JDK 8

WPROWADZENIE DO WYBRANYCH ZAGADNIEŃ

by Szymon Stępniak / @wololock / crafted with reveal.js

ABOUT ME

Software Developer @ Ideazone (Torun, Poland)

Java/Groovy, Grails contributor, TDD-holic

@wololock

AGENDA

1. Co nowego w Javie?
2. Istotne zmiany w interfejsach
3. Wyrażenia Lamdba jako syntactic sugar
4. java.util.stream.Stream

WYBRANE NEW FEATURES

- 101 Generalized Target-Type Inference
- 103 Parallel Array Sorting
- 104 Annotations on Java Types
- 107 Bulk Data Operations for Collections
- 109 Enhance Core Libraries with Lambda
- 117 Remove the Annotation-Processing Tool (apt)
- 120 Repeating Annotations
- 122 Remove the Permanent Generation
- 126 Lambda Expressions & Virtual Extension Methods
- 135 Base64 Encoding & Decoding
- 150 Date & Time API
- 153 Launch JavaFX Applications
- 155 Concurrency Updates
- 160 Lambda-Form Representation for Method Handles
- 161 Compact Profiles
- 162 Prepare for Modularization
- 174 Nashorn JavaScript Engine
- 184 HTTP URL Permissions
- 185 JAXP 1.5: Restrict Fetching of External Resources

HTTP://OPENJDK.JAVA.NET/PROJECTS/JDK8/FEATURES

JAVA 8 POWSTAŁA Z UDZIAŁEM SPOŁECZNOŚCI

https://java.net/projects/adoptopenjdk/

2. ISTOTNE ZMIANY W INTERFEJSACH

METODY STATYCZNE W INTERFEJSACH

```
public interface InterfaceWithStaticMethod {

    public static String itWorks(int number) {
        return String.format("Works for %d", number);
    }
}
```

Zastosowanie: implementacja metod statycznych w interfejsie pozwoli zrezygnować z tzw. klas narzędziowych (przykład: java.util.Collections dla interfejsu java.util.Collection<E>

METODY DOMYŚLNE W INTERFEJSACH

```
// (stripped down version of the real Iterable.java)
public interface Iterable<T> {

    Iterator<T> iterator(); // As in prior versions

    // example of a new JDK 8 default method
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

DZIEDZICZENIE METOD DOMYŚLNYCH

```
public interface A {
    default int foo(Integer n) {
        return n + n;
    }
}

public interface B {
    default int foo(Integer n) {
        return n + 2;
    }
}

public class C implements A,B {
    @Override
    public int foo(Integer n) {
        return A.super.foo(n);
    }
}
```

INTERFEJSY FUNKCYJNE

```
@FunctionalInterface
public interface Command<T,V> {
    V execute(T parameter);

    default void itMayContainsDefaultMethod() {
        //...rocket science
    }

    default Long andAnotherDefaultMethodAndStillBeAFunctionalInterface()
        return 20L;
    }
}
```

@FunctionalInterface - opcjonalnie

Każdy interfejs posiadający dokładnie jedną metodę abstrakcyjną jest interfejsem funkcyjnym

A ZATEM INFTERFEJSAMI FUNKCYJNYMI SĄ M.IN.

- java.lang.Runnable
- java.awt.event.ActionListener
- java.lang.Comparable<T>

JAVA.UTIL.FUNCTION

FUNCTION<T, R>

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
}
```

dokonuje przekształcenia wartości typu T na wartość typu R

PREDICATE<T>

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);
}
```

test logiczny dla wartości typu T

CONSUMER<T>

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

wykonuje bezwynikową operację na wartości typu T

SUPPLIER<T>

```
@FunctionalInterface
public interface Supplier<T> {

    T get();
}
```

dostarcza bezwarunkową wartość typu T

BIFUNCTION<T, U, R>

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);
}
```

analogicznie do Function, z tą różnicą że obsługiwane są dwa argumenty funkcji

BINARYOPERATOR<T>

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
}
```

zwyczajne działanie binarne dla wartości typu T

PRZED JAVA 8

Podobny zbiór interfejsów "funkcyjnych" udostępnia do dawna biblioteka Guava

http://code.google.com/p/guava-libraries/wiki/FunctionalExplained

ZOBACZMY JAK TO WYGLĄDA W PRAKTYCE

3. WYRAŻENIA LAMBDA

Wyrażenie lambda - "syntactic sugar" dla korzystania z interfejsów funkcyjnych

() -> {}

(Integer x, Integer y) -> { return x + y; }

(Integer x, Integer y) -> { x + y }

(x,y) -> x + y

PORÓWNAJMY

```
(x,y) -> x + y
```

Z

```
new Function<Integer,Integer> {
    @Override
    public Integer apply(Integer x, Integer y) {
        return x + y;
    }
}
```

OCZYWISTE SKOJARZENIA Z DOMKNIĘCIAMI

```
//Groovy example:
def sqrt = { x -> x * x }

def marshall = { Payment payment ->
    [
        id: payment.id,
        amount: payment.amount,
        attempts: payment.history.findAll { attempt -> attempt.succeed()
    ]
}
```

Niemniej

LAMBDA != CLOSURE

Lamdba jako obiekt anonimowy ma dostęp do: pól obiektu w ramach którego jest zdefiniowana, pól statycznych oraz zmiennych finalnych w zewnętrznym lokalnym zakresie

Domknięcie natomiast tworzy kopię środowiska w momencie deklaracji, pozwalając na modyfikacje wszystkich widocznych zmiennych w ramach powstałej kopii

ALE TO NIE WSZYSTKO

```
Function<Integer, String> toString = x -> x.toString();
```

można zastąpić referencją do metody:

```
Function<Integer, String> toString = Object::toString;
```

Przykład referencji do metody statycznej

Integer::valueOf

Przykład referencji do metody klasy

Object::toString

Przykład referencji do metody obiektu

x::toString

Przykład referencji do konstruktora

Object::new

ZOBACZMY TERAZ JAK WYRAŻENIA LAMBDA UPROSZCZĄ NASZ PRZYKŁAD WYKORZYSTANIA FUNKCJI

4. STREAM API

java.util.stream.Stream

Nie mylimy z I/O Stream

Myślimy o nim jak o jednorazowym leniwym iteratorze

```
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);
        Stream<Integer> streamOfNumbers = numbers.stream();
    }
}
```

Domyślny Stream jest sekwencyjny. Jeśli możemy go jednak zrównoleglić:

stream.parallel()

RODZAJE OPERACJI WYKONYWANYCH NA STRUMIENIU

Wyróżniamy dwa rodzaje operacji dla java.util.stream.Stream:

intermediate (tzw. pośrednia) - nie zamykają strumienia i zwracają nowy strumień zawierający transformację wynikającą ze wskazanej operacji

terminal (tzw. kończąca) - operacja, która musi być wykonana jako ostatnia, uruchamiająca zarejestrowane wcześniej transformacje i zwracająca oczekiwany wynik. Po jej zakończeniu, strumień ginie.

PRZYKŁADY OPERACJI POŚREDNICH

filter(Predicate<? super T> predicate)

ogranicza zawartość strumienia do obiektów spełniających predykat

map(Function<? super T, ? extends R> mapper)

przekształca każdy element strumienia za pomocą przekazanej funkcji

flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

przekształca każdy element zagnieżdżonych strumieni i zwraca nowy strumień o płaskiej strukturze

distinct()

ogranicza zawartość strumienia do unikalnych obiektów, stosując Object.equals(Object)

sorted() oraz sorted(Comparator<? super T> comparator)

przekształca strumień do postaci posortowanej

peek(Consumer<? super T> action)

wykonuje dodatkową akcję z każdym elementem strumienia, bez przekształcania jego postaci

limit(long maxSize)

ogranicza strumień do podanego rozmiaru

PRZYKŁADY OPERACJI KOŃCZĄCYCH

forEach(Consumer<? super T> action);

wykonuje akcję z każdym elementem strumienia, zamykając go jednocześnie

reduce(T identity, BinaryOperator<T> accumulator)

rozpoczynając z wartością początkową **identity** wykonuje operację binarną dla każdego elementu strumienia oraz wartości wcześniej obliczonej

collect(Collector<? super T, A, R> collector)

"zbiera" wszystkie elementy pozostające w strumieniu i zwraca je w postaci determinowanej przez podany Collector

min(Comparator<? super T> comparator)

zwraca najmniejszą wartość z perspektywy wskazanego comparatora

max(Comparator<? super T> comparator)

analogicznie jak wyżej, z tą różnicą że zwracana jest największa wartość

boolean anyMatch(Predicate<? super T> predicate)

sprawdza, czy w strumieniu znajduje się co najmniej jeden obiekt spełniający predykat

boolean allMatch(Predicate<? super T> predicate)

sprawdza, czy wszystkie znajdujące się w strumieniu obiekty spełniają zadany predykat

boolean noneMatch(Predicate<? super T> predicate)

zaprzeczenie dla operacji allMatch

Optional<T> findFirst()

zwraca pierwszy element strumienia

Optional<T> findAny()

zwraca dowolny element strumienia

```
public class StreamExample {

    public static void main(String[] args) {
        List<String> names = Arrays.asList("John","Joe","James","Phillip'

        List <Integer> result = names.stream()
            .filter(name -> name.startsWith("J"))
            .map(String::length)
            .collect(Collectors.toList());
    }
}
```

NALEŻY PAMIĘTAĆ O TYM, ŻE:

- operacje pośrednie na strumieniu łączone są tzw. łańcuch wywołań
- transformacje strumienia nie modyfikują wejściowej kolekcji, z której został utworzony
- dopóki nie wywołamy operacji kończącej, żadna transformacja nie zostanie zastosowana
- próba ponownego wywołania dowolnej operacji kończącej kończy się wyjątkiem
- Collection.parallelStream() utworzy zrównoleglony strumień (ilość wątków = ilości procesorów)

ZOBACZMY STRUMIENIE W AKCJI

5. CO JESZCZE?

java.util.Optional<T>

bardziej wysublimowane podejście do ochrony przed NPE

```
public class StreamExample {

    public static void main(String[] args) {
        Optional<String> optionalName = Optional.of("Lorem Ipsum");

        assert optionalName.isPresent()
        assert optionalName.get().equals("Lorem ipsum")

        Optional<String> noName = Optional.of(null);
        assert optional.orElse("qweasdzxc").equals("qweasdzxc");

        //Optional.ifPresent(Consumer<T> consumer)
    }
}
```

Time API - java.time

http://docs.oracle.com/javase/tutorial/datetime/

- API wzorowane na popularnej bibliotece Joda-Time
- czytelne i proste w użyciu API
- ułatwiające manipulowanie czasem (np. przesunięcia z uwzględnieniem stref czasowych, tworzenie dat na podstawie przesunięć o x dni, tygodni etc.)
- praktycznie w całości **immutable**

NA TYM LISTA NOWOŚCI SIĘ NIE KOŃCZY

- Nashorn JavaScript Engine
- Java Compact Profiles
- nowe metody w standardowym API Java (np. String.join(delimiter, strings...)
- nowości w Concurrency API (np. CompletableFuture<T>)
- nowości w IO/NIO API (np. wykorzystanie lambda do czytania z wejścia)
- Base64 Encoding & Decoding
- Type Annotations
- .....

ALE O TYM MUSICIE PRZEKONAĆ SIĘ SAMI! :-)

GDZIE WARTO ZAJRZEĆ?

- https://github.com/AdoptOpenJDK/lambda-tutorial
- http://winterbe.com/posts/2014/03/16/java-8-tutorial/
- http://www.techempower.com/blog/2013/03/26/everything-about-java-8/
- http://zeroturnaround.com/rebellabs/java-8-revealed-lambdas-default-methods-and-bulk-data-operations/

DZIĘKUJĘ ZA UWAGĘ!