Zaawansowane programowanie w Javie

Studia zaoczne – Wykład 2

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/

Wyrażenia lambda

Składnia wyrażeń lambda

- Wyrażenie lambda jest blokiem kodu, który można przekazać do późniejszego wykonania, raz lub kilka razy.
- We wcześniejszych przykładach widzieliśmy wiele sytuacji, gdzie taki blok kodu by się przydał:
  - do przekazania metody porównującej do `Arrays.sort`,
  - do uruchomienia zadania w oddzielnym wątku,
  - do określenia akcji, jaka powinna zostać wykonana po kliknięciu przycisku.
- Java jest jednak językiem obiektowym, w którym (praktycznie) wszystko jest obiektem. W Javie nie ma typów funkcyjnych.
- Zamiast tego funkcje mają postać obiektów – instancji klas implementujących określony interfejs.
- Wyrażenia lambda udostępniają wygodną składnię do tworzenia takich instancji.

Wyrażenia lambda

Składnia wyrażeń lambda

- Ponownie zajmijmy się przykładem sortowania z użyciem interfejsu `Comparator`.
- Przekazujemy kod sprawdzający, czy jeden ciąg znaków jest krótszy od innego. Obliczamy: `pierwszy.length() - drugi.length()`.
- Czym są pierwszy i drugi? Są to ciągi znaków. Java jest językiem o silnym typowaniu i musimy to również określić:

```
(String pierwszy, String drugi) ->
    pierwszy.length() - drugi.length()
```

- Powyższe wyrażenie jest pierwszym przykładem wyrażenia lambda. Takie wyrażenie jest po prostu blokiem kodu z opisem zmiennych, które muszą zostać do niego przekazane.

Wyrażenia lambda

Składnia wyrażeń lambda

- Jeżeli obliczeń wyrażenia lambda nie da się zapisać w jednym wyrażeniu, należy zapisać je w taki sam sposób jak przy tworzeniu metody – wewnątrz nawiasów klamrowych z jawnie zapisaną instrukcją **return**. Na przykład:

```
(String pierwszy, String drugi) -> {
    int różnica =
        pierwszy.length() < drugi.length();
    if (różnica < 0)
        return -1;
    else if (różnica > 0)
        return 1;
    else
        return 0;
}
```

Wyrażenia lambda

Składnia wyrażeń lambda

- Jeżeli wyrażenie lambda nie ma parametrów, należy umieścić puste nawiasy, jak w przypadku metody bez parametrów:

```
Runnable task = () -> {
    for (int i = 0; i < 1000; i++) {
        doWork();
    }
}
```

- Jeżeli typy parametrów wyrażenia lambda mogą być jednoznacznie ustalone, można je pominąć. Na przykład:

```
Comparator<String> comp = (pierwszy, drugi) ->
    pierwszy.length() - drugi.length();
```

Wyrażenia lambda

Składnia wyrażeń lambda

- Jeżeli metoda ma jeden parametr domyślnego typu, można nawet pominąć nawiasy:

```
EventHandler<ActionEvent> listener = event ->
    System.out.println("Oh, nie!");
// Zamiast (event) -> lub (ActionEvent event) ->
```

- Nigdy nie określa się typu wartości zwracanej przez wyrażenie lambda. Kompilator jednak ustala go na podstawie treści kodu i sprawdza, czy zgadza się on z oczekiwanym typem. Wyrażenie

```
(String pierwszy, String drugi) ->
    pierwszy.length() - drugi.length()
```

może być użyte w kontekście, w którym oczekiwany jest wynik typu **`int`** (lub typu kompatybilnego, jak `Integer`, **`long`** czy **`double`**).

Wyrażenia lambda

Przykłady

- Program `r03/r03_04/LambdaDemo.java`
- Program `r03/r03_04/ButtonDemo.java`

Wyrażenia lambda

Interfejsy funkcyjne

- Jak już widzieliśmy, w języku Java istnieje wiele interfejsów określających działania, takich jak `Runnable` czy `Comparator`. Wyrażenia lambda są kompatybilne z tymi interfejsami.
- Można umieścić wyrażenie lambda wszędzie tam, gdzie oczekiwany jest obiekt implementujący jedną metodę abstrakcyjną. Taki interfejs nazywany jest interfejsem funkcyjnym.
- Aby zademonstrować konwersję do interfejsu funkcjonalnego, przyjrzyjmy się metodzie `Arrays.sort`.
- Jej drugi parametr wymaga instancji interfejsu `Comparator` zawierającego jedną metodę. Wstawmy tam wyrażenie lambda:

```
Arrays.sort(słowa, (pierwszy, drugi) ->
    pierwszy.length() - drugi.length());
```

Wyrażenia lambda

Interfejsy funkcyjne

- W tle drugi parametr metody `Arrays.sort` zamieniany jest na obiekt pewnej klasy implementującej `Comparator<String>`.
- Wywołanie metody `compare` na tym obiekcie powoduje wykonanie treści wyrażenia lambda.
- Zarządzanie takimi obiektami i klasami jest w pełni zależne od implementacji i dobrze zoptymalizowane.
- W większości języków programowania obsługujących literały funkcyjne można deklarować typy funkcyjne takie jak

```
(String, String) -> int
```

- Powoduje to umieszczenie w tak zadeklarowanej zmiennej funkcji oraz jej wykonanie.

Wyrażenia lambda

Interfejsy funkcyjne

- W języku Java wyrażenie lambda można wykorzystać tylko w jeden sposób.
- Należy umieścić je w zmiennej, której typem jest interfejs funkcyjny, tak by została zamieniona w instancję tego interfejsu.
- Nie można przypisać wyrażenia lambda do zmiennej typu `Object`, wspólnego typu nadrzędnego dla wszystkich klas języka Java.
- Wynika to z tego, że `Object` to klasa, a nie interfejs funkcyjny.

Interfejsy funkcyjne

- Biblioteka standardowa udostępnia dużą liczbę interfejsów funkcyjnych. Jednym z nich jest interfejs `Predicate`:

```
@FunctionalInterface
public interface Predicate<T>
{
    boolean test(T t); // metoda abstrakcyjna
    // Pozostałe metody instancyjne
    // Dodatkowe metody statyczne
    // Dodatkowe metody domyślne
}
```

Wyrażenia lambda

Interfejsy funkcyjne

- Klasa `ArrayList` ma metodę `removeIf`, której parametrem jest `Predicate`.
- Jest to zaprojektowane z myślą o zastosowaniu wyrażenia lambda.
- Przykładowo, poniższe wyrażenie usuwa wszystkie wartości `null` z tablicy `ArrayList`:

```
list.removeIf(e -> e == null);
```

Wyrażenia lambda

Referencje do metod i konstruktora

- Zdarza się, że jest już metoda wykonująca działanie, które chcielibyśmy wykorzystać w innym miejscu kodu.
- Istnieje specjalna składnia dla referencji do metod, która jest nawet krótsza niż wyrażenie lambda wywołujące metodę.
- Podobny skrót stosuje się w przypadku konstruktorów.

Wyrażenia lambda

Referencje do metod

- Załóżmy, że chcemy sortować ciągi znaków bez zwracania uwagi na wielkość liter. Powinniśmy wywołać

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

- Zamiast tego można też przekazać takie wyrażenie:

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

- Wyrażenie `String::compareToIgnoreCase` jest referencją do metody, która stanowi odpowiednik poniższego wyrażenia lambda

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

Wyrażenia lambda

Referencje do metod

- W kolejnym przykładzie rozważmy metodę `isNull` z klasy `Objects`. Wywołanie

```
Objects.isNull(x)
```

  zwraca wartość wyrażenia `x == null`.

- Nie widać, aby warto było w tym przypadku tworzyć metodę, ale zostało to zaprojektowane w taki sposób, by przekazywać tu metodę. Wywołanie

```
list.removeIf(Objects::isNull);
```

  usuwa wszystkie wartości `null` z listy.

Wyrażenia lambda

Referencje do metod

- Jako inny przykład załóżmy, że chcemy wyświetlić wszystkie elementy listy.
- Klasa `ArrayList` ma metodę `forEach` wykonującą funkcję na każdym jej elemencie.
- Można by było użyć wywołania

```
list.forEach(x -> System.out.println(x));
```

- Przyjemniej jednak byłoby, gdybyśmy mogli przekazać po prostu metodę `println` do metody `forEach`.
- Można to zrobić w następujący sposób:

```
list.forEach(System.out::println);
```

Wyrażenia lambda

Referencje do metod

- Jak można było zobaczyć w tych przykładach, operator `::` oddziela nazwę metody od nazwy klasy lub obiektu. Istnieją trzy rodzaje użycia tego operatora:
  1. `Klasa::metodaInstancyjna`
  2. `Klasa::metodaStatyczna`
  3. `obiekt::metodaInstancyjna`
- W pierwszym przypadku pierwszy argument staje się odbiorcą metody i wszystkie inne argumenty są przekazywane do metody.
- Przykładowo, wyrażenie

```
String::compareToIgnoreCase
```

jest równoważne z wyrażeniem lambda

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

Wyrażenia lambda

Referencje do metod

- W drugim przypadku wszystkie argumenty są przekazywane do metody statycznej.
- Wyrażenie `Objects::isNull` jest równoważne z wyrażeniem

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

- W trzecim przypadku metoda jest wywoływana na danym obiekcie i argumenty są przekazywane do metody instancji.
- W tej sytuacji wyrażenie

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

jest równoważne z wyrażeniem lambda

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

Wyrażenia lambda

Referencje do metod

- Gdy istnieje wiele przeładowanych metod z tą samą nazwą, kompilator na podstawie kontekstu będzie próbował ustalić, którą z nich chcemy wywołać.
- Przykładowo, istnieje wiele wersji metody `println`. Po przekazaniu do metody `forEach` zmiennej typu `ArrayList<String>` zostanie wybrana metoda `println(String)`.
- W referencji do metody można wykorzystać referencję **`this`**. Przykładowo, wyrażenie `this::equals` jest równoważne z wyrażeniem lambda `x -> this.equals(x)`.

Wyrażenia lambda

Referencje do metod

- W klasie wewnętrznej można wykorzystać referencję **`this`** do klasy zewnętrznej poprzez `KlasaZewnętrzna.this::metoda`.
- Można wykorzystać też słowo kluczowe **`super`**.

Przykład (Referencje do metod)

- Program `r03/r03_05/MethodReferenceDemo.java`

Referencje do konstruktora

- Referencje do konstruktora są odpowiednikami referencji do metod, tyle że w ich przypadku nazwą metody jest **`new`**.
- Na przykład `Employee::new` jest referencją do konstruktora klasy `Employee`.
- Jeśli klasa ma więcej niż jeden konstruktor, od kontekstu zależy, który z nich zostanie wywołany.
- Poniżej znajduje się przykład wykorzystania takiej referencji do konstruktora. Załóżmy, że mamy listę nazwisk:
  `List<String> names = ...;`
- Potrzebujemy listy pracowników, jednej dla każdego nazwiska. Jak zobaczymy, można wykorzystać strumienie, by wykonać to samo bez pętli: zamienić listę na strumień i wywołać metodę `map`.

Referencje do konstruktora

- Powoduje to wykonanie funkcji i zbiera wszystkie rezultaty:

```
Stream<Employee> stream =
    names.stream().map(Employee::new);
```

- Ponieważ `names.stream()` zawiera obiekty klasy `String`, kompilator wie, że `Employee::new` odwołuje się do konstruktora `Employee(String)`.
- Można tworzyć też referencje do konstruktora z typami tablicowymi.
- Przykładowo, `int[]::new` jest referencją do konstruktora z jednym argumentem: długością tablicy. Jest to odpowiednik wyrażenia lambda `n -> new int[n]`.

Wyrażenia lambda

Referencje do konstruktora

- Referencje do konstruktora z typami tablicowymi pozwalają na ominięcie ograniczenia języka Java polegającego na tym, że nie jest możliwe skonstruowanie tablicy elementów typu generycznego.
- Przykładowo, próba utworzenia tablicy obiektów typu `java.util.Map.Entry`:

```
Entry<String, Integer>[] entries =
    new Entry<String, Integer>[100];
```

  skutkuje błędem kompilacji `generic array creation`.
- Zauważmy jednak, że typ `Entry<String, Integer>[]` jest całkowicie poprawny.
- Sposób inicjalizacji zmiennej takiego typu pokazany jest w programie `EntryArrayDemo.java`.

Referencje do konstruktora

- Z tego powodu metody takie jak `Stream.toArray` zwracają tablicę elementów typu `Object`, a nie tablicę elementów takiego samego typu jak obiekty znajdujące się w strumieniu:

```
Object[] employees = stream.toArray();
```

- Nie jest to jednak satysfakcjonujące. Użytkownik potrzebuje tablicy pracowników, a nie obiektów.
- Aby rozwiązać ten problem, inna wersja `toArray` akceptuje referencje do konstruktora:

```
Employee[] employees =
    stream.toArray(Employee[]::new);
```

Wyrażenia lambda

Referencje do konstruktora

- Metoda `toArray` wywołuje ten konstruktor, by uzyskać tablicę odpowiedniego typu. Następnie wypełnia ją i zwraca dane w tablicy.

Przykład (Referencje do konstruktora)

- Program `r03/r03_05/ConstructorReferenceDemo.java`

Przetwarzanie wyrażeń lambda

Implementacja odroczonego wykonania

- Wiemy już, jak tworzyć wyrażenia lambda i przekazać je do metody, która oczekuje interfejsu funkcyjnego.
- Teraz zobaczymy, jak napisać metody, które mogą korzystać z wyrażeń lambda.
- Celem korzystania z wyrażeń lambda jest odroczone wykonanie.
- Gdybyśmy chcieli wykonać jakieś polecenia w danym miejscu kodu bezzwłocznie, zrobilibyśmy to bez opakowywania go w wyrażenie lambda.

Przetwarzanie wyrażeń lambda

Implementacja odroczonego wykonania

- Istnieje wiele powodów opóźnienia wykonania kodu – są to:
  - wykonanie kodu w oddzielnym wątku,
  - wielokrotne wykonanie kodu,
  - wykonanie kodu we właściwym miejscu algorytmu (na przykład operacja porównania przy sortowaniu),
  - wykonanie kodu w reakcji na zdarzenie (kliknięcie przycisku, odebranie danych itd.),
  - wykonanie kodu tylko w razie potrzeby.

Przetwarzanie wyrażeń lambda

Implementacja odroczonego wykonania

- Popatrzmy na prosty przykład. Załóżmy, że chcemy powtórzyć działanie n razy. Działanie i licznik są przekazywane do metody `repeat`:

```
repeat(10, () -> System.out.println("Witaj, Świecie!"));
```

- Aby wykorzystać wyrażenie lambda, musimy wybrać (lub w rzadkich przypadkach utworzyć) interfejs funkcyjny. W takim przypadku możemy użyć na przykład interfejsu `Runnable`:

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++)
        action.run();
}
```

- Zauważmy, że kod z wyrażenia lambda wykonywany jest po wywołaniu `action.run()`.

Przetwarzanie wyrażeń lambda

Implementacja odroczonego wykonania

- Skomplikujemy teraz ten przykład. Chcemy do działania przekazać informację o numerze iteracji, w której jest wywoływane.
- W takim przypadku musimy wybrać interfejs funkcyjny, który ma metodę z parametrem **`int`** i zwraca **`void`**.
- Zamiast tworzenia własnego polecane jest wykorzystanie jednego ze standardowych interfejsów.
- Standardowym interfejsem do przetwarzania wartości typu **`int`** jest interfejs `IntConsumer`:

```
public interface IntConsumer {
    void accept(int value);
}
```

Przetwarzanie wyrażeń lambda

Implementacja odroczonego wykonania

- Oto ulepszona wersja metody `repeat`:

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

- A wywołuje się ją w taki sposób:

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

Przykład

- Program `r03/r03_06/RepeatDemo.java`