Lambda

Spis treści

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

Przedsmak

Pokazanie materiału o lambdach ma za zadanie dać taki mały przed smaczek programowania funkcyjnego. W pełnym zakresie ten materiał zostanie zrealizowany w ramach warsztatów. Na razie to jest tylko mały przedsmak.

Programowanie funkcyjne i "→", czyli lambda

> Czyli, że piszemy funkcje, a nie metody i tyle tak?

Programowanie funkcyjne jest sposobem na pisanie kodu bardziej deklaratywnie. To oznacza, że bardziej piszemy, co chcemy osiągnąć, zamiast opisywać, jak można to zrobić. Prowadzi to do tego, że bardziej skupiamy się na pisaniu instrukcji mówiących o tym, co ma być efektem naszego działania, niż rozpisywaniu jak do tego efektu dojść. Zobaczysz co mam na myśli jak wejdziemy już głębiej w temat ☺.

Jednocześnie w tym podejściu dajemy możliwość przypisywania funkcji do zmiennych. Wcześniej tego nie robiliśmy prawda? A niedługo zobaczysz, że można przekazać funkcję jako argument metody.

Poruszymy w tym materiale tematykę lambd, czyli prostego sposobu na implementowanie metod w interfejsach, które mają tylko jedną metodę. Interfejs z jedną metodą abstrakcyjną jest nazywany w Javie interfejsem funkcyjnym. Jeżeli kojarzysz coś takiego jak **closure** z innych języków, to lambda jest czymś analogicznym. Innymi słowy, lambda jest jak metoda, którą możesz przekazywać do metod, jakby była zmienną. Możesz ją też przypisywać do innych zmiennych.

Lambda cechuje się też odroczonym wykonaniem (po angielsku brzmi to o wiele lepiej - *Deferred execution*). Możesz zadeklarować lambdę 'teraz' ale uruchomić ją o wiele później w kodzie.

Pamiętajmy też, że z założenia, Java jest językiem obiektowym. W podejściu funkcyjnym staramy się bardziej skupiać na opisaniu, co ma być efektem naszego działania, niż zajmować się stanem obiektów.

Lambdą możemy zaimplementować tylko interfejs z jedną metodą abstrakcyjną. Nie możemy w ten sposób implementować klasy abstrakcyjnej, taka została podjęta decyzja przez twórców Javy.

Jaki problem lambda rozwiązuje

Wyobraź sobie taki interface:

```
public interface Checkable {

    boolean test(Animal a);
}
```

Zgodnie z tym, co napisałem, jest to interfejs funkcyjny, bo zawiera deklarację tylko jednej metody.

Wyobraź sobie, że mamy teraz taki fragment kodu (klasa `Animal` ma pole `name`, konstruktor, getter i metodę `toString()`):

```
public class LambdaExample {

    public static void main(String[] args) {
        List<Animal> animals = List.of(
            new Animal("rabbit"),
            new Animal("dog"),
            new Animal("bird")
        );
    }

    private static void print(List<Animal> animals, Checkable checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.println(checker.toString() + ": " + animal);
            }
        }
    }
}
```

Chcemy wykonać sprawdzenie `checker.test(animal)`, wiedząc, że argumentem metody `print()` jest interfejs `Checkable`.

Z wiedzą, którą mamy dotychczas, wiemy, że należałoby stworzyć klasę np. `CheckIsRabbit`, utworzyć jej instancję i przekazać utworzony obiekt do metody `print()`.

```
public class CheckIsRabbit implements Checkable {
    @Override
    public boolean test(final Animal a) {
        return "rabbit".equals(a.getName());
    }
}
```

Następnie moglibyśmy wykonać metodę `print()` w następujący sposób:

```
print(animals, new CheckIsRabbit());
```

Czyli żeby to zrobić, musimy mieć stworzony fizycznie na dysku plik `CheckIsRabbit.java`, musimy go napisać, co chwilkę zajmuje, a na koniec stworzyć obiekt tej klasy, tylko żeby zrobić proste sprawdzenie.

W klasie `CheckIsRabbit` mamy określone konkretne zachowanie będące implementacją metody `test()` z interfejsu Checkable. Oczywiście, jeżeli chcielibyśmy dokonać innego sprawdzenia, musielibyśmy stworzyć następną klasę np. `CheckIsDog` i zaimplementować w niej konkretne sprawdzanie, czy przekazany obiekt `Animal` jest psem.

Wracamy do lambdy

Jak zrobić to samo za pomocą lambdy? O tak:

```
print(animals, a -> "rabbit".equals(a.getName()));
print(animals, a -> "dog".equals(a.getName()));
```

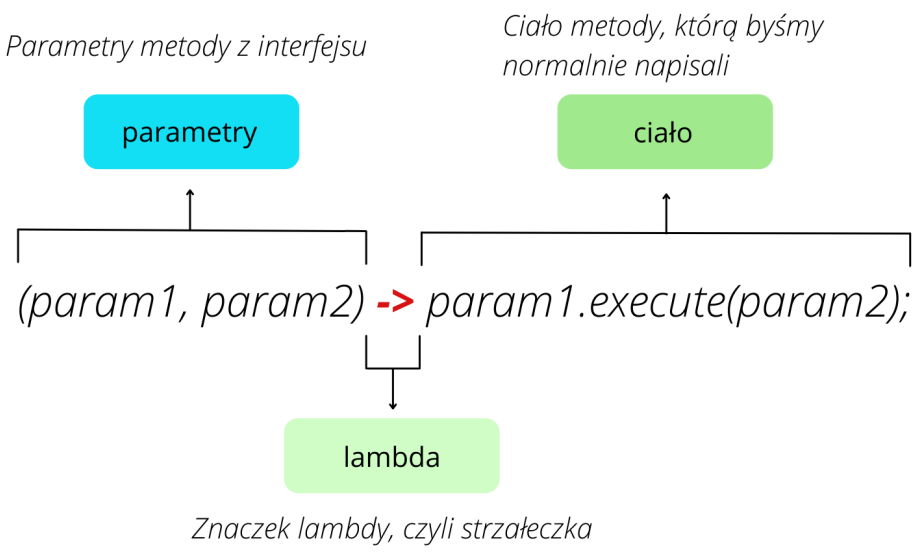
Dzięki temu zapisowi udało nam się dokonać sprawdzenia, czy obiekt jest psem albo królikiem w jednej linijce.

Podstawy konstrukcji lambda

Ten temat jest bardzo ważny (wiem, że często mówimy, że któryś temat jest ważny i wychodzi na to, że w sumie to wszystko jest ważne, no ale tu tak jest). Zapis lambdy jest często używany na co dzień, dlatego poświęć czas na zrozumienie tej konstrukcji. Skupmy się jednak na składni lambdy:

```
a -> "rabbit".equals(a.getName())
```

Czyli:



Zapis, który został pokazany, jest zapisem skrótowym. Zauważ, że nie podajemy typu zmiennej, deklarując parametr lambdy. Nie musimy w tym przypadku tego robić, bo Java jest w stanie się tego 'domyślić', bo przecież implementujemy tylko jedną metodę z interfejsu funkcyjnego, więc Java sama jest w stanie sobie sprawdzić jakiego typu jest parametr lambdy. Na końcu wyrażenia w lambdzie nie ma też słówka return, pomimo że jest to przecież implementacja metody, która coś zwraca. W końcu jest to zapis skrócony ☺.

Pełny zapis wyglądałby w ten sposób:

```
(Animal a) -> {
    return "rabbit".equals(a.getName());
}; // tutaj musi być średnik
```

Czyli podajemy jakiego typu jest parametr lambdy, na którym będziemy operować, dodajemy słówko `return`, pojawia się też średnik na końcu wyrażenia. Zwróć uwagę na nawiasy, które pojawiły się wokół `(Animal a)`, oraz na nawiasy klamrowe definiujące gdzie lambda się zaczyna i kończy.

Lambdę można przypisać do zmiennej, czyli możemy napisać coś takiego:

```
Checkable checkable = a -> "rabbit".equals(a.getName());
```

Rozmawiając o składni lambdy, podajmy kilka przykładów.

Implementujemy metodę, która nie przyjmuje parametrów i zwraca `boolean`:

```
SomeInterface someVariable = () -> true;

// możemy zapisać to również tak
SomeInterface someVariable = () ->  {
    return true;
};
```

W przykładach powyżej, gdy nie mieliśmy żadnego parametru w implementowanej metodzie, musimy podać puste nawiasy, nie możemy tego pominąć.

Implementacja metody, która przyjmuje jeden parametr typu `String` i zwraca `Stringa`:

```
SomeInterface someVariable = param -> param.toUpperCase();

// możemy zapisać to również tak
SomeInterface someVariable = (param) -> param.toUpperCase();

// możemy zapisać to również tak
SomeInterface someVariable = (String param) -> param.toUpperCase();

// możemy też tak
SomeInterface someVariable = param -> {
    return param.toUpperCase();
};

// i możemy też tak
SomeInterface someVariable = (String param) -> {
    return param.toUpperCase();
};
```

Możemy też implementować metody, które przyjmują więcej niż jeden parametr, wszystkie poniższe zapisy są równoważne:

```
SomeInterface someVariable = (a, b) -> a + b;
SomeInterface someVariable = (Integer a, Integer b) -> a + b;
SomeInterface someVariable = (a, b) -> {
    return a + b;
};
SomeInterface someVariable = (Integer a, Integer b) -> {
    return a + b;
};
```

Jakich kombinacji alpejskich natomiast robić nie wolno?

```
// Gdy mamy więcej niż jeden parametr, muszą być one w nawiasach
SomeInterface someVariable = a, b -> a + b; // błąd kompilacji

// Jeżeli zaczniemy używać nawiasów klamrowych,
// to muszą być one napisane w pełni,
// nie możemy pominąć ani słówka return ani średnika
SomeInterface someVariable = (a, b) -> {
    a + b; // brak return, błąd kompilacji
};

SomeInterface someVariable = (a, b) -> {
    return a + b // brak średnika, błąd kompilacji
};
```

Zapis z nawiasem klamrowym często jest stosowany wtedy, gdy chcemy w lambdzie wykonać kilka instrukcji po sobie. Istnieje też filozofia, że jak potrzebujemy zrobić coś takiego, to lepiej jest wyciągnąć to, co powinno być w nawiasach klamrowych do metody i wywołać wtedy w ciele lambdy tę metodę, kod staje się wtedy czytelniejszy.

Wspomniałem też wcześniej o **deferred execution**. Polega to na tym, że lambda nie zostanie uruchomiona na etapie jej definiowania, czyli linijkę wyżej, jeżeli chcielibyśmy lambdę uruchomić,

musimy faktycznie wywołać metodę, którą lambda implementuje, czyli:

```
Checkable checkable = a -> "rabbit".equals(a.getName());
// kod zdefiniowany w lambdzie wywoła się dopiero w linijce niżej
checkable.test(animals.get(0));
```

Dzięki temu możemy definiować lambdę w jednym miejscu w kodzie, a faktycznie uruchamiać ją w innym.

Czego jeszcze nie wolno robić?

Jeżeli w parametrach lambdy zdefiniujemy jakąś nazwę zmiennej, nie możemy stworzyć zmiennej o takiej samej nazwie w ciele lambdy. Tak samo, jak w metodach:

```
SomeInterface someVariable = (a, b) -> {
    int a = 10; // błąd kompilacji
    return a + b;
};
```

Trzeba też pamiętać, że aby w ciele lambdy można było użyć jakiejś zmiennej, która jest zdefiniowana przed definicją lambdy, to musi ona być **effectively final**. Przykład:

```
int variable = 10;
variable = 15;
// dostaniemy błąd kompilacji przez zmienną variable, bo nie jest ona effectively final
Function<Integer, Integer> someVariable = (param) -> param + variable;
```

Zmienna 'variable' byłaby **effectively final**, jeżeli po jej inicjalizacji nie zmienialibyśmy jej wartości, czyli nie próbowali potem przypisać do niej wartości 15. Inaczej mówiąc, zmienna jest **effectively final**, jeżeli możemy dopisać do niej słówko `final` i nie dostaniemy błędu kompilacji.

```
final int variable = 10
variable = 15; // tutaj dostaniemy błąd kompilacji
```

Czy używanie lambdy ma sens?

Jeżeli pojawia Ci się pytanie: *Czy jak użyjemy lambdy to oznacza, że nie musimy tworzyć żadnego obiektu?*, to odpowiem tak: na pewno nie tworzymy obiektu `new CheckIsRabbit()`, bo zarówno lambda, jak i `CheckIsRabbit` są sposobami na osiągnięcie tego samego efektu, czyli zaimplementowanie interfejsu `Checkable`, ale użycie lambdy nie oznacza utworzenia obiektu new `CheckIsRabbit()`, bo są to 2 inne mechanizmy.

Natomiast na pytanie: *czy nie jest tworzony żaden nowy obiekt podczas użycia lambdy i dzięki temu oszczędzamy pamięć?* na tym etapie ciężko jest odpowiedzieć na to pytanie, dlatego postaram się to uprościć.

Jeżeli lambda nie ma parametrów, to tworzona jest taka jakby jej "statyczna" instancja i później JVM

może ją reużyć. Jeżeli natomiast lambda operuje na pewnych obiektach będących jej argumentem - specyfikacja daje JVM dowolność jak do tego podejść, czy dla każdej lambdy tworzyć nowy obiekt, czy starać się w jakiś sposób reużywać istniejące już obiekty. Czyli każda implementacja JVM, w zależności od vendora, może zachowywać się w inny sposób.

Dlatego też skupmy się na ten moment na tym, że lambda daje nam możliwość użycia bardziej zwięzłego zapisu do osiągnięcia tego samego efektu.

Wbudowane interfejsy funkcyjne

W przykładach powyżej zdefiniowaliśmy interface Checkable, w którym metoda przyjmowała `Animal` i zwracała `boolean`. Twórcy Javy starali się wyjść użytkownikom naprzeciw i przewidzieć typowe interfejsy funkcyjne, które programista musiałby napisać sam. Dlatego API Javy daje nam kilka interfejsów funkcyjnych, które możemy wykorzystywać. Będą one poruszone w przyszłości, bo przecież teraz rozmawiamy o wstępie do programowania funkcyjnego, ale na tę chwilę chciałem poruszyć taki interface jak `Predicate`.

`Predicate`, to interfejs funkcyjny, który dostarcza nam metodę, która przyjmuje dowolny obiekt jako argument i zwraca `true`. Metoda ta nazywa się `test()`. Mam nadzieję, że widzisz już, że mówię o tym, bo zamiast pisać interface `Checkable`, moglibyśmy użyć interfejsu `Predicate`.

```
public class LambdaExample {

    public static void main(String[] args) {
        List<Animal> animals = List.of(
            new Animal("rabbit"),
            new Animal("dog"),
            new Animal("bird")
        );
    }

    private static void print(List<Animal> animals, Predicate<Animal> checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.println(checker.toString() + ": " + animal);
            }
        }
    }
}
```

W przypadku Listy mogliśmy napisać jakie typy możemy przetrzymywać w kolekcji, tak samo jest tutaj, `Predicate` przyjmuje typ generyczny, na którym będziemy operować. Dokładnie o typach generycznych będziemy rozmawiać później.

Jeżeli teraz chcielibyśmy wywołać metodę `print()` przekazując implementację interfejsu `Predicate`, możemy to zrobić tak samo, jak poprzednio:

```
print(animals, a -> "rabbit".equals(a.getName()));
```

To tyle na ten moment, w warsztatach będziemy bardziej zgłębiać tematykę ☺.