Wyrazenia lambda i strumienie w Javie 8

Metody Programowania

Krzysztof Pawłowski

kpawlowski@pjwstk.edu.pl

25 listopada 2017

Co nowego w Javie 8?

- Lambda expressions
- Functional Interfaces
- Method references
- Default Methods
- Collections - Streams API

Co nowego w Javie 8?

- Type Annotations
- new Date-Time API
- Scripting - Nashorn Javascript Engine
- Parallel Array Sorting
- Standard Encoding and Decoding Base64

Czym jest wyrażenie lambda?

- Sparametryzowany blok kodu, który może być wykonany później w kodzie.
- Tak jak metoda, składa się z listy formalnych parametrów oraz treści.

Przykłady

```
() -> 1
 x -> x * x
(x, y) -> (x + y)
(String s1, String s2) -> (s1.equals(s2))
(String s) -> System.out.println(s)
```

Zalety wyrażeń lambda

- Wsparcie dla kolekcji - łatwiejsze iterowanie, filtrowanie, wyciąganie danych.
- Jasne i zwięzłe definiowanie interfejsu funkcyjnego przy użyciu wyrażenia.
- Zwiększenie wydajności w środowiskach wielordzeniowych.

Iterowanie zewnętrzne vs wewnętrzne

Iterowanie zewnętrzne

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

for (int number : numbers) {
  System.out.println(number);
}
```

Iterowanie wewnętrzne

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

numbers.forEach(value -> System.out.println(value));
```

Interfejs funkcyjny vs wyrażenie lambda

Interfejs funkcyjny

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}

public class ListenerTest {
  public static void main(String[] args) {
    JButton testButton = new JButton("Test␣Button");
    testButton.addActionListener(new ActionListener() {
      @Override public void actionPerformed(ActionEvent e)
        System.out.println("Click␣Detected␣by␣Anon␣Class");
      }
   });
}
```

Interfejs funkcyjny vs wyrażenie lambda

Wyrażenie lambda

```
public class ListenerTest {
  public static void main(String[] args) {
    JButton testButton = new JButton("Test␣Button");
    testButton.addActionListener(e ->
          System.out.println("Click␣Detected␣by␣Lambda"));
  }
}
```

Wbudowane generyczne interfejsy funkcyjne

Jakiego typu mogą być wyrażenia lambda?

- Interfejsy z pakietu java.util.function
  - Predicate<T>
  - Consumer<T>
  - Function<T, R>
  - Supplier<T>
  - UnaryOperator<T>
  - BinaryOperator<T>
  - ...
  - więcej pod adresem: `http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`

Interfejs Predicate<T>

- Reprezentuje predykat (funkcję, której wartości są typu logicznego) o jednym argumencie.

```
public interface Predicate<T> {
  Predicate<T>                and(Predicate<? super T> other);
  static <T> Predicate<T>     isEqual(Object targetRef);
  Predicate<T>                negate();
  Predicate<T>                or(Predicate<? super T> other);
  boolean                     test(T t);
}
```

Przykład

```
Predicate<String> stringsLongerThan10 =
                  (String s) -> s.length() > 10;
boolean isLongerThan10 =
        stringsLongerThan10.test(``Ala ma kota.'')
```

Interfejs Consumer<T>

- Reprezentuje operację, która przyjmuje jeden argument i nie zwraca żadnego wyniku.
- W przeciwieństwie do większości interfejsów funkcyjnych, interfejs *Consumer* operuje poprzez efekty uboczne (ang. *side effects*).

```
public interface Consumer<T> {
  void          accept(T t);
  Consumer<T>   andThen(Consumer<? super T> after);
}
```

Przykład

```
Consumer<String> printToStdOut =
                 (String s) -> System.out.print(s);
printToStdOut.accept(``Hello World!'');
```

Interfejs Function<T, R>

- Reprezentuje funkcję jednoargumentową, która daje wynik typu *R*.

```
public interface Function<T, R> {
  Function<T,V>  andThen(Function<? super R,? extends V>
                         after);
  R              apply(T t);
  Function<V,R>  compose(Function<? super V,? extends T>
                         before);

  static Function<T,T>  identity();
}
```

Przykład

```
Function<String, Integer> stringLength =
                  (String s) -> s.length();
int strLen = stringLength.apply(``'Ala␣ma␣kota.');
```

Interfejs Supplier<T>

- Reprezentuje funkcję bezargumentową, która "dostarcza" pewien obiekt.

```
public interface Supplier<T> {
  T   get();
}
```

Przykład

```
Supplier<String> stringSupplier =
                  () -> new String("Ala ma kota");
String s = stringSupplier.get();
```

Interfejs UnaryOperator<T>

- Reprezentuje operację na pojedynczym operandzie, której wynik jest tego samego typu jak operand.
- Jest szczególnym przypadkiem interfejsu *Function<T>*

```
public interface UnaryOperator<T> extends Function<T,T> {
  // zestaw metod z Function<T,T>
  static <T> UnaryOperator<T> identity();
}
```

Przykład

```
UnaryOperator<Integer> multipleBy10 =
                  (Integer i) -> i * 10;
int multipliedByTen = multipleBy10.apply(10)
```

Wyrażenia lambda w praktyce

Kod dostępny pod adresem:

- `https://github.com/krzysztof-pawlowski/Teaching-JINT-StreamsExamples`

Przekazywanie metod (ang. method references)

- Za pomocą wyrażeń lambda tworzyliśmy anonimowe metody.
- Jako parametry funkcji możemy przekazywać nazwane metody:
  - statyczne – NazwaKlasy::nazwaStatycznejMetody,
  - instancji klas – nazwaObiektu::nazwaMetody,
  - oraz konstruktor – NazwaKlasy::new.

Przykład

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

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

Metody domyślne (ang. default methods)

- Wprowadzone, aby do Collections API moc wprowadzic Streams API
- Pozwalają rozszerzać interfejsy o domyślną implementację metod z zachowaniem wstecznej kompatybiności.
- Mogą być dodane do każdego interfejsu - wtedy klasa implementująca ten interfejs, jeśli nie nadpisze tej metody, odziedziczy domyślną implementację tej metody

Metody domyślne - problem diamentu

```
interface Foo {
  default void talk() {
    System.out.println("Foo!");
  }
}

interface Bar {
  default void talk() {
    System.out.println("Bar!");
  }
}

public class FooBar implements Foo, Bar {
  //???
}
```

Czym są strumienie w Javie 8?

- Klasy z pakietu java.util.stream.
- Wspierają operacje funkcyjne na sekwencjach elementów (strumieniach):
  - redukcje (ang. *reduce*) – suma, średnia, min itd.,
  - odwzorowania (ang. *map*),
  - filtrowanie.
- Wspierają zrównoleglanie operacji.
- Są zitegrowane z Collections API.

Kolekcje vs strumienie

Strumienie w przeciwieństwie do kolekcji:

- nie przechowują elementów,
- mają naturę funkcyjną (źródłowa kolekcja nie jest modyfikowana),
- stosują leniwe wyliczanie,
- mogą być nieograniczone,
- mogą być odwiedzone tylko raz w ich cyklu życia (jak iterator).

Tworzenie strumieni

Strumienie możemy tworzyć na wiele sposobów, np.:

- metody stream() i parallelStream() klas reprezentujących kolekcje (np. LinkedList<T>),
- Arrays.stream(Object[]),
- Stream.of(Object[]),
- IntStream.range(int, int),
- Random.ints().

Operacje na strumieniach

Operacje na strumieniach dzielimy na

- pośrednie (ang. *intermediate*):
  - jako wynik dają nowy strumień,
  - są wyliczane leniwie,
  - mogą przechowywać stan (np. distinct, sorted) bądź nie przechowywac stanu (np. filter, map),
- końcowe (ang. *terminal*):
  - przechodzą przez strumień w celu obliczenia wyniku bądź wykonania efektu ubocznego,
  - kończą sekwencję operacji strumieniowych (ang. *stream pipeline*).

Zrównoleglanie strumieni

- Jak zrównoleglić obliczanie sekwencji operacji na strumieniach?
  - Przy tworzeniu strumienia użyć metody parallelStream() zamiast stream(),
  - lub dodać operację parallel() po utworzeniu strumienia.
- Nie wszystkie operacje na strumieniach da się zrównoleglić.
- Aby było to możliwe, funkcje przekazywane jako argumenty operacji na strumieniach nie powinny zmieniać stanu (ang. *stateless*) oraz modyfikować wejściowej struktury danych (ang. *non-interfering*).

Efekty uboczne

- Powinny być unikane (mogą powodować zmianę stanu, problemy z wątkami).
- Często można je zastąpić np. redukcją.
- Nieszkodliwy efekt uboczny: println.

```
ArrayList<String> results = new ArrayList<>();
stream.filter(s -> s.length() > 10)
      .forEach(s -> results.add(s));
```

można zastąpić przez:

```
List<String> results = stream.filter(s -> s.length() > 10)
      .collect(Collectors.toList());
```

Porządek elementów w strumieniach

- Zależny od źródłowej struktury danych.
- Operacja sorted().
- Operacja unsorted().
- Pominięcie kolejności elementów może zwiększyć wydajność.

Operacje redukcji

- Redukcja to operacja, która dla ciągu elementów daje skalarny wynik (np. suma), bądź akumuluje elementy w nowej liście.
- Ogólne operacje redukcji:
  - reduce()
  - collect()
- Wyspecjalizowane operacje redukcji:
  - sum()
  - max()
  - count()
  - ...

Przykład

```
List<Integer> = Arrays.asList(1, 2, 3, 4);
int sum = numbers.parallelStream()
                 .reduce(0, Integer::sum);
```

Operacja reduce()

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner);
```

Przykład

```
int sumOfWidths = pictures.stream()
            .reduce(0,
                    (sum, b) -> sum + b.getWidth(),
                    Integer::sum);
```

Operacja collect()

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Przykład

```
List<String> strings = stream.map(Object::toString)
                .collect(ArrayList::new,
                         ArrayList::add,
                         ArrayList::addAll);

List<String> strings = stream.map(Object::toString)
                .collect(Collectors.toList());
```

Klasa Collectors - przykłady

```
Set<String> set = people.stream().map(Person::getName)
     .collect(Collectors.toCollection(TreeSet::new));

String joined = things.stream()
     .map(Object::toString)
     .collect(Collectors.joining(",␣"));

int total = employees.stream()
     .collect(Collectors.summingInt(Employee::getSalary)));

Map<Department, List<Employee>> byDept
     = employees.stream()
     .collect(Collectors.groupingBy(Employee::getDept));
```

Źródło: http://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

Strumienie w praktyce

Kod dostępny pod adresem:

- `https://github.com/krzysztof-pawlowski/Teaching-MPR/tree/master/lecture/`