektu liczbę całkowitą typu `int`.

Rozstrzyganie konfliktów metod domyślnych

- Co się dzieje, gdy tworzymy klasę implementującą oba te interfejsy?

```
public class Employee
    implements Person, Identified
{
    ...
}
```

- Klasa dziedziczy dwie metody `getId` dostarczone przez interfejsy `Person` i `Identified`.
- Nie ma sposobu, by kompilator mógł wybrać, która z nich jest lepsza. Kompilator zgłasza błąd i pozostawia programiście rozwiązanie tego problemu.

Rozstrzyganie konfliktów metod domyślnych

- Można utworzyć metodę `getId` w klasie `Employee` a w niej zaimplementować własny mechanizm nadawania identyfikatorów lub przekazać to do jednej z wywołujących konflikt metod w taki sposób:

```
public class Employee
    implements Person, Identified
{
    public int getId() {
        return Identified.super.getId();
    }
    ...
}
```

Interfejsy i wyrażenia lambda

Rozstrzyganie konfliktów metod domyślnych

- Słowo kluczowe `super` pozwala na wywołanie metody typu nadrzędnego. W takim przypadku musimy określić, który typ nadrzędny chcemy wykorzystać.
- Składnia może nie wygląda najlepiej, ale jest spójna ze składnią wywoływania metod klasy nadrzędnej.

Rozstrzyganie konfliktów metod domyślnych

- Załóżmy teraz, że interfejs `Identified` nie zawiera domyślnej implementacji `getId`:

```
interface Identified {
    int getId();
}
```

- Czy klasa `Employee` może odziedziczyć metodę domyślną z interfejsu `Person`?
- Na pierwszy rzut oka może to się wydać rozsądne. Ale skąd kompilator ma wiedzieć, czy metoda `Person.getId` robi to, czego oczekuje się od metody `Identified.getId`?
- Może ona przecież zwracać na przykład informację o tym, z czym dana osoba się identyfikuje, a nie jej numer identyfikacyjny.

Rozstrzyganie konfliktów metod domyślnych

- Projektanci języka Java postawili na bezpieczeństwo i spójność.
- Nie ma znaczenia, jaki konflikt występuje między dwoma interfejsami; jeżeli przynajmniej jeden z interfejsów zawiera implementację, to kompilator zgłasza błąd i pozostawia programiście rozwiązanie problemu.
- Jeżeli żaden z interfejsów nie zawiera domyślnej lub współdzielonej metody, konflikt się nie pojawi.
- Implementując klasę, można zaimplementować metodę lub pozostawić ją bez implementacji i zadeklarować klasę jako abstrakcyjną.

Interfejsy i wyrażenia lambda

Rozstrzyganie konfliktów metod domyślnych

- Jeśli klasa rozszerza klasę nadrzędną i implementuje interfejs, dziedzicząc taką samą metodę z obu, to reguły są prostsze.
- W takim przypadku liczą się tylko metody klasy nadrzędnej, a metody domyślne z interfejsu są po prostu ignorowane.
- Jest to w rzeczywistości częstszy przypadek niż konflikty między interfejsami.

Metody prywatne

- Od wersji Java 9 metody w interfejsie mogą być prywatne.
- Metoda prywatna może być metodą statyczną lub instancyjną, ale nie może być metodą domyślną, ponieważ może ona zostać nadpisana.
- Ponieważ metody prywatne mogą być używane tylko w metodach samego interfejsu, ich użycie jest ograniczone do bycia metodami pomocniczymi dla innych metod interfejsu.

Metody prywatne

- Przykładowo, załóżmy, że klasa `IntSequence` udostępnia metody

```
static of(int a)
static of(int a, int b)
static of(int a, int b, int c)
```

- Wtedy każda z tych metod mogłaby wywołać metodę pomocniczą

```
private static IntSequence
makeFiniteSequence(int... values)
{
    ...
}
```

Metody statyczne i domyślne

- Rozstrzyganie konfliktów metod domyślnych
  - `Person.java`
  - `Identified.java`
  - `Employee.java`

Interfejsy i wyrażenia lambda

Przykłady interfejsów

- Interfejs `Comparable`
  - `Employee.java`
- Interfejs `Comparator`
  - `SortDemo.java`
- Interfejs `Runnable`
  - `RunnableDemo.java`
- Wywołania zwrotne interfejsu użytkownik
  - `ButtonDemo.java`