# Notatki - Programowanie funkcyjne

## Spis treści



Ta cześć notatek dotyczy wprowadzenia teoretycznego do programowania funkcyjnego. Zaparz sobie dobrą kawę i możemy jechać z tematem 😄.

# Czym jest paradygmat programowania

Cytując Wikipedię:

> Paradygmat programowania definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego. Przykładowo, w programowaniu obiektowym jest on traktowany jako zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu funkcyjnym definiujemy, co trzeba wykonać, a nie w jaki sposób.

## Programowanie imperatywne a deklaratywne

- **Programowanie imperatywne** – paradygmat programowania, który opisuje proces wykonywania jako sekwencję instrukcji zmieniających stan programu.
- **Programowanie deklaratywne** — paradygmat programowania, który stoi w opozycji do imperatywnego. W przeciwieństwie do programów napisanych imperatywnie, programista opisuje warunki, jakie musi spełniać końcowe rozwiązanie (co chcemy osiągnąć), a nie szczegółową sekwencję kroków, które do niego prowadzą (jak to zrobić).

Różnice między tymi podejściami:

*Tabela 1. Programowanie imperatywne a deklaratywne*

| Programowanie imperatywne | Programowanie deklaratywne |
|---|---|
| Opisujesz jak coś zrobić | Opisujesz co chcesz osiągnąć |
| Możemy przekazywać typy | Możemy przekazywać typy i funkcje |
| Typy mutowane | Typy niemutowane |
| Wątkowo-niebezpieczne (Not thread safety) | Wątkowo-bezpieczne (Thread safety) |

# Programowanie funkcyjne w skrócie

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, przykładowo funkcja może być przekazana jako argument metody.

Mówiąc o programowaniu funkcyjnym będziemy rozmawiać też o interfejsach funkcyjnych. Interfejs z jedną metodą abstrakcyjną jest nazywany w Javie interfejsem funkcyjnym. Będziemy też mocno używać lambd, (które zakładam, że masz już na tym poziomie opanowane. Jeżeli nie, to w ramach warsztatu znajdziesz materiały wyjaśniające/odświeżające).

Tematyka lambd jest tutaj mocno poruszana, gdyż 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. Lambdą możemy zaimplementować tylko interfejs z jedną metodą abstrakcyjną, czyli interface funkcyjny. Nie możemy w ten sposób implementować klasy abstrakcyjnej, taka została podjęta decyzja przez twórców Javy.

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.

Przejdźmy natomiast do pojęć.

# Pojęcia związane z programowaniem funkcyjnym

## First Class Citizen

First Class Citizen - (polska Wikipedia nazywa to Typem Pierwszoklasowym) - oznacza to określenie typu danych, który:

- może być przechowywany w zmiennej

- może być przekazywany do metody
- może być zwracany przez wywołanie metody
- posiada tożsamość

Z angielskiej wikipedii (bo lepsza 😄) link:

> In programming language design, a first-class citizen (also type, object, entity, or value) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable.

# Brak stanu

Brak stanu oznacza brak stanu zewnętrznego w odniesieniu do funkcji. Funkcja może mieć swój chwilowy tymczasowy stan wewnętrzny, ale nie może odwoływać się do żadnych zmiennych, pól, atrybutów klasy, w której się znajduje. Dla przykładu, funkcja poniżej nie modyfikuje stanu obiektu, w którym jest zdefiniowana.

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

A w przykładzie poniżej już tak:

```
public class SomeClass {

    private int someField = 0;

    public int someMethod(int a) {
        someField = a;
        return a * 2;
    }
}
```

# Immutability

Immutable variables, zmienne niemutowalne. Zakładam, że wiesz już na tym etapie, w jaki sposób osiągnąć definicję klasy immutable. W skrócie, dążymy do tego, żeby w żaden sposób nie dało się zmodyfikować stanu obiektu, na którym wykonujemy operację. Jeżeli chcemy coś zmienić to musimy stworzyć nowy obiekt na podstawie tego poprzedniego ze zmienioną wartością jakiegoś pola. Takie podejście ułatwia nam unikania efektów ubocznych, które są wyjaśnione poniżej.

# Brak efektów ubocznych

Oznacza to, że funkcja nie może zmienić żadnego stanu obiektów zewnętrznych, które znajdują się poza

funkcją. Zmiana stanu czegokolwiek co znajduje się poza funkcją jest rozumiana jako efekt uboczny - *side effect*. Stan w tym przypadku odnosi się np. do pól w klasie, w której jest zdefiniowana metoda. Stan w tym rozumieniu może się też odnosić do stanu danych na dysku komputera, na którym pracujemy, lub też stanu obiektów w bazie danych, na których możemy operować.

# Function jako First Class Citizen

Oznacza to, w odniesieniu do definicji o tym co oznacza stwierdzenie **First Class Citizen**, że funkcję możesz przypisać do zmiennej (stworzyć instancję funkcji), dokładnie tak jak ze Stringiem czy Psem. Możesz przyjąć funkcję jako zmienną w metodzie i również możesz taką funkcję zwrócić przy wywołaniu metody. Zobaczysz, że w Javie jest to zrealizowane w ten sposób, że funkcja może być reprezentowana jako `Obiekt` stworzony na podstawie definicji interfejsu `Function`. Do tego dojdą nam lambdy i wszystko się poukłada.

## Dygresja o refleksji

Tutaj mała dygresja, w Javie istnieje też mechanizm nazywany **refleksją**. Nie chcę tej tematyki poruszać bo często spotykałem się z opinią, że jeżeli zabierasz się za używanie refleksji w kodzie, to znaczy, że coś poszło nie tak na etapie rozkminiania/rozumienia problemu. Sam z resztą też tak uważam ☺. Najczęściej jest ona natomiast używana do pisania frameworków i wtedy już może się przydać (w telegraficznym skrócie framework to duży zestaw narzędzi i bibliotek w kodzie, z którego możesz korzystać, będzie o tym potem). W skrócie w refleksji chodzi o to, że program może sam siebie modyfikować. Podczas działania programu możesz przeprowadzić inspekcję obiektu, na którym operujesz. Możesz pobrać listę pól obiektu, listę metod oraz dużo innych informacji. Następnie możesz takie pola, metody modyfikować w trakcie działania programu.

Dlatego o tym wspominam, bo ktoś może zacząć się zastanawiać czy refleksja jest przykładem programowania funkcyjnego. Moje zdanie na ten temat jest krótkie. Mówi się, że programowanie funkcyjne zostało wprowadzone w Javie 8. Refleksja istniała wcześniej. Skoro wtedy nie mówiło się o tym, że w Javie można programować funkcyjnie, to rozumiem, że refleksja takiego podejścia nie reprezentuje. Jeżeli jest to dla Ciebie interesujące to spróbuj sobie pogooglać o tym czym jest refleksja, nie chcę wchodzić w tę tematykę, bo uważam, że nie jest na tym etapie potrzebna, tymczasem ciśniemy dalej 😄.

# Czyste funkcje (Pure functions)

Aby funkcja była pure musi spełniać takie założenia:

- Nie mieć side effects
- Wartość zwracana przez funkcję zależy tylko od parametrów wejściowych, tzn. wynik funkcji nie zależy np. od wartości pól obiektu

Przykład czystej funkcji:

```
public class PureFunctionBelow{
    public int sum(int a, int b) {
        return a + b;
    }
```

```
}
```

Tak jak wspominałem, wynik zależy tylko od parametrów wejściowych. Funkcja nie ma *side effects*, nie modyfikuje stanu obiektu. I znowu, przykład funkcji nie-pure:

```
public class NonPureFunctionBelow{
    private int state = 12;
    public int sum(int a, int b) {
        this.state += a;
        return this.state += b;
    }
}
```

Zauważ jak zmienna `state` jest użyta w funkcji i jak zmienia to stan obiektu.

# Higher Order Functions

Żeby funkcja była wyższego rzędu, musi spełniać poniższe wytyczne (jedno lub drugie):

- Funkcja przyjmuje jako parametr jedną lub więcej funkcji
- Funkcja zwraca jako rezultat inną funkcję

W Javie możemy osiągnąć takie zachowanie poprzez implementację funkcji, która przyjmuje jako parametr wyrażenie lambda i zwraca po wykonaniu inne wyrażenie lambda. Treść przykładu jest bez sensu, więc się tym nie przejmuj, no ale - przykład:

```
public class HigherOrderFunctionExample {

    public static void main(String[] args) {
        HigherOrderFunctionExample example = new HigherOrderFunctionExample();
        example.call(() -> "banana", something -> System.out.println("it is food"));
    }

    public <T> Food<T> call(Bag<T> bag, Monkey<T> monkey) {
        return () -> {
            T something = bag.get();
            monkey.isFood(something);
            return something;
        };
    }

    private static interface Bag<T> {
        T get();
    }

    private static interface Food<T> {
        T bite();
    }

    private static interface Monkey<T> {
        public void isFood(final T something);
    }
```

```
}
```

Zwróć uwagę, że metoda `call()` przyjmuje 2 wyrażenia lambda jako argument wywołania i jednocześnie zwraca wyrażenie lambda, będące implementacją interfejsu zdefiniowanego jako typ zwracany z metody. Zakładam, że znasz już na tym etapie takie pojęcie jak **interfejs funkcyjny**. Jest to taki interfejs, który posiada tylko jedną metodę abstrakcyjną, która może być zaimplementowana przy wykorzystaniu wyrażenia lambda.

W przykładzie powyżej widać, że wymienione interfejsy są interfejsami funkcyjnymi - mogą być zaimplementowane przy wykorzystaniu wyrażenia lambda. Dzięki zapisowi jak wyżej możemy powiedzieć, że `call()` jest **Higher Order Function**.

## Rekurencja ponad pętle

Aby zrozumieć rekurencję musisz zrozumieć rekurencję... Funkcja w podejściu funkcyjnym zamiast wywoływać pętlę w środku, wywołuje sama siebie, w ten sposób realizując pętlę.

# Co możemy z tym zrobić?

W Javie możliwość programowania funkcyjnego została wprowadzona w wersji 8. Jednocześnie mówi się, że to właśnie ósemka była przełomowa. W Javie, możliwości programowania funkcyjnego służą do wykonywania określonych czynności bez spełnienia wszystkich założeń czysto funkcyjnych języków programowania jakim jest np. **Haskell**. Można zatem powiedzieć, że w Javie niektóre koncepcje programowania funkcyjnego zostały ograniczone, dlatego skupimy się tylko na tym **co Java potrafi**, a nie na tym czego nie 😉.

# Często zadawane pytania

**Czy programowanie funkcyjne jest szybsze?**

Java nie została zaprojektowana z myślą o programowaniu funkcyjnym. Dlatego mechanizmy, które zostaną przedstawione w tym warsztacie często są mniej wydajne. Dużo przykładów kodu wykona się szybciej w implementacji imperatywnej. Trzeba o tym pamiętać, a zwłaszcza wtedy jak zabieramy się za pracę na dużej ilości danych. Z drugiej strony natomiast, często może być tak, że mimo, że zajmie to dłużej to wykonanie takiego kodu może zająć mniej pamięci operacyjnej, więc warto rozważyć te kwestie na indywidualnych przypadkach.

**Czy programując funkcyjnie mamy czytelniejszy kod?**

Bardzo sporna kwestia, bo tyczy się to gustu. Wiadomo, jak ktoś tych zapisów nie zna to dla niego nie będzie to bardziej czytelne 😄. Składnia nie jest oczywista, trzeba z nią się obyć, więc czasem nie warto komplikować. Czasem czytelniejsze jest napisanie czegoś imperatywnie, zdarza się.

**Po przeczytaniu/wysłuchaniu tego wszystkiego pojawia się pytanie, po co właściwie ja o tym wszystkim gadam?**

W tym warsztacie poruszymy tematykę jak można podejść w Javie do pisania funkcyjnego. Zobaczysz, że wielu interfejsów funkcyjnych nie musisz pisać na własną rękę bo Java już trochę ich dostarcza. No i

pogadamy o Streamach, czyli jak fajnie przetwarzać elementy jak jest ich kilka.