

Wyrażenia lambda i interfejsy funkcyjne

Czego się dowiesz

- Czym są wyrażenia lambda,
- czym są interfejsy funkcyjne,
- przykłady wykorzystania interfejsów Function, Consumer, Predicate

Wstęp

Java od zawsze była prezentowana jako sztandarowy przykład języka obiektowego, tzn. wszystko ma być obiektem i już (oprócz typów prostych). W Javie 8 zaczęto wprowadzać koncepcje programowania funkcyjnego, które w ostatnich latach jest coraz bardziej doceniane. Programowanie funkcyjne jest początkowo ciężkie do zrozumienia dla kogoś, kto rozpoczynał programowanie w języku obiektowym czy proceduralnym, nie jest ono do końca naturalne w rozumieniu tego, że program jest po prostu pewnym skończonym ciągiem instrukcji, które zmieniają stan obiektów i zmiennych w programie.

W programowaniu funkcyjnym skupiamy się na wyniku poprzez jego opis za pomocą matematycznej funkcji. Języki funkcyjne dostarczają nam narzędzi, które pozwalają na opis problemu i pozwalają nie martwić się jak właściwie coś jest wykonywane.

W Javie 8 wprowadzono elementy programowania funkcyjnego, które są przydatne przede wszystkim w przetwarzaniu kolekcji, czyli np. list. Pomagają także zastępować klasy anonimowe prostymi wyrażeniami, które są po prostu bardziej czytelne. Nie oznacza to oczywiście, że od teraz wszędzie gdzie tylko się da należy używać w Javie programowania funkcyjnego, jednak pokażę kilka zastosowań, gdzie staje się to po prostu przydatne i znacząco skraca zapis. Na początku zaczniemy jednak od podstawowych pojęć i mechanizmów.

Wyrażenia Lambda

Wyrażenie lambda, to taka konstrukcja w języku Java, która przyjmuje dowolne argumenty lub nie przyjmuje ich wcale i może zwracać na ich podstawie pewien wynik.

W rozumieniu matematycznym funkcję rozumiemy, np jako:

$$f(x) = x * x$$

czyli funkcję potęgującą dowolną liczbę. W Javie zapiszemy ją jako wyrażenie lambda takiej postaci:

(int x) -> x*x;

Widzimy tutaj trzy elementy:

- **(int x)** - argumenty funkcji podane w nawiasach okrągłych,
- **->** - strzałka, czyli operator wskazujący, że jest to wyrażenie lambda,
- **x*x** - wyrażenie, ciało funkcji.

Możliwy jest także zapis w nieco innej formie.

1. Wykorzystanie słowa kluczowego return

(int x) -> return x*x;

2. Pomińcie typów argumentów funkcji.

Język Java jest silnie typowany, czyli typ zmiennych musi być znany na etapie kompilacji. Przekazując argumenty funkcji kompilator będzie najczęściej w stanie wydedukować ich typ na podstawie wykonywanego wyrażenia, więc przy argumentach możemy pominąć podawanie typu i zapisać krótko:

(x) -> x*x;

3. Dodatkowe lub zbędne nawiasy.

Możemy wykonać bardziej złożone instrukcje w ciele funkcji, a także pominąć nawiasy okrągłe przy jej argumentach:

```
x -> { if(x>0) return x*x; else return 0; }
```

przy czym nie muszą one być zapisane w jednym wierszu.

Na tym etapie wyrażenia lambda wydają się bezużyteczne, ale wyobraź sobie teraz, że taką funkcję możemy przekazać np. do kolekcji i zostanie ona wykonana na każdym elemencie naszej listy. Będzie to znaczna oszczędność czasu, a także podniesie to czytelność kodu.

Interfejsy funkcyjne

Jeżeli chcesz w Javie przekazać wyrażenie lambda jako argument metody, czy konstruktora, albo chcesz je przypisać zmiennej, to musisz najpierw zdefiniować interfejs funkcyjny (eng. functional interface) lub skorzystać z jednego z dostępnych interfejsów funkcyjnych w bibliotece JDK.

Interfejsem funkcyjnym nazywamy interfejs, który posiada tylko jedną metodę abstrakcyjną.

W Javie 8 zdefiniowano interfejsy funkcyjne dla najpopularniejszych zastosowań, pełną listę możesz znaleźć **pod tym linkiem**, poniżej wymieniono te, które pokażemy na przykładach:

- **Consumer<T>** - posiada metodę **accept(T t)** - przyjmuje argument typu T, ma za zadanie wykonać pewną operację i nie zwraca wyniku,
- **Function<T, R>** - posiada metodę **apply(T t)** - reprezentuje funkcję przyjmującą argument typu T i zwracającą argument typu R,
- **Predicate<T>** - posiada metodę **test(T t)** - przyjmuje argument typu T i zwraca wartość typu boolean,
- **Supplier<T>** - posiada metodę **get()** - tworzy nowy obiekt typu T.

Interfejs Function

Najprostszym przykładem wykorzystania funkcji może być transformacja pewnego napisu na inny. Załóżmy, że chcemy stworzyć funkcję, która przyjmuje jako argument dowolny napis, a jej zadaniem jest zwrócenie tego samego napisu, ale zamienionego na małe litery i bez zbędnych białych znaków na początku i na końcu.

Moglibyśmy po prostu wywołać kilka metod jedna po drugiej osiągając zamierzony efekt:

```
String original = "    WIELKI NAPIS    ";  
original = original.toLowerCase().trim();
```

Powyższy kod można także opakować w metodę, która przyjmuje String i w wyniku także zwraca String. Zamiast wywoływać kilka metod, można wywołać tylko ją:

```
String getLowerCaseTrim(String original) {  
    return original.toLowerCase().trim();  
}
```

Powyższa metoda przyjmuje obiekt typu String i zwraca w wyniku również String. Odpowiada ona więc takiemu wyrażeniu lambda jak `String s -> String`. Wśród interfejsów funkcyjnych znajdziemy interfejs Function, którego metoda *apply()* pasuje do tej sytuacji. Ma ona następującą sygnaturę:

```
R apply(T t);
```

Jak widać, jest to interfejs parametryzowany typami generycznymi. Pod T oraz R możemy podstawić dowolne typy obiektowe. Jeżeli pod jeden i drugi parametr podstawimy typ String, to metoda przyjmuje obiekt String i zwraca String - dokładnie tak samo jak w naszym przykładzie. Interfejs ten możemy wykorzystać w naszym kodzie, a metodę *getLowerCaseTrim()* możemy zastąpić wyrażeniem lambda. Na początek przypiszmy wyrażenie lambda do zmiennej:

```
Function<String, String> func = (String s) -> s.toLowerCase().trim();
```

Wyrażenie lambda, zapisane po prawej stronie równania, przyjmuje parametr typu String, zamienia tekst na małe litery, usuwa białe znaki z końca i początku i zwraca tak zmodyfikowany napis w wyniku. Ze względu na to, że nasza zmienna ma określony typ generyczny, czyli zapisaliśmy `Function<String, String>`, to wirtualna maszyna Javy może wywnioskować typy i wyrażenie lambda można zapisać z pominięciem typu:

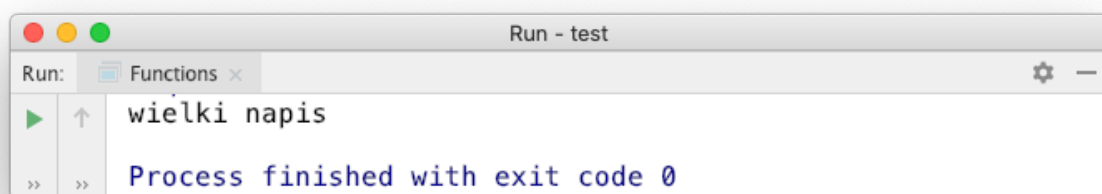
```
Function<String, String> func = text -> text.toLowerCase().trim();
```

W celu wywołania utworzonej funkcji z jakimś argumentem należy wywołać metodę, która zdefiniowana jest w naszym interfejsie funkcyjnym. U nas chodzi o metodę *apply()*.

```
import java.util.function.Function;

class Functions {
    public static void main(String[] args) {
        // funkcja przyjmuje String i zwraca String
        Function<String, String> func = text -> text.toLowerCase().trim();
        String original = "    WIELKI NAPIS    ";
        // wywołujemy funkcję przekazując jej original jako argument
        String lowerCaseTrim = func.apply(original);
        System.out.println(lowerCaseTrim);
    }
}
```

Zapis `func.apply(original)` oznacza "wywołaj wyrażenie lambda/funkcję przypisaną do zmiennej `func` z argumentem `original`".



Interfejs Consumer

Założmy, że teraz chcemy wyświetlić jakiś tekst trzy razy. Możemy dodać po prostu trzy instrukcje `System.out.println()` co wyglądałoby tak:

```
class Functions {  
    public static void main(String[] args) {  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("abc");  
    }  
}
```

Jeżeli w ramach tej samej metody będziemy chcieli wyświetlić także inny tekst trzy razy, to zaczyna nam się duplikować dużo kodu:

```
class Functions {  
    public static void main(String[] args) {  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("abc");  
        System.out.println("xxx");  
        System.out.println("xxx");  
        System.out.println("xxx");  
    }  
}
```

Pierwsze co robimy w takiej sytuacji, to wydzielamy metodę z powtarzalnym kodem. W naszym przypadku metoda ta będzie przyjmowała `String` i wyświetliła go 3 razy.

```
class Functions {  
    public static void main(String[] args) {  
        print3Times("abc");  
        print3Times("xxx");  
    }  
  
    private static void print3Times(String abc) {  
        System.out.println(abc);  
        System.out.println(abc);  
        System.out.println(abc);  
    }  
}
```

Metoda `print3Times()` odpowiada takiemu wyrażeniu lambda jak `String s -> void`, czyli przyjmuje parametr typu `String` i nie zwraca nic w wyniku. Wśród interfejsów funkcyjnych znajduje się interfejs `Consumer` z metodą `accept()` o takiej sygnaturze:

```
void accept(T t);
```

Również jest to interfejs generyczny, więc jeśli pod T podstawimy *String*, to otrzymujemy taką samą sygnaturę jak nasza metoda *print3Times()*. Oznacza to, że zamiast metody, możemy zdefiniować wyrażenie lambda, które przypiszemy do zmiennej *Consumer*.

```
import java.util.function.Consumer;

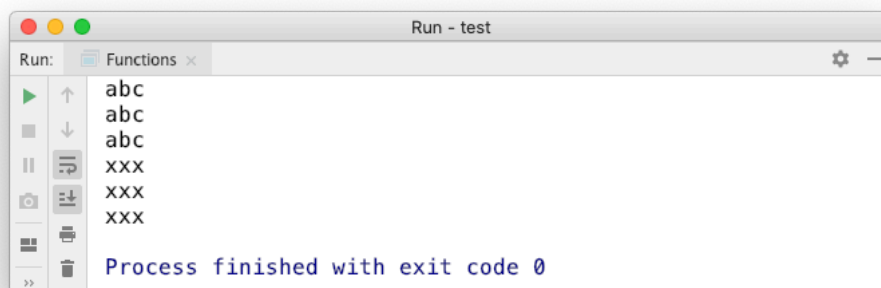
class Functions {
    public static void main(String[] args) {
        Consumer<String> print3Times = s -> {
            System.out.println(s);
            System.out.println(s);
            System.out.println(s);
        };

        print3Times.accept("abc");
        print3Times.accept("xxx");
    }
}
```

Ponieważ w ramach ciała wyrażenia lambda wykonujemy kilka operacji, konkretnie wywołujemy trzykrotnie metodę *System.out.println()*, to konieczne było dodanie dodatkowych nawiasów klamrowych. Zauważ, że w ramach nawiasów klamrowych nie ma żadnego returna, wynika to z sygnatury metody *accept()*, która jest typu *void*.

W celu wywołania wyrażenia lambda, na zmiennej *print3Times* wywołujemy metodę *accept()*, przekazując jej argument, który chcemy wyświetlić.

W tym przypadku nie zyskujemy wiele w porównaniu do zdefiniowania osobnej metody, czytelność pozostaje praktycznie na tym samym poziomie, ilość samego kodu również jest zbliżona, ale jak widzisz jest to ciekawa alternatywa.



Interfejs Predicate

Założmy, że mamy w programie zmienną z wiekiem pewnej osoby. Chcemy sprawdzić, czy osoba ta jest pełnoletnia. Jeśli tak, to wykonamy pewną operację. Standardowo zapiszemy to w taki sposób:

```
class Functions {  
    public static void main(String[] args) {  
        int personAge = 19;  
        if (personAge >= 18) {  
            //jakieś operacje  
        }  
    }  
}
```

Kod ten da się usprawnić. Nie do końca wiadomo, dlaczego porównujemy się akurat z wartością 18 - jest to magiczna liczba. Moglibyśmy wprowadzić jakąś stałą, ale jeszcze lepiej będzie wydzielić metodę, której nadamy nazwę, czyli znaczenie.

```
class Functions {  
    public static void main(String[] args) {  
        int personAge = 19;  
        if (checkIfAdult(personAge)) {  
            //jakieś operacje  
        }  
    }  
  
    static boolean checkIfAdult(int age) {  
        return age >= 18;  
    }  
}
```

Dzięki takiemu zapisowi raczej już jest jasne co oznacza liczba 18.

Metoda *checkIfAdult()* przyjmuje parametr typu *int* i zwraca wartość typu *boolean*, odpowiada więc takiemu wyrażeniu lambda jak *int x -> boolean*. Odpowiada to interfejsowi *Predicate*, w którym zdefiniowana jest metoda *test()*:

```
boolean test(T t);
```

Jeżeli pod *T* podstawimy typ *Integer*, to otrzymujemy niemal to samo co w naszym przypadku. Dzięki temu, że mamy mechanizm autoboxingu i autounboxingu, możemy założyć, że nie będzie tutaj żadnego problemu. Zamiast osobnej metody, możemy więc zdefiniować zmienną typu *Predicate* i przypisać do niej wyrażenie lambda, które będzie pasowało do sygnatury metody *test()*.


```
import java.util.function.Predicate;

class Functions {
    public static void main(String[] args) {
        int personAge = 19;
        Predicate<Integer> checkIfAdult = age -> age >= 18;
        if (checkIfAdult.test(personAge)) {
            //jakieś operacje
        }
    }
}
```

Rozwiązanie takie ma sens szczególnie wtedy, kiedy danego predykatu używamy tylko w jednej metodzie. Definiowanie osobnych metod może nam niepotrzebnie "zaśmiecać" klasę i musimy się wtedy zastanawiać, czy metoda taka była wykorzystywana tylko w tym jednym miejscu, czy jeszcze gdzieś. Definiując zmienną lokalną, tak jak powyżej, nie ma tego problemu.

Interfejs Supplier

W ostatnim przykładzie pokażę Ci jak możemy tworzyć obiekty z danymi wylosowanymi z kilku tablic. Na początku potrzebna będzie nam klasa *Person*, która reprezentuje osobę, a każda osoba opisana jest przez imię, nazwisko i wiek.

Person.java

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

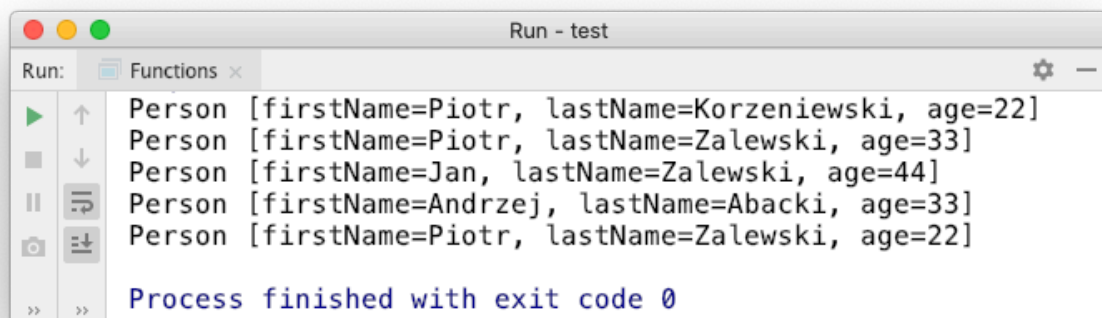
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```
public int getAge() {  
    return age;  
}  
  
public void setAge(int age) {  
    this.age = age;  
}  
  
@Override  
public String toString() {  
    return "Person [firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + "];"  
}  
}
```

Teraz w metodzie main zdefiniuję tablice z imionami, nazwiskami i liczbami reprezentującymi wiek osób. Na początku stworzymy listę 5 obiektów Person z losowym imieniem, nazwiskiem i wiekiem korzystając z klasycznej pętli.

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
class Functions {  
    public static void main(String[] args) {  
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};  
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};  
        int[] ages = {22, 33, 44, 55};  
        Random random = new Random();  
        List<Person> people = new ArrayList<>();  
        for (int i = 0; i < 5; i++) {  
            String firstName = firstNames[random.nextInt(firstNames.length)];  
            String lastName = lastNames[random.nextInt(lastNames.length)];  
            int age = ages[random.nextInt(ages.length)];  
            Person randomPerson = new Person(firstName, lastName, age);  
            people.add(randomPerson);  
        }  
        for (Person person : people) {  
            System.out.println(person);  
        }  
    }  
}
```

W wyniku zobaczysz np.:



```
Run: Functions x  
Person [firstName=Piotr, lastName=Korzeniewski, age=22]  
Person [firstName=Piotr, lastName=Zalewski, age=33]  
Person [firstName=Jan, lastName=Zalewski, age=44]  
Person [firstName=Andrzej, lastName=Abacki, age=33]  
Person [firstName=Piotr, lastName=Zalewski, age=22]  
  
Process finished with exit code 0
```

Obiekty standardowo tworzymy wywołując konstruktor, zapisując:

```
Person person = new Person("Jan", "Kowalski", 42);
```

Można powiedzieć, że wywołanie konstruktora to trochę jakby powiedzieć, że tworzymy coś z niczego. Gdyby odzwierciedlić to przy pomocy wyrażenia lambda, to miałyby ono np. taką sygnaturę `() -> Person`.

Wśród interfejsów funkcyjnych istnieje coś takiego jak `Supplier`, gdzie zdefiniowana jest metoda `get()` o takiej sygnaturze:

```
T get();
```

Metoda `get()` nie przyjmuje żadnych parametrów i zwraca obiekt typu `T`. Jeżeli pod `T` podstawimy `Person`, to otrzymujemy to, czego szukamy. Zapiszmy fragment kodu i wyrażenie lambda, które pozwoli nam utworzyć obiekty `Person` z losowymi danymi.

```
class PersonOperators {  
    public static void main(String[] args) {  
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};  
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};  
        int[] ages = {22, 33, 44, 55};  
        Random random = new Random();  
        Supplier<Person> supplier = () -> {  
            String firstName = firstNames[random.nextInt(firstNames.length)];  
            String lastName = lastNames[random.nextInt(lastNames.length)];  
            int age = ages[random.nextInt(ages.length)];  
            return new Person(firstName, lastName, age);  
        };  
        System.out.println(supplier.get());  
        System.out.println(supplier.get());  
        System.out.println(supplier.get());  
    }  
}
```

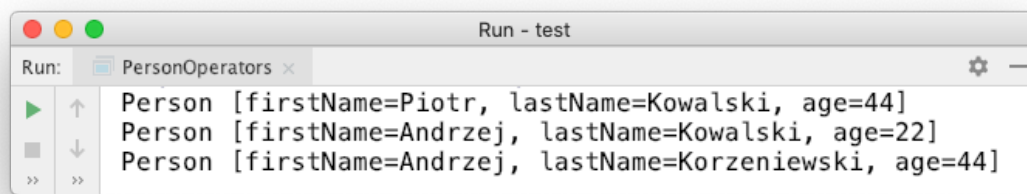
Na początku zdefiniowane są trzy tablice przechowujące odpowiednio imiona, nazwiska oraz wiek osób. Obiekt typu `Random` pozwoli nam wylosować liczbę, odpowiadającą indeksowi obiektu lub wartości, który będziemy pobierali z tablicy. Do zmiennej `supplier` przypisujemy wyrażenie lambda, które jest zgodne z sygnaturą abstrakcyjnej metody `get()` zdefiniowanej w tym interfejsie. Takie wyrażenie lambda nie przyjmuje żadnego argumentu, a zwraca obiekt, w naszym przypadku typu `Person`.

Zapis typu `firstNames[random.nextInt(firstNames.length)]` oznacza "pobierz z tablicy *firstNames* obiekt zapisany pod losowym indeksem z zakresu od 0 do długości tej tablicy. Analogicznie postępujemy z losowaniem nazwiska i wieku, a na końcu zwracamy gotowy obiekt *Person*.

Wywołując metodę `get()`, za każdym razem zostanie utworzony obiekt *Person* z losowym imieniem, nazwiskiem i wiekiem. Im więcej danych będzie w tablicach źródłowych, tym lepiej. Zamiast wywoływać metodę `get()` bezpośrednio w metodzie `println()`, moglibyśmy oczywiście też najpierw obiekty zapamiętać w zmiennych:

```
Person person1 = supplier.get();
System.out.println(person1);
```

Po uruchomieniu powyższego przykładu zobaczysz np. taki wydruk:



Teraz wystarczy dodać do tego wszystkiego pętlę i gotowe.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.function.Supplier;

class Functions {
    public static void main(String[] args) {
        String[] firstNames = {"Jan", "Karol", "Piotr", "Andrzej"};
        String[] lastNames = {"Abacki", "Kowalski", "Zalewski", "Korzeniewski"};
        int[] ages = {22, 33, 44, 55};
        Random random = new Random();
        Supplier<Person> supplier = () -> {
            String firstName = firstNames[random.nextInt(firstNames.length)];
            String lastName = lastNames[random.nextInt(lastNames.length)];
            int age = ages[random.nextInt(ages.length)];
            return new Person(firstName, lastName, age);
        };
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            people.add(supplier.get());
        }
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

Podsumowanie

W tej lekcji pokazałem ci podstawową mechanikę korzystania z wyrażeń lambda. Zapamiętaj, że w Javie istnieje coś takiego jak interfejsy funkcyjne, czyli takie interfejsy, które mają tylko jedną metodę abstrakcyjną (mogą mieć także inne metody, np. domyślne, statyczne, albo prywatne). Do zmiennej tego typu można przypisać wyrażenie lambda, które będzie zgodne z metodą zdefiniowaną w takim interfejsie. Prawdziwe korzyści korzystania z wyrażeń lambda zobaczysz jednak, gdy połączymy je z operacjami na kolekcjach oraz strumieniach.