Zaawansowane programowanie obiektowe i funkcyjne

Wykład 5: Programowanie funkcyjne. Wyrażenie lambda i interfejsy funkcyjne

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

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.1
5 marca 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.

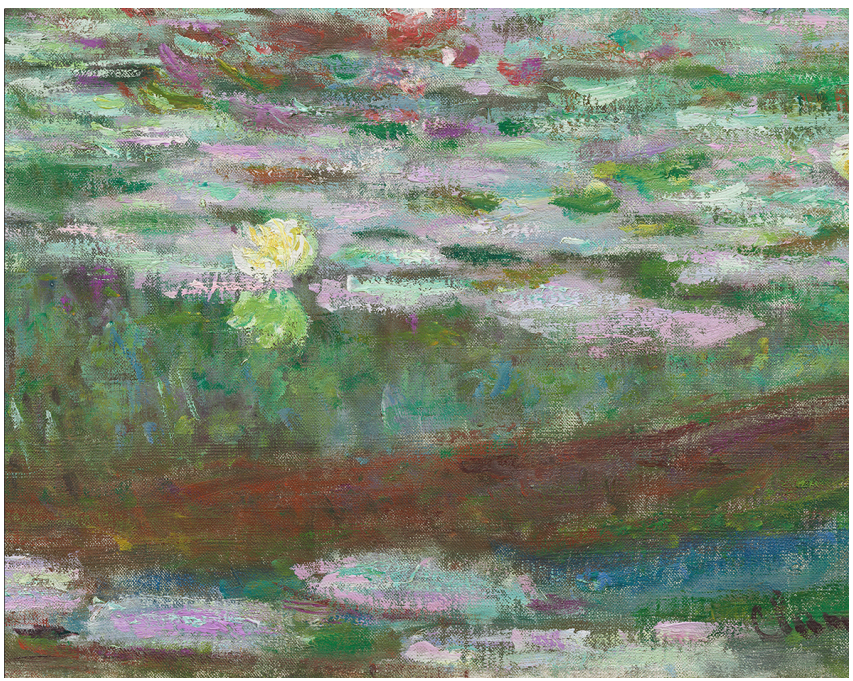
Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Rzeczpospolita
Polska

Politechnika
Warszawska

Wstęp

- Duża część pracy programisty skupia się na tworzeniu narzędzi potrzebnych do osiągnięcia celu zamiast na samym celu.
- To trochę tak jakby uznać, że technika malarska sama w sobie jest ważniejsza niż całokształt obrazu.



[?]

Programowanie funkcyjne

- Skupia się na dopowiedzi na pytanie co zrobić? zamiast jak to zrobić?.
- Co za tym idzie chcemy jak najbardziej zredukować obudowę technologiczną naszego programowania i dążymy do stworzenia pojedynczej funkcji realizującej określone zadania.
- Pojedynczą funkcję łatwo rozpropagować przy programowaniu obliczeń równoległych.
- Na funkcjach opiera się funkcyjne programowanie reaktywne (ang. *Functional reactive programming*), gdzie zagnieżdżone wywołania funkcji sterują przepływem i propagacją danych [?].

Programowanie funkcyjne w Javie

- Java została stworzona jako język silnie ukierunkowany na programowanie obiektowe.
- Programowanie funkcyjne nie było wspierane aż do Java 8, która wprowadza wyrażenia funkcyjne – wyrażenia lambda

Wyrażenie Lambda

Definicja

Wyrażenie lambda jest blokiem kodu przeznaczonym do późniejszego, wielokrotnego, wykorzystania

- Ponieważ Java jest zorientowana obiektowo to blok kodu jest instancją implementującą określony interfejs.
- Wyrażenia Lambda dostarczają poręczną składnię do tworzenia takich instancji

Składnia

Wyrażenie lambda składa się z:

- Listy parametrów oddzielonych przecinkami i ograniczonych nawiasami,
- Operatora strzałki,
- Ciała, składającego się z pojedynczego wyrażenia lub bloku kodu

Przykład

```
1  (int a, int b)-> {return a+b}
```

Parametry

- Typ parametrów określamy bezpośrednio w liście parametrów.
- Dopuszczalne są następujące wyjątki.
  - brak parametrów:

```
1  Runnable task =
2      ()-> {for(int i = 0; i < 10; i++) println(i);}
```

  - domyślne parametry

```
1  Comparator<String> comp =
2      (s1, s2) -> s1.length() - s2.length();
```

  - pojedynczy domyślny parametr

```
1  EventHandler<ActionEvent> listener =
2      event ->
3  System.out.println("I have bad feelings about this!");
```

Typ wynikowy

- Nigdy nie określamy bezpośrednio jakiego typu wartość zwrócimy.
- Kompilator określa typ na podstawie analizy kodu i weryfikuje czy otrzymał oczekiwany typ.
- Użycie `return` jest konieczne tylko w wypadku wieloliniowego kodu.

```
1  (String s1, String s2) -> {
2      int diff = s1.length() < s2.length();
3      if (diff < 0) return -1;
4      else if (diff > 0) return 1;
5      else return 0;
6  }
```

Zasięg wyrażeń lambda

- Zasięg wyrażeń lambda jest taki sam jak zasięg zmiennych lokalnych.
- Oceńmy poniższy zapis

```
1  int first = 0;
2
3  Comparator<String> comp = (first, second) ->
       first.length() - second.length();
```

- Jest on błędny, bo zmienna `first` już istnieje, a nadajemy tę nazwę parametrowi wyrażenia.

Użycie this

- Wewnątrz ciała wyrażeń lambda możemy używać this.

```
1  public class Application() {
2    public void doWork() {
3      Runnable runner = () -> {
           System.out.println(this.toString());
4      };
5    }
6  }
```

- W tym wypadku this odnosi się do instancji klasy Application.
- Zwróćmy uwagę, że jest to inne zachowanie niż w przypadku klas wewnętrznych.

Wykorzystywanie zmiennych zewnętrznych

- Wyrażenia lambda mogą korzystać ze zmiennych zewnętrznych

```
1  public static void repeatMessage(String text, int count) {
2    Runnable r = () -> {
3      for (int i = 0; i < count; i++) {
4        System.out.println(text);
5        }
6      };
7    new Thread(r).start();
8  }
```

- Jednakże wyrażenie lambda jest fragmentem kodu do wielokrotnego wykorzystywania.
- Jak zatem przechowuje informację o zmiennych zewnętrznych?

Domknięcie

- Tworząc wyrażenie lambda musimy zapamiętać:
  1. blok kodu,
  2. parametry,
  3. wartości zmiennych zewnętrznych
- Blok kodu plus wartości zmiennych tworzą domknięcie.

Ograniczenia domknięcia

- Wyrażenia lambda mogą używać zewnętrznych zmiennych tylko jeżeli są one faktycznie finalne
  - Oznacza to, że zmienna nie musi być oznaczona jako `final`, ale mogłaby być bo od jej inicjacji nie zmienia wartości.
  - Zatem wartość zmiennej nie może się zmieniać podczas jej wykorzystywania przez wyrażenie lambda.
  - Także wyrażenie lambda nie może zmieniać wartości zmiennej.
- Czy poniższy kod jest poprawny?

```
1  for (int i = 0; i < n; i++) {
2    new Thread(() -> System.out.println(i)).start();
3  }
```

- Nie, gdyż zmienna `i` zmienia wartość.
- Czy nie da się używać wyrażenia lambda wewnątrz pętli?

Poprawne używanie wyrażeń wewnątrz pętli

- Możemy używać wyrażeń lambda w pętlach przechodzących po liście parametrów.

```
1  for (String arg : args) {
2    new Thread(() -> System.out.println(arg)).start();
3  }
```

- Powyższy kod zadziała poprawnie ponieważ zmienna `arg` jest tworzona na nowo w każdej iteracji.

Interfejs funkcyjny

Definicja

Interfejs z dokładnie jedną metodą abstrakcyjną[1]

- Akceptowanym wyjątkiem jest posiadane metod zdefiniowanych w klasie `Object`, ponieważ każda klasa implementująca interfejs ma je i tak zaimplementowane.

[1] Pamiętajmy, że Java 8 dopuszcza implementacje domyślne w interfejsach

Przykład

Implementacja interfejsu funkcyjnego

```
1  button.addActionListener(
2    new ActionListener() {
3      public void actionPerformed(ActionEvent e) {
4        System.out.println("Action performed by " +
             e.getSource());
5      }
6    }
7  )
```

Wyrażenie lambda

```
1  button.addActionListener(e -> {
2    System.out.println("Action performed by " +
         e.getSource());
3  });
```

Wywołanie wyrażenia lambda

- Wyrażenie lambda możemy wywołać wtedy i tylko wtedy gdy jest ono implementacją jakiegoś interfejsu funkcyjnego.
- Na szczęście istnieje wiele predefiniowanych bardzo ogólnych interfejsów, które możemy wykorzystać.
- Możemy też stworzyć własny interfejs funkcyjny.

Popularne interfejsy funkcyjne[2]

Interfejs	In	Out	Opis
Runnable	-	void	wykonanie akcji
Supplier<T>	none	T	dostarczenie T
Consumer<T>	T	void	konsumpcja T
BiConsume<T, U>	T, U	void	konsumpcja T i U
Function<T, R>	T	R	funkcja z parametrem T
UnaryOperator<T>	T	T	operator T
BiFunction<T, U, R>	T, U	R	funkcja z parametrami T i U
BinaryOperator<T>	T, T	T	operator binarny T
Predicate<T>	T	boolean	funkcja logiczna
BiPredicate<T, U>	T, U	boolean	dwuarg. fun. logiczna

[2]Istnieją też interfejsy dla typów podstawowych [?]

Dodatkowe możliwości interfejsów funkcyjnych

- Interfejsy funkcyjne oferują szersze możliwości dzięki domyślnym implementacjom dodatkowych metod.
- Interfejs `Predicate` określa wartość logiczną zwracaną przez abstrakcyjną metodę `isEqual`.
- Jednocześnie implementuje metody `or`, `and` i `negate`.
- Pozwala to tworzyć złożone wyrażenia logiczne

```
1  Predicate.isEqual(a).or(Predicate.isEqual(b)).
```

- Możemy wyliczyć wartościowanie powyższego wyrażenia logicznego korzystając z wyrażenia lambda

```
1  x -> a.equals(x) || b.equals(x).
```

Przykład użycia interfejsu dla typu prostego

- Interfejs `IntConsumer` pozwala zdefiniować metodę wykorzystującą wartość całkowitą.
- Użyjmy go, aby wielokrotnie wywołać blok kodu.

```
1  public static void repeat(int n, IntConsumer action) {
2      for (int i = 0; i < n; i++) action.accept(i);
3  }
```

- Możemy teraz użyć funkcji `repeat` aby *n*-krotnie powtórzyć jakąś czynność dla kolejnych liczb całkowitych:

```
1  repeat(10, i -> System.out.println("Final countdown: "
       + (9 - i)));
```

Nadpisanie interfejsu funkcyjnego

- Nadpisując interfejs funkcyjny `TemporalAdjuster` możemy tworzyć własne warunki wyszukiwania daty.

Najbliższy weekend

```
1  TemporalAdjuster NEXT_WEEKEND =
2    w -> {EnumSet<DayOfWeek> weekend =
           EnumSet.range(DayOfWeek.SATURDAY, DayOfWeek.SUNDAY);
3      LocalDate result = (LocalDate) w;
4      do {
5        result = result.plusDays(1);
6      }
7      while (!weekend.contains(result.getDayOfWeek()));
8    return result;
9  };
10 LocalDate.now().with(NEXT_WEEKEND);
11 // 2018-11-17
```

Interfejsy autorskie

- Zdefiniujmy interfejs przypisujący kolor współrzędnym:

```
1  @FunctionalInterface
2  public interface PixelFunction {
3      Color apply(int x, int y);
4  }
```

- Utwórzmy funkcję tworzącą obrazek:

```
1  BufferedImage createImage (int width, int height,
       PixelFunction f) {
2      BufferedImage image = new BufferedImage(width,
           height,
3      BufferedImage.TYPE_INT_RGB);
4      for (int x = 0; x < width; x++)
5          for (int y = 0; y < height; y++) {
6              Color color = f.apply(x, y);
7              image.setRGB(x, y, color.getRGB());
8          }
9      return image;
10 }
```

- Zdefiniujmy co ma być na obrazku:

```
1  BufferedImage polishFlag = createImage(150, 100,
2  (x, y) -> y < 50 ? Color.WHITE : Color.RED);
```

Motywacja

- Na początku wyeliminowaliśmy nazwę klasy.
- Potem usunęliśmy całą strukturę klasy.
- Została nam pojedyncza metoda.
- Czy można posunąć się dalej?
- Czy możemy wyeliminować pisanie metody?
- Pewnie, możemy ją od kogoś pożyczyć.

Referencje do metod

- Zamiast używać wyrażeń lambda możemy bezpośrednio wskazać istniejącą metodę.
- Wyrażenie:

```
Arrays.sort(strings, (x, y) ->
    x.compareToIgnoreCase(y));
```

- Możemy zastąpić poprzez

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

- `String::compareToIgnoreCase` jest referencją do metody `compareToIgnoreCase` klasy `String`.

Typy referencji

- Istnieją trzy typy referencji
  1. `Klasa::metoda_instancji`
  2. `Klasa::metoda_statyczna`
  3. `Obiekt::metoda_instancji`

Klasa::metoda_instancji

- W wyrażeniu

```
1  (x, y) -> x.compareToIgnoreCase(y).
```

  - pierwszy parametr jest właścicielem metody,
  - drugi parametr to argument metody[3].

- Ponieważ znaczenie parametrów jest jednoznacznie określone możemy zastąpić wyrażenie poprzez

```
1  String::compareToIgnoreCase
```

[3] Potencjalne kolejne parametry też będą argumentami

Klasa::metoda_statyczna

- W wyrażeniu

```
1  x -> Objects.isNull(x)
```

  - parametr `x` jest argumentem metody statycznej[4].

- Ponieważ znaczenie parametrów jest jednoznacznie określone możemy zastąpić wyrażenie poprzez

```
1  Objects::isNull
```

[4] Potencjalne kolejne parametry też będą argumentami

Obiekt::metoda_instancji

- W wyrażeniu

```
1  x ->System.out.println(x)
```

  - wywołujemy metodę `println` dla konkretnego obiektu `System.out`,
  - parametr `x` jest argumentem metody dynamicznej[5].

- Ponieważ znaczenie parametrów jest jednoznacznie określone możemy zastąpić wyrażenie poprzez

```
1  System.out::println
```

[5] Potencjalne kolejne parametry też będą argumentami

Referencje do nadpisanych metod

- W przypadku istnienia kilku metod o tej samej nazwie kompilator decyduje, którą z nich wybrać.
- Decyzja zależy od kontekstu.
- Dla instancji `ArrayList<String> strings` wywołajmy następujący kod

```
1  strings.forEach(System.out::println);
```

- Kompilator domyśli się, że należy wywołać metodę `println(String)`.

Referencje do konstruktora

- Referencja do konstruktora przyjmuje formę `Klasa::new`.
- Konstruktor jest dobierany na podstawie parametrów.
- Wywołanie `Employee::new` utworzy nową instancję klasy `Employee` class.
- Referencja pozwala tworzyć nowe obiekty w locie, co jest wykorzystywane w strumieniowym przetwarzaniu danych.

Funkcje wyższego rzędu

Definicja

Funkcja wyższego rzędu przyjmuje jedną lub więcej funkcji jako argumenty lub zwraca funkcję.

- W Javie jest to funkcja, która przyjmuje jako argument interfejs funkcyjny lub zwraca interfejs funkcyjny.

Funkcja wyższego rzędu zwracająca funkcję

- Funkcja wyższego rzędu może zwracać funkcję (interfejs funkcyjny):

```
1  public static Comparator<String> compareInOrder(int order) {
2      return (x, y) -> order * x.compareTo(y);
3  }
```

- Funkcja pozwala nam ustalać porządek sortowania.

```
1  Arrays.sort(students, compareInOrder(-1));
```

Funkcja wyższego rzędu z funkcją jako argumentem

- Poniższa metoda odwraca sortowanie dowolnego komparatora klasy `String`

```
1  public static Comparator<String>
       reverse(Comparator<String> comp) {
2      return (x, y) -> comp.compare(y, x);
3  }
```

- Możemy ją wywołać z dowolnym argumentem będącym interfejsem funkcyjnym `Comparator<String>`.

```
1  reverse(String::compareToIgnoreCase)
```

- Zauważmy, że funkcja także zwraca taki interfejs. Pozwala to na tworzenie złożeń funkcji, wielokrotnie przetwarzających dane wejściowe.

Metoda comparing

- Interfejs `Comparator` posiada kilka użytecznych metod domyślnych.
- Metoda `comparing` pozwala określić klucz, czyli argument po którym będą porównywane dane.
- Argument określany jest przez wskazanie zwracającej go metody[6].

```
1  Arrays.sort(coins, Comparator.comparing(Coin::getName));
```

- Natomiast używając metody `thenComparing` możemy dodawać następne warunki porównania.

```
1  Arrays.sort(coins,
       Comparator.comparing(Coin::getYear).thenComparing(Coin::getName)
```

[6] enkapsulacja

Określanie sposobu porównywania

- Nie jesteśmy skazani na porównywanie kluczy w sposób domyślny.
- Możemy zdefiniować funkcję je porównującą.

```
1  Arrays.sort(coins, Comparator.comparing(Coin::getName,
2  (s, t) -> s.length() - t.length()));
```

- W danym przypadku łatwiej będzie się jednak posłużyć metodą porównującą liczby całkowite `comparingInt`

```
1  Arrays.sort(coins,
       Comparator.comparingInt(s->s.getName().length()))
```

Porównywanie z `null`

- Odwiecznym problemem analizy danych są braki w danych.
- W Javie brakujące obiekty zastępujemy poprzez `null`.
- Jak jednak potraktować je podczas porównywania?
- Dwie metody `nullsFirst` i `nullsLast` pozwalają nam sortować dane z brakami.

```
Arrays.sort(coins, comparing(Coin::getPreviousOwner,
nullsFirst(naturalOrder())));
```

- Aby ich użyć musimy określić komparator porządkujący pozostałe dane.
- Można się posłużyć komparatorami `naturalOrder` i `reverseOrder`, które można stosować do każdej klasy implementującej interfejs `Comparable`.

Bibliografia