Notatki - Interfejsy Funkcyjne i Method Reference

Spis treści

Interfejs funkcyjny

Czym jest interfejs funkcyjny?

Wraz z Java 8 zostało wprowadzone pojęcie (i za razem mechanizm) interfejsu funkcyjnego. Czyli znowu coś z tym programowaniem funkcyjnym. **Functional interface** to taki interfejs, który ma tylko jedną metodę abstrakcyjną. Może jednocześnie mieć wiele innych metod defaultowych oraz statycznych, ale metodę abstrakcyjną może mieć tylko jedną.

Przypomnij sobie, że jeżeli napiszemy w interface jakąkolwiek metodę w sposób pokazany poniżej, to metoda taka jest domyślnie rozumiana jako `public abstract`, czyli jest publiczna i abstrakcyjna.

```
public interface SomeInterface {

    String someMethod(final Integer arg1, String arg2);
}
```

Mając już zdefiniowany interface w sposób pokazany powyżej, możemy zaimplementować ten interface w sposób "klasyczny", czyli stworzyć klasę, która zaimplementuje ten interface i nadpisać metodę `someMethod()`. Możemy również wykorzystać do tego mechanizm **lambdy**. Lambdą możemy zaimplementować tylko interfejs z jedną metodą abstrakcyjną. Nie możemy w ten sposób implementować klasy abstrakcyjnej, taka została podjęta decyzja przez twórców Javy.

Zostało to zrobione w ten sposób, bo gdy mamy tylko jedną metodę abstrakcyjną w takim interface to Java jest w stanie sama wykombinować, która z metod jest implementowana przy wykorzystaniu lambdy. Jeżeli w interface mielibyśmy dwie metody abstrakcyjne, to na moment pisania tego tekstu nie ma mechanizmu, który pozwoliłby wybrać, która z metod ma być zaimplementowana tę konkretnie lambdą. Filozoficznie patrząc, można próbować wywnioskować, że jeżeli dana lambda ma podane parametry pasujące do sygnatury tylko jednej z metod to dałoby się to jakoś ograć, ale na moment

pisania tego tekstu możemy implementować lambdą tylko interfejs z jedną metodą abstrakcyjną.

Adnotacja @FunctionalInterface

Tak jak zostało wspomniane wcześniej, na tym etapie zakładamy, że wiesz już, że interface, który możemy określić jako interfejs funkcyjny możemy zaimplementować przy wykorzystaniu lambdy. Zanim jednak przejdziemy dalej, powiemy sobie o ułatwieniu, jakie daje nam adnotacja `@FunctionalInterface`.

Pamiętasz adnotację `@Override` i to, że dawała nam ona pewne sprawdzenia, dzięki czemu nie musieliśmy sprawdzać niektórych rzeczy ręcznie, bo kompilator zrobi to za nas? Podobnie jest w tym przypadku. Jeżeli chcemy zaznaczyć, że jakiś interface jest funkcyjny i w ten sposób dać o tym znać albo innym developerom albo sobie z przyszłości to zastosujemy adnotację `@FunctionalInterface`.

Wspominam tutaj o 'sobie z przyszłości', bo jak czytamy ten sam kod za 3 miesiące to często kompletnie nie pamiętamy o co nam chodziło, więc lepiej jest się zabezpieczać na przyszłość 😁.

Adnotacja `@FunctionalInterface` jest umieszczana nad definicją interfejsu, który uznajemy za funkcyjny, np:

```
@FunctionalInterface
public interface SomeInterface {

    String someMethod(final int arg1, final boolean arg2);

}
```

Wcześniej zostało wspomniane, że interfejs funkcyjny pozwala nam mieć metodę defaultową, poniżej przykład:

```
@FunctionalInterface
public interface SomeInterface {

    String someMethod(final String arg1, final String arg2);

    default String someDefaultMethod() {
        System.out.println("Calling some default method");
        return "called some default method";
    }

}
```

Oprócz metody defaultowej możemy też mieć metodę statyczną i taki interfejs nadal będzie funkcyjny o ile mamy tylko jedną metodę abstrakcyjną.

```
@FunctionalInterface
public interface SomeInterface {

    String someMethod(final Integer arg1, String arg2);
```

```
    default String someDefaultMethod() {
        System.out.println("Calling some default method");
        return "called some default method";
    }

    static String someStaticMethod() {
        System.out.println("Calling some static method");
        return "called some static method";
    }

}
```

Zastosowanie adnotacji `@FunctionalInterface` objawi się jednak gdy będziemy próbowali do interfejsu, który ma być rozumiany jako interfejs funkcyjny dodać więcej niż jedną metodę abstrakcyjną:

```
// @FunctionalInterface
// Jeżeli teraz odkomentujemy adnotacją @FunctionalInterface to dostaniemy błąd kompilacji
public interface SomeInterface {

    String someMethod(final Integer arg1, String arg2);

    String someMethod2(final Integer arg1, String arg2);

    default String someDefaultMethod() {
        System.out.println("Calling some default method");
        return "called some default method";
    }

    static String someStaticMethod() {
        System.out.println("Calling some static method");
        return "called some static method";
    }

}
```

Po co w takim razie stosować tę adnotację? Jeżeli chcemy zaznaczyć, że dany interface ma być używany w kontekście interfejsu funkcyjnego i w przyszłości gdy ktoś będzie chciał to zmienić to musi mieć na uwadze, że popsuje kilka funkcjonalności, gdzie ten interface jest użyty aby implementować go lambdą.

Przykładowo możemy znaleźć takie zastosowanie w interfejsach `Comparator` oraz `Comparable`. Jak spojrzysz w ich definicje to zwrócisz uwagę, że `Comparator` jest oznaczony jako `@FunctionalInterface`, a `Comparable` już nie (przynajmniej na etapie pisania tego tekstu). Oznacza to, że `Comparator` ma być stosowany w formie lambdy, a `Comparable` nie. Możemy to samo wykorzystać przy pisaniu naszych programów żeby dać znać, że jeden interface ma być stosowany do implementowania go lambdą, a inny już nie.

Method reference

Method reference, (który można rozumieć jako referencję do metody) jest mechanizmem, który również został wprowadzony w Java 8 aby skrócić zapis lambdy. Chociaż czasami (rzadko) tak to skraca, że wychodzi jeszcze dłużej, ale o tym przekonasz się w praktyce 😁. **Method reference** bazuje na tym, że niektóre lambdy mogą zostać zastąpione nazwą metody, której sygnatura pasuje w danym wywołaniu i

może być ona użyta zamiast konkretnej lambdy. Czyli zamiast pisać lambdę możemy wskazać "referencję" do metody, która może zostać użyta zamiast lambdy. Wtedy interesuje nas ciało tej metody, które zostanie wywołane w momencie gdy miałaby być wywołana wspomniana lambda. Tak samo jak w przypadku lambdy, będzie miało tutaj miejsce **deferred execution**. Aby zastosować mechanizm **method reference** wykorzystujemy zapis `NazwaKlasy::NazwaMetody`. Sam mechanizm **method reference** może być stosowany zarówno w przypadku metod statycznych, metod instancyjnych i konstruktorów.

O ile początkowo może wydawać się to trudne w zrozumieniu, to z praktyką i doświadczeniem okazuje się, że ten mechanizm jest całkiem przydatny ☺. Przechodząc natomiast do przykładów.

Metody statyczne

Zacznijmy od pokazania przykładów z metodami statycznymi. Zdefiniujmy dwa interfejsy funkcyjne `MilkProducer` z metodą `produce()` oraz `MilkConsumer` z metodą `consume()`. Metody te są oznaczone jako 11 i 12. Każda z nich może być zaimplementowana przy wykorzystaniu lambdy co zostało pokazane w linijkach oznaczonych jako 1 i 2. Gdy mamy już zdefiniowane takie implementacje tych interfejsów, możemy wywołać ten kod w linijkach 3 i 4.

Innym sposobem na implementację interfejsu `MilkProducer` jest właśnie **method reference**. Pokażemy to na przykładzie poniżej.

```
public class MethodReferenceExamples {

    public static void main(String[] args) {
        MilkProducer milkProducer = () -> "someString"; ①
        MilkConsumer milkConsumer = someVariable -> "anotherString"; ②

        System.out.println(milkProducer.produce()); ③
        System.out.println(milkConsumer.consume("what to consume")); ④

        MilkProducer milkProducer2 = MethodReferenceExamples::milkReference1; ⑤
        MilkConsumer milkConsumer2 = MethodReferenceExamples::milkReference2; ⑥

        System.out.println(milkProducer2.produce()); ⑦
        System.out.println(milkConsumer2.consume("what to consume")); ⑧
    }

    private static String milkReference1() { ⑨
        return "someStringFromMethod";
    }

    private static String milkReference2(String arg) { ⑩
        return "anotherStringFromMethod: " + arg;
    }

    interface MilkProducer {
        String produce(); ⑪
    }

    interface MilkConsumer {
        String consume(String toConsume); ⑫
    }

}
```

W kodzie jest zdefiniowana metoda `milkReference1()`, jest oznaczona jako 9. Metoda ta nie przyjmuje żadnych argumentów i zwraca `String`. Czyli w sumie można zauważyć, że sygnatura tej metody pasuje do sygnatury metody w linijce 11. Zatem możemy wykorzystać metodę z linijki 9 do implementacji metody w linijce 11. Następuje to w linijce 5. Mówimy tym zapisem, że aby zaimplementować interfejs funkcyjny `MilkProducer` i zdefiniowaną w nim metodę z linijki 11 chcemy wykorzystać metodę z linijki 9, która jest zdefiniowana w klasie `MethodReferenceExamples`. To samo zdanie w kodzie jest zapisane jako `MethodReferenceExamples::milkReference1`. Specjalnie rozróżniliśmy metody po nazwach, ale Javie nie jest to potrzebne. Moglibyśmy nazwać obie metody z linijek 9 i 10 nazwą `milkReference()` i Java rozróżniłaby o którą metodę nam chodzi w linijce 5 na podstawie sygnatury.

To samo dzieje się analogicznie w linijce 6. Linijka 6 określa implementację interfejsu `MilkConsumer`. Interface ten jest funkcyjny i definiuje metodę w linijce 12. Możemy zauważyć, że metoda z linijki 10 nadaje się do implementacji metody z linijki 12. Możemy to zatem zapisać w linijce 6, mówiąc, że chcemy zaimplementować interface `MilkConsumer` wykorzystując metodę `milkReference2()` z klasy `MethodReferenceExamples`. Zapisując to w kodzie wygląda to tak `MethodReferenceExamples::milkReference2`. Tak samo jak w poprzednim przypadku, nie musimy rozróżniać metod z linijek 9 i 10 nazywając je inaczej. Java jest w stanie sama wywnioskować, którą metodę chcemy użyć w linijkach 5 i 6 na podstawie sygnatury tych metod. Zatem nazywanie metod `milkReference1` i `milkReference2` nie jest potrzebne. Obie mogą się nazywać `milkReference`, bo mają różne sygnatury.

Oczywiście teraz ten kod należy wywołać w linijkach 7 i 8 aby zostało wydrukowane na ekranie to co jest zdefiniowane w ciałach metod implementujących.

Jeżeli ktoś jest teraz zdziwiony, że metodą statyczną zaimplementowaliśmy interfejs funkcyjny, to mogę napisać "no cóż... można i tak".

Aby się opatrzeć z tym zapisem, poniżej możesz znaleźć jeszcze jeden przykład:

```
public class MethodReferenceExamples {

    public static void main(String[] args) {
        Calculator calculator1 = (a, b) -> a + b; ①
        int added = calculator1.add(5, 9);
        System.out.println(added);

        Calculator calculator2 = MethodReferenceExamples::add;
        // Calculator calculator3 = MethodReferenceExamples::someMethod; ②
    }

    private static int someMethod() {
        return 12;
    }

    static int add(int a, int b) {
        return a + b;
    }

    interface Calculator {
        int add(int a, int b);
    }
}
```

① Intellij w tej linijce sam podpowiada, żeby wykorzystać zapis `Integer::sum` zamiast lambdy.

② Błąd kompilacji bo sygnatura metody `someMethod` nie pasuje do metody `add()` z interfejsu `Calculator`.

Metody instancyjne

Teraz przyjedziemy do przykładu mechanizmu **method reference** z metodami instancyjnymi.

```
public class MethodReferenceExamples {

    public static void main(String[] args) {
        MethodReferenceExamples examples = new MethodReferenceExamples();
        examples.run();
    }

    private void run() {
        String burek = Optional.of(new Dog("Burek"))
            // zamiast .map(dog -> dog.getName())
            .map(Dog::getName) ①
            .orElse("otherDogName");

        Optional.of(new Dog("other Burek"))
            // zamiast .ifPresent(dog -> printSomething(dog));
            .ifPresent(MethodReferenceExamples::printSomething); ②

        Optional.of(new Dog("doggo again"))
            // zamiast .ifPresent(dog -> printDoggy(dog))
            .ifPresent(this::printDoggy); ③
    }

    private void printDoggy(final Dog dog) { ④
        System.out.println("printing doggy");
    }

    private static void printSomething(final Dog dog) {
        System.out.println("printing");
    }

    private class Dog {

        private final String name;

        public Dog(final String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }
    }

}
```

Oczywiście kolejny przykład komplikuje bardziej sytuację. Pokazane tutaj zostały 3 sposoby wykorzystania mechanizmu **method reference**, przy czym jeden z nich już znamy (ten z numerem 2, jest tutaj statyczna metoda `printSomething()` wywołana z klasy `MethodReferenceExamples`). Ciekawy natomiast jest zapis z linijki 1 i 3.

W linijce 1 metoda `map()`, przyjmuje interface `Function` (o którym niedługo się dowiemy). Na ten moment wystarczy nam wiedza, że metoda `map()` przyjmuje interfejs funkcyjny, w którym jest zdefiniowana metoda, która przyjmuje jeden argument dowolnego typu i zwraca jeden rezultat dowolnego typu. Czyli jako argument metody `map()` możemy przekazać lambdę `dog → dog.getName()`, gdzie parametrem wejściowym jest `Dog`, a wyjściowym `String`. Czyli inaczej mówiąc, skoro metoda `map()` przyjmie lambdę, która na wejściu ma `Dog`, a na wyjściu `String`, to równie dobrze możemy znaleźć metodę w klasie `Dog`, która zwróci nam jakąś wartość ze swojego stanu, dlatego wywołujemy tutaj getter = `getName()`. To właśnie mówi zapis `Dog::getName`. Jednocześnie zauważ, że korzystamy tutaj z metody `getName()`, która nie przyjmuje żadnych parametrów. Dajemy w ten sposób namiar na metodę, która jest zdefiniowana w klasie `Dog`, nie ma parametrów wejściowych i zwraca jakiś inny typ. Typem zwracanym nie musi być `String`, może być cokolwiek innego.

W linijce 3 natomiast wywołujemy metodę `ifPresent()`, która przyjmuje interfejs funkcyjny `Consumer` (o którym niedługo się dowiemy). Na ten moment wystarczy nam wiedza, że metoda `ifPresent()` przyjmuje interfejs funkcyjny, w którym jest zdefiniowana metoda, która przyjmuje jeden argument dowolnego typu i nic nie zwraca - tylko konsumuje. Skoro ten interfejs funkcyjny określa metodę, która coś przyjmuje i nic nie zwraca to do tej sygnatury pasuje metoda `printDoggy()` z linijki 4. Przyjmuje ona klasę `Dog` i nic nie zwraca. Zapis z linijki 3 `this::printDoggy` oznacza, że zamiast lambdy, (która jest zakomentowana) możemy tutaj przekazać metodę `printDoggy()`, która jest instancyjna i zdefiniowana w tej samej klasie - dlatego słówko `this`. Próba wywołania tego samego w formie `MethodReferenceExamples::printDoggy` oznaczałaby próbę wywołania w kontekście statycznym i dostaniemy wtedy błąd kompilacji bo metoda `printDoggy()` nie jest statyczna.

Czyli jeżeli chcemy wskazać **method reference** do metody instancyjnej z innej klasy niż obecnie się znajdujemy, która jednocześnie pasuje do kontekstu wywołania (czyli z klasy `Dog` robimy klasę `String`) to zastosujemy zapis `Dog:getName`. Jeżeli natomiast chcemy wskazać referencję do metody z tej samej klasy, w której jest obecnie kontekst naszego obiektu to napiszemy `this::printDoggy`.

Poniżej możesz znaleźć jeszcze jeden przykład - dla opatrzenia się z tym mechanizmem 😁 (klasa `Dog` ma tę samą sygnaturę).

```
public class MethodReferenceExamples {

    public static void main(String[] args) {
        MethodReferenceExamples examples = new MethodReferenceExamples();
        examples.run();
    }

    private void run() {
        List<Dog> dogs = new ArrayList<>();
        dogs.sort((a, b) -> a.getName().compareTo(b.getName())); ①
        dogs.sort(Comparator.comparing(Dog::getName)); ②

        Optional.of(new Dog("Fafik")).map(Dog::getName); ③

        Optional.of(new Dog("Fafik")).ifPresent(fafik -> System.out.println(fafik)); ④
        Optional.of(new Dog("Fafik")).ifPresent(System.out::println); ⑤

        String someName = "someName";
        Optional.of(new Dog("Fafik")).ifPresent(dog -> System.out.println(dog + someName)); ⑥
    }

}
```

① Zapis w tej linijce mówi, że implementacja komparatora polega na tym, że porównujemy wynik `getName()` z `a` z wynikiem `getName()` z `b`.

② Pokazuje to samo wywołanie co 1 i zastąpienie, które proponuje nam Intellij. Zamiast tak jak w linijce 1 podawać, że porównujemy wynik `getName()` z `a` z wynikiem `getName()` z `b`, możemy to zapisać tak jak w linijce 2. Czyli, że będziemy porównywać ze sobą dwie wartości, które zwraca nam metoda `getName()` z klasy `Dog`.

③ Pokazuje ten sam przykład, który był pokazany wcześniej.

④ Pokazuje klasyczny zapis wydruku przy wykorzystaniu metody `Optional.ifPresent()`.

⑤ To jest ciekawy przykład. Możemy w ten sposób zastąpić zapis z linijki 4. Zapis ten jest o tyle ciekawy, że metoda `ifPresent()` przyjmuje interface `Consumer`, który został opisany wcześniej. Czyli jako implementację lambdy możemy napisać to tak jak w linijce 4. Możemy też w takim razie wskazać referencję do metody `println()`, która jest zdefiniowana w polu statycznym `out`, w klasie `System`. Możemy wykorzystać tę metodę, gdyż jej sygnatura jest analogiczna do metody z interfejsu `Consumer`. Czyli przyjmujemy coś i zwracamy nic.

⑥ Zapis w tej linijce u**nie**możliwia nam zastosowanie **method reference** bo zanim wywołamy metodę `println()` wykonujemy konkatenację `dog + someName`. W takim przypadku użycie **method reference** nie jest możliwe.

Konstruktory

Możesz już na tym etapie się domyślać (i skoro napisane zostało już o tym wcześniej 😁), że skoro możliwe jest wskazanie referencji do metody, która ma określoną sygnaturę, to równie dobrze możemy to zrobić z konstruktorem, który ma określoną sygnaturę. Przykład poniżej.

```
public class MethodReferenceExamples {

    public static void main(String[] args) {
        MethodReferenceExamples examples = new MethodReferenceExamples();
        examples.run();
    }

    private void run() {
        SteeringWheel steeringWheel = new SteeringWheel(12.40);

        Car car = Optional.of(steeringWheel)
            .map(Car::new) // zamiast .map(sw -> new Car(sw))  ①
            .orElse(new Car(new SteeringWheel(0.0)));
    }

    private class Car {
        private final SteeringWheel steeringWheel;

        public Car(final SteeringWheel steeringWheel) {
            this.steeringWheel = steeringWheel;
        }
    }

    private class SteeringWheel {
        private final double diameter;

        public SteeringWheel(final double diameter) {
            this.diameter = diameter;
```

```
        }
    }
}
```

① Linijka ta pokazuje pokazuje przykład wywołania metody `map()`, która przyjmuje interfejs funkcyjny `Function`, który został wspomniany wcześniej. Interface ten określa metodę, która przyjmuje obiekt A i zwraca obiekt B. W tym przypadku możemy zatem określić implementację, która na podstawie `obiektu klasy SteeringWheel` stworzy obiekt klasy `Car`. To właśnie pokazuje lambda w linijce 1. A zapis `Car::new`? Oznacza on wskazanie referencji do konstruktora klasy `Car`. Efekt tego wywołania będzie taki sam jak lambdy pokazanej w komentarzu w linijce 1.

Podsumowanie

Mając już wiedzę na temat interfejsów funkcyjnych oraz tego jak je definiować, dokładając wiedzę o adnotacji `@FunctionalInterface` i method reference możemy przejść do omówienia wbudowanych interfejsów funkcyjnych, które są dostępne w Java i z których możemy śmiało korzystać.