Notatki - Lambda - Przypomnienie

Spis treści

Notatki dotyczące lambdy dotyczą przypomnienia czym są lambdy i po co się je stosuje. Jeżeli znasz dobrze ten temat, to nie musisz przez nie przechodzić - no chyba, że chcesz, to nie zabraniam 😈.

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.

To co z tą lambdą

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.

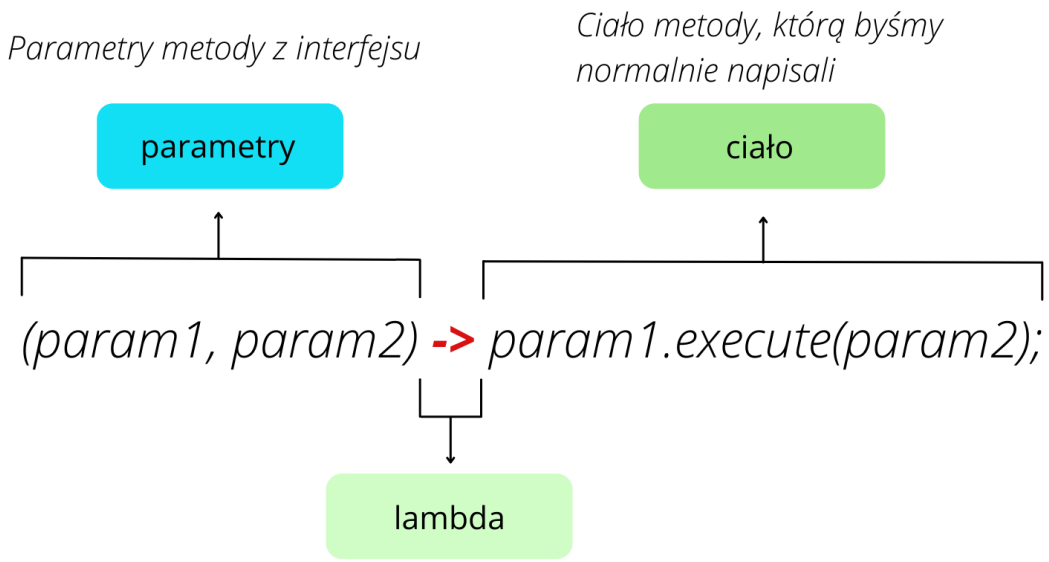
Podstawy konstrukcji lambda

Generalnie 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:



Znaczek lambdy, czyli strzałeczka

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, żę jest to przecież implementacja metody, która coś zwraca. Nie ma też średnika. 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.

Z lambdą związany jest również mechanizm **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, 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.

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.

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.

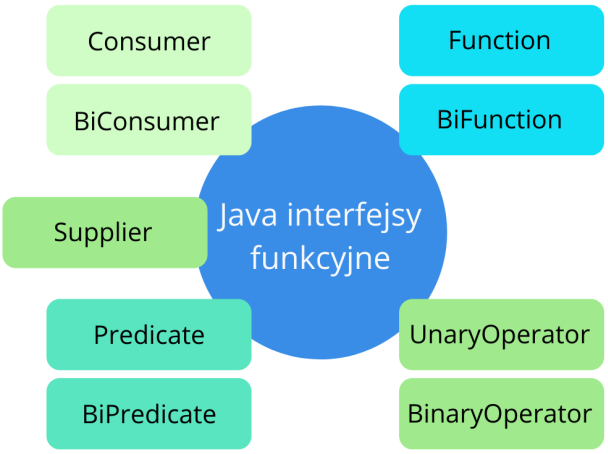
Notatki - Wbudowane Interfejsy Funkcyjne

Spis treści

Wbudowane interfejsy funkcyjne

We wcześniejszych przykładach rozmawialiśmy o własnych interfejsach funkcyjnych oraz o tym jakie warunki interface musi spełniać aby był uznany za interface funkcyjny.

Twórcy Javy starali się wyjść użytkownikom na przeciw i przewidzieć typowe interfejsy funkcyjne, które programista musiałby napisać sam. Dlatego API Javy daje nam kilka interfejsów funkcyjnych, które możemy wykorzystywać.



Obraz 1. Interfejsy funkcyjne, które oferuje Java

Wszystkie wymienione interfejsy zostaną omówione poniżej.

Predicate

Predicate, to interfejs funkcyjny, który dostarcza nam metodę `test()`, która przyjmuje dowolny obiekt jako argument i zwraca `boolean`. Stosujemy go najczęściej, gdy chcemy "odfiltrować" jakąś wartość.

Definicja tego interfejsu wygląda w ten sposób:

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

    boolean test(T t);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class LambdaExample {

    public static void main(String[] args) {
        List<Animal> animals = List.of(
            new Animal("rabbit"),
            new Animal("dog"),
            new Animal("bird")
        );
        print(animals, a -> "rabbit".equals(a.getName())); ①
    }

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

`Predicate` przyjmuje typ generyczny, który jest parametrem wejściowym metody `test()`. Typem zwracanym metody `test()` jest `boolean`. Określając lambdę, która implementuje ten interface musimy podać jeden argument, którego typ jest określony typem generycznym, natomiast typem zwracanym tej lambdy musi być boolean. Oczywiście możemy też zastosować **method reference**.

Kolejne wykorzystanie `Predicate` w praktyce:

```
public class ExamplePredicate {

    public static void main(String[] args) {
        Predicate<String> predicate1 = someString -> someString.isEmpty(); ①
        Predicate<String> predicate2 = String::isEmpty; ②
        System.out.println(predicate1.test("zajavka"));
        System.out.println(predicate2.test("zajavka is cool"));
```

```
        String someValue = Optional.of("zajavka is the coolest")
            // zamiast .filter(someString -> someString.isEmpty()) ③
            .filter(String::isEmpty) ④
            .orElse("nonEmpty");
    }

    private static void voidMethod() {}

    interface SomeInterface {
        void someMethod(String abc);
    }
}
```

① Implementacja `Predicate` wykorzystując lambdę.

② Implementacja `Predicate` wykorzystując **method reference**, która robi to samo co przykład 1.

③ Implementacja `Predicate` wykorzystując lambdę w metodzie `Optional.filter()`.

④ Implementacja `Predicate` wykorzystując **method reference**, która robi to samo co przykład 3.

Consumer

Consumer, to interfejs funkcyjny, który dostarcza nam metodę `accept()`, która przyjmuje dowolny obiekt jako argument i nic nie zwraca. Stosujemy go najczęściej, gdy chcemy "skonsumować" jakąś wartość i nic nie zwrócić.

Definicja tego interfejsu wygląda w ten sposób:

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

    void accept(T t);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleConsumer {

    public static void main(String[] args) {
        Consumer<String> consumer1 = value -> {return;}; ①
        //System.out.println(consumer1.accept("ain't gonna work")); ②

        Consumer<String> consumer2 = value -> {
            System.out.println("printing: " + value);
            return;
        }; ③
        consumer2.accept("this shall work"); ④

        Consumer<String> consumer3 = value -> someMethod(); ⑤
        consumer3.accept("this shall work"); ⑥

        Consumer<String> consumer4 = value -> {
```

```
            someMethod2();
            return;
        }; ⑦
        consumer4.accept("this shall work"); ⑧

        Consumer<String> consumer5 = value -> someMethod2(); ⑨
        consumer5.accept("this shall work"); ⑩

        Consumer<String> consumer6 = ExampleConsumer::someMethodReference; ⑪
        consumer6.accept("this shall work"); ⑫
    }

    private static void someMethod() {}

    private static boolean someMethod2() {
        return true;
    }

    private static void someMethodReference(String arg) {}
}
```

① Implementacja interfejsu przy wykorzystaniu lambdy, możemy zastosować tutaj słówko `return` na takiej samej zasadzie jak robimy to w metodach zwracających `void`.

② Wywołanie, które same w sobie jest poprawne, natomiast nie możemy przekazać rezultatu tego wywołania do metody `println()`, gdyż wywołanie zwraca `void`.

③ Implementacja interfejsu przy wykorzystaniu lambdy, gdzie najpierw drukujemy tekst na ekranie, a potem wychodzimy z lambdy jak w przykładzie 1.

④ Poprawne wywołanie metody.

⑤ Implementacja interfejsu przy wykorzystaniu lambdy, możemy zapisać to w ten sposób, zwyczajnie w ciele lambdy wywołujemy metodę, która nic nie przyjmuje, a zwraca `void`.

⑥ Poprawne wywołanie metody.

⑦ Implementacja interfejsu przy wykorzystaniu lambdy, możemy to zapisać w ten sposób, co jest trochę analogiczne do przykładu 5, na koniec wychodzimy słówkiem `return` jak w przykładzie 1.

⑧ Poprawne wywołanie metody.

⑨ Implementacja interfejsu przy wykorzystaniu lambdy, natomiast zapis ten może wydawać się nieintuicyjny ze względu na to, że metoda `someMethod2()` zwraca `true`. Ten zapis jest jednak analogiczny do zapisu 7, zatem jest poprawny.

⑩ Poprawne wywołanie metody.

⑪ Zastosowanie mechanizmu method reference do implementacji interfejsu.

⑫ Poprawne wywołanie metody.

Supplier

Supplier, to interfejs funkcyjny, który dostarcza nam metodę `get()`, która nic nie przyjmuje i zwraca nam obiekt dowolnego typu. Stosujemy go najczęściej, gdy chcemy "dostarczyć" jakąś wartość z niczego.

Definicja tego interfejsu wygląda w ten sposób:

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

    public T get();

}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleSupplier {

    public static void main(String[] args) {
        Supplier<String> supplier1 = () -> "value";
        String supplied1 = supplier1.get();
        System.out.println(supplied1); ①

        //Supplier<String> supplier2 = ExampleSupplier::provideInt; ②

        Supplier<String> supplier3 = () -> ExampleSupplier.provideString();
        String supplied3 = supplier3.get();
        System.out.println(supplied3); ③

        Supplier<String> supplier4 = ExampleSupplier::provideString;
        String supplied4 = supplier4.get();
        System.out.println(supplied4); ④
    }

    private static int provideInt() {
        return 23;
    }

    private static String provideString() {
        return "23";
    }

}
```

① Implementacja `Supplier` wykorzystując lambdę. Najprostszy przykład, w którym z niczego dostarczamy `String`.

② Implementacja `Supplier` wykorzystując method reference, typ generyczny określa `String`, natomiast metoda `provideInt()` dostarcza `int`, stąd mamy błąd kompilacji.

③ Implementacja `Supplier` wykorzystując lambdę. Lambda nie przyjmuje żadnego parametru, natomiast możemy w niej wywołać metodę zwracającą `String`. Intellij sam podpowiada aby zamienić to wywołanie na to w przykładzie 4.

④ Implementacja `Supplier` wykorzystując **method reference**, przykład robi dokładnie to samo co przykład 3.

BiPredicate

Oprócz interfejsów takich jak `Predicate` lub `Consumer` zobaczymy jeszcze interfejsy z przedrostkiem `Bi`. Oznacza to najczęściej, że coś dzieje się w takim interfejsie

> podwójnie.

BiPredicate, to interfejs funkcyjny, który dostarcza nam metodę `test()`, która przyjmuje dwa parametry dowolnego typu i zwraca `boolean`. Stosujemy go najczęściej, gdy chcemy "odfiltrować" jakąś wartość przyjmując dwa argumenty.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiPredicate {

    public static void main(String[] args) {
        BiPredicate<String, Dog> biPredicate1 = (str, dog) -> dog.likes(str);
        System.out.println(biPredicate1.test("abc", new Dog())); ①

        BiPredicate<String, Dog> biPredicate2 = ExampleBiPredicate::doggoLikes;
        System.out.println(biPredicate2.test("doggo", new Dog())); ②
    }

    private static boolean doggoLikes(String string, Dog dog) {
        return string.isEmpty();
    }
}
```

```
class Dog {

    public boolean likes(final String str) {
            return "doggo".equals(str);
    }
}
```

① Implementacja `BiPredicate` wykorzystując lambdę. Wywołując zdefiniowany interface przekazujemy dwa argumenty i wartością zwracaną jest `boolean`.

② Implementacja `BiPredicate` wykorzystując method reference. Definicja metody `doggoLikes()` pasuje (dwa parametry i typ zwracany boolean) do definicji metody `test()` z interfejsu `BiPredicate`, dlatego możemy wykorzystać metodę `doggoLikes()`.

BiConsumer

BiConsumer, to interfejs funkcyjny, który dostarcza nam metodę `accept()`, która przyjmuje dwa parametry dowolnego typu i nic nie zwraca. Stosujemy go najczęściej, gdy chcemy "skonsumować" jakieś dwie wartości i nic nie zwrócić.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    void accept(T t, U u);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiConsumer {

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

    private void run() {
        Cat cat = new Cat("Mruczek");
        Car car = new Car();

        BiConsumer<Cat, Car> biConsumer = (ct, cr) -> cr.enter(ct); ①

        System.out.println(car); ②
        biConsumer.accept(cat, car); ③
        System.out.println(car); ④
    }

}
```

```
class Cat {
    private final String name;

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

    @Override
    public String toString() {
        return "Cat{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

```
class Car {

    private Cat cat;

    public void enter(final Cat cat) {
        this.cat = cat;
    }
```

```
    @Override
    public String toString() {
        return "Car{" +
            "cat=" + cat +
            '}';
    }
}
```

① Implementacja `BiConsumer` wykorzystując lambdę, w której określamy, że `Cat` wsiada do `Car`. Lambda nic nie zwraca, konsumujemy dwa obiekty.

② Zawartość obiektu klasy `Car` bez kota w środku.

③ Kot wsiada do samochodu.

④ Zawartość obiektu klasy `Car` z kotem w środku.

BiSupplier

Ten jest szybki bo nie ma czegoś takiego 😁.

Function

Function, to interfejs funkcyjny, który dostarcza nam metodę `apply()`, która przyjmuje parametr dowolnego typu i zwraca parametr dowolnego typu. Stosujemy go najczęściej, gdy chcemy przekształcić jeden obiekt w drugi.

Definicja tego interfejsu wygląda w ten sposób:

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

    R apply(T t);

    // reszta metod
}
```

Warto zwrócić tutaj uwagę, że pierwszy typ generyczny oznacza przyjmowany przez metodę `apply()` parametr, natomiast drugi typ generyczny oznacza typ zwracany z metody `apply()`.

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleFunction {

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

    private void run() {
        Function<String, Cat> function1 = name -> new Cat(name); ①
        Function<String, Cat> function2 = Cat::new; ②
        System.out.println(function2.apply("mruczek"));
```

```
        Function<String, Integer> function3 = input -> input.length(); ③
        Function<String, Integer> function4 = String::length; ④
        System.out.println(function4.apply("this string length"));

        Function<String, Integer> function5 = s -> s.length() + 123; ⑤
        System.out.println(function5.apply("abc"));
    }
}
```

```
class Cat {
    private final String name;

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

    @Override
    public String toString() {
        return "Cat{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

① Implementacja `Function` wykorzystując lambdę. Wywoływany jest tutaj konstruktor, w którym tworzymy obiekt klasy `Cat` na podstawie przekazanego `Stringa`.

② Implementacja `Function` wykorzystując method reference, która robi to samo co linijka 1.

③ Implementacja `Function` wykorzystując lambdę. Wywoływana jest tutaj metoda `length()` z klasy String, dlatego definicja `Function<String, Integer>` określa najpierw `String`, bo jest to typ wejściowy do metody `apply()`, a potem `Integer` bo jest to typ zwracany z tej metody.

④ Implementacja `Function` wykorzystując method reference, która robi to samo co linijka 3.

⑤ Implementacja `Function` wykorzystując lambdę. W tym przypadku lambda wyciąga długość przekazanego `Stringa` i dodaje do niej `123`. Nie jest możliwe zapisanie tego samego przy wykorzystaniu method reference.

BiFunction

BiFunction, to interfejs funkcyjny, który dostarcza nam metodę `apply()`, która przyjmuje dwa parametry dowolnego typu i zwraca parametr dowolnego typu. Stosujemy go najczęściej, gdy chcemy przekształcić dwa obiekty w jakiś trzeci obiekt.

Definicja tego interfejsu wygląda w ten sposób:

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

    R apply(T t, U u);

    // reszta metod
```

```
}
```

Warto zwrócić tutaj uwagę, że pierwsze dwa typy generyczne oznaczają przyjmowane przez metodę `apply()` parametry, natomiast trzeci typ generyczny oznacza typ zwracany z metody `apply()`.

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiFunction {

    public static void main(String[] args) {
        BiFunction<String, Integer, Long> function1 = (a, b) -> (long) (a.length() + b);
        System.out.println(function1.apply("string1", 10)); ①

        BiFunction<String, String, Dog> function2 = (a, b) -> new Dog(a + b);
        System.out.println(function2.apply("mruczek", "mruczaasty")); ②

        BiFunction<String, String, Dog> function3 = Dog::new;
        System.out.println(function3.apply("kiciek","kiciasty")); ③
    }
}
```

```
class Dog {
    private final String name;

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

    public Dog(final String name1, final String name2) { ④
        this.name = name1 + name2;
    }

    @Override
    public String toString() {
        return "Dog{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

① Implementacja `BiFunction` wykorzystując lambdę. Przekazujemy dwa parametry `String` oraz `Integer` i zwracamy sumę długości `Stringa` i wartości `Integer`. Typem zwracanym jest `Long` stąd rzutowanie. Wywołując metodę `apply()` przekazujemy dwa argumenty.

② Implementacja `BiFunction` wykorzystując lambdę. Do wywołania konstruktora przekazujemy skonkatenowany `String` (dlatego pierwsze dwa generyki definicji `BiFunction` to `String`) i zwracamy `Dog`.

③ Implementacja `BiFunction` wykorzystując method reference, która robi to samo co linijka 2, tyle, że w tym przypadku korzystamy z drugiego konstruktora, który przyjmuje dwa argumenty, a nie jeden tak jak w poprzednim przypadku. Gdyby nie konstruktor oznaczony jako 4, to method reference nie byłoby możliwe.

UnaryOperator

UnaryOperator, to interfejs funkcyjny, który jest specjalnym rodzajem interfejsu `Function`. `UnaryOperator` dziedziczy z `Function`. Określa tę samą metodę `apply()` co `Function`, przy czym **Unary** oznacza, że i typ wejściowy i typ wejściowy tej metody są takie same. Stosujemy go najczęściej, gdy chcemy przekształcić obiekt w obiekt tej samej klasy ale dokonać przy okazji pewnych zmian.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

    // metoda accept jest dziedziczona z Function

    // metoda identity() ma implementację jak pokazano w przykładzie
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleUnaryOperator {

    public static void main(String[] args) {
        UnaryOperator<String> unaryOperator1 = a -> a + "!enriched";
        System.out.println(unaryOperator1.apply("someString")); ①

        UnaryOperator<String> unaryOperator2 = b -> b;
        System.out.println(unaryOperator2.apply("someString")); ②

        UnaryOperator<String> unaryOperator3 = UnaryOperator.identity();
        System.out.println(unaryOperator3.apply("someString")); ③
    }

}
```

① Implementacja `UnaryOperator` wykorzystując lambdę. Typ wejściowy i wyjściowy to `String`.

② Implementacja `UnaryOperator` wykorzystując lambdę. Ponownie typ wejściowy i wyjściowy to `String`, lambda nie zmienia wejściowego `Stringa` tylko przekazuje go w takiej formie w jakiej został on do niej przekazany.

③ Implementacja `UnaryOperator` wykorzystując metodę `identity()`, której implementacja jest pokazana w przykładzie definicji interfejsu `UnaryOperator`.

BinaryOperator

BinaryOperator, to interfejs funkcyjny, który jest specjalnym rodzajem interfejsu `BiFunction`. `BinaryOperator` dziedziczy z `BiFunction`. Określa tę samą metodę `apply()` co `BiFunction`, przy czym **Binary** oznacza, że i typ dwóch parametrów wejściowych i typ wyjściowy tej metody są takie same. Stosujemy go najczęściej, gdy chcemy przekształcić dwa obiekty tej samej klasy w obiekt tej samej klasy ale np, je skleić.

Definicja tego interfejsu wygląda w ten sposób:

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

    // metoda accept jest dziedziczona z BiFunction

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBinaryOperator {

    public static void main(String[] args) {
        BinaryOperator<String> binaryOperator1 = (a, b) -> a + ", " + b;
        System.out.println(binaryOperator1.apply("string1", "string2")); ①

        BinaryOperator<String> binaryOperator2 = ExampleBinaryOperator::concat;
        System.out.println(binaryOperator2.apply("string1", "string2")); ②
    }

    public static String concat(String arg1, String arg2) {
        return arg1 + ", " + arg2;
    }

}
```

① Implementacja `BinaryOperator` wykorzystując lambdę. W tym przykładzie sklejamy ze sobą dwa `Stringi` przy wykorzystaniu przecinka.

② Implementacja `BinaryOperator` wykorzystując method reference. Efekt tego wywołania jest taki sam jak efekt wywołania 1.

Gdzie możemy znaleźć wbudowane interfejsy funkcyjne

Poniżej znajdziesz przykład, w którym pokazane są metody klas Javy, które korzystają z wymienionych interfejsów funkcyjnych.

```
public class ExampleSummingUp {

    public static void main(String[] args) {
        Optional.of("someValue")
            .map(value -> "value" + 192) ①
            .filter(value -> value.length() > 2) ②
            .ifPresent(value -> System.out.println(value + "another")); ③

        List<String> strings = new ArrayList<>();
        strings.removeIf(value -> value.equals("12")); ④

        Map<String, String> map = new HashMap<>();
        map.put("1", "someValue1");
```

```
        map.put("2", "someValue2");
        map.put("3", "someValue3");
        System.out.println(map);
        map.replaceAll((key, value) -> value + "!new"); ⑤
        System.out.println(map);
    }
}
```

① Metoda `Optional.map()` przyjmuje jako argument interface `Function`, który określa logikę zamiany jednego typu danych w drugi.

② Metoda `Optional.filter()` przyjmuje jako argument interface `Predicate`, który zostawia obiekt w `Optional` tylko gdy wynik `Predicate` zwróci `true`.

③ Metoda `Optional.ifPresent()` przyjmuje jako argument interface `Consumer`, który "konsumuje" przekazaną wartość o ile jest ona obecna w `Optional`.

④ Metoda `List.removeIf()` przyjmuje jako argument interface `Predicate`, który definiuje, że element ma być usunięty z `Listy` jeżeli wynik wywołania `Predicate` jest `true`.

⑤ Metoda `Map.replaceAll()` przyjmuje jako argument interface `BiFunction`, który określa logikę jak zamienić wszystkie wpisy w mapie.

Lambdy a obsługa wyjątków

Gdy korzystamy z lambd, najwygodniej jest używać wyjątków **unchecked**. Wynika to z tego, że w lambdzie nie mogą być wywoływane metody, które deklarują wyrzucenie wyjątku checked. Jedynym wyjściem jest albo obsłużenie tego wyjątku wewnątrz takiej metody aby nie deklarowała ona możliwości wyrzucenia takiego wyjątku, albo korzystanie z wyjątków **unchecked**. Demonstruje to poniższy przykład.

```
public class LambdaWithException {

    public static void main(String[] args) {
        //Supplier<String> stringSupplier1 = () -> stringCreatorChecked(); ①
        //Supplier<String> stringSupplier2 = LambdaWithException::stringCreatorChecked; ②

        Supplier<String> stringSupplier3 = () -> wrapper(); ③
        Supplier<String> stringSupplier4 = LambdaWithException::wrapper; ④

        Supplier<String> stringSupplier5 = () -> stringCreatorUnChecked(); ⑤
        Supplier<String> stringSupplier6 = LambdaWithException::stringCreatorUnChecked; ⑥
    }

    private static String wrapper() {
        try {
            return stringCreatorChecked();
        } catch (Exception e) {
            return "nope";
        }
    }

    private static String stringCreatorChecked() throws IOException {
        throw new IOException();
    }
```

```
    private static String stringCreatorUnChecked() throws RuntimeException {
        throw new RuntimeException();
    }
}
```

① Błąd kompilacji, gdyż metoda `stringCreatorChecked()` wyrzuca **checked exception**.

② Błąd kompilacji, gdyż metoda `stringCreatorChecked()` wyrzuca **checked exception**.

③ Kod kompiluje się poprawnie gdyż metoda `wrapper()` wrapuje wyrzucenie wyjątku **checked exception** i obsługuje go wewnątrz.

④ Kod kompiluje się poprawnie gdyż metoda `wrapper()` wrapuje wyrzucenie wyjątku **checked exception** i obsługuje go wewnątrz.

⑤ Kod kompiluje się poprawnie gdyż metoda `stringCreatorUnChecked()` wyrzuca wyjątek **unchecked**, a to już jest dozwolone.

⑥ Kod kompiluje się poprawnie gdyż metoda `stringCreatorUnChecked()` wyrzuca wyjątek **unchecked**, a to już jest dozwolone.

Notatki - Programowanie funkcyjne - Streamy

Spis treści

Od razu powiem, że jest to bardzo ważny materiał, bo na co dzień używa się go dużo. Bardzo nawet.

Dokładnie nazywa się to **Java Stream API** i chodzi w tym o to, żeby móc w sposób funkcyjny operować na sekwencjach danych (w praktyce często są to kolekcje). Całe Java **Stream API** zostało dodane w Javie 8, jak z resztą inne "rzeczy" funkcyjne, o których rozmawialiśmy już wcześniej.

Od razu zastrzegam, że jeżeli ktoś już miał do czynienia z Java IO (operacje na plikach), to tam jest coś takiego jak `InputStream` i `OutputStream`, to to nie to 😄. Tamto dotyczy operacji na streamach bajtów, na ten moment nie wchodzę w ten temat. Tutaj będziemy rozmawiać o operacjach na streamach obiektów w kolekcjach.

Czym jest Stream?

Zanim przejdę do definicji, dla ułatwienia, przedstawmy sobie `Stream` jako linię montażową w fabryce. Produkujemy jakiś produkt, np. samochód, który przechodzi przez linię produkcyjną. Produkowany samochód przechodzi po kolei przez różne stanowiska, na których wykonywane są jakieś operacje. Każde stanowisko (dla uproszczenia) wykonuje daną operację tylko raz (nie tworzy się dla tego samego samochodu dwóch szyb przednich, tworzy się tylko jedną jak skończymy to auto jedzie dalej). Na pierwszym stanowisku tworzony jest szkielet takiego auta, na następnym auto jest malowane na jakiś kolor, na kolejnym sa dokładane koła itp.

Zwróć uwagę, że samochód nie może zostać pomalowany dopóki nie ma stworzonego szkieletu, tzn. kolejne stanowisko nie może wykonać swojej pracy dopóki to poprzednie jej nie skończyło.

Kolejna rzecz jaką trzeba wyróżnić jest to, że elementy nie wracają do tyłu. To znaczy, że pracownik produkujący koła na linii montażowej, jak skończy pracę z jednym konkretnym elementem to zaczyna pracować na kolejnym analogicznym elemencie. (Tak przynajmniej jest na linii montażowej, która jest używana w tym przykładzie ☺)

Należy również zaznaczyć, że ilość tworzonych samochodów jest skończona i zależy od ilości zamówień. Jeżeli mamy 10 zamówień, to utworzymy dokładnie 10 samochodów.

Można również wyobrazić sobie, że na takiej linii montażowej, kolejne etapy wykonują się w sposób sekwencyjny, czyli nie wykonujemy najpierw etapu 4, potem 5, a potem 2, tak jak np. możemy pobrać elementy z listy na kolejnych indeksach. Tutaj rozumiemy to bardziej w formie sekwencji przetwarzania danych.

Widzisz już, że podczas takiego `Streamu`, mogą dziać się różne operacje. Albo używając tutejszego słownictwa, mogą dziać się w rurociągu... (*Stream pipeline*). Ktoś musi zacząć taką pracę na linii montażowej, coś dzieje się po drodze i ktoś musi taką pracę skończyć. *Stream Pipeline* - są to operacje, które mogą zostać na takim `Streamie` danych uruchomione.

`Stream` w Javie można rozumieć jako sekwencję danych. Sekwencja taka ma swoje źródło, operacje, które mogą zostać wykonane w trakcie, oraz określony sposób zakończenia przetwarzania takiej sekwencji.

Należy też zaznaczyć, że w obrębie tego warsztatu będziemy cały czas rozmawiać o **sequential** Streamach. Oznacza to, że dane/wartości/obiekty są przetwarzane przez `Stream` w sekwencji, inaczej mówiąc, oznacza to, że dane/wartości/obiekty są przetwarzane pojedynczo, element po elemencie. Możliwe jest przetwarzanie danych równolegle, zwyczajnie o nie będziemy o tym jeszcze rozmawiać ☺.

Chcę tutaj również zaznaczyć, że nie poruszymy w obrębie tego warsztatu całej możliwej wiedzy dotyczącej `Streamów`. Wiedza z tej tematyki będzie cały czas rozszerzana w kolejnych materiałach, gdy będziemy poznawać kolejne mechanizmy.

Stream pipeline

W *Stream pipeline* można wyróżnić 3 części:

- **Źródło** - z niego zaczyna się `Stream`,
- **Operacje pośrednie** - służą do transformacji jednego `Streama` w drugi. Może być jedna, może być dużo. `Streamy` korzystają z mechanizmu tzw. ***Lazy Evaluation***. ***Lazy Evaluation*** oznacza, że operacje pośrednie nie zostaną wykonane dopóki nie uruchomimy operacji terminującej. (Zapamiętaj stwierdzenie, że gdy w coś jest ***Lazy***, znaczy, najczęściej że zostanie wykonane kiedyś w przyszłości, a nie od razu).
- **Operacja terminująca** - krok, który faktycznie produkuje jakiś rezultat. Do zapamiętania, że `Stream` może być uruchomiony tylko raz. Po wykonaniu operacji terminującej, nie można już z tego samego `Streama` korzystać. Później zostanie pokazany przykład.

Cechy operacji na Streamach

Małe podsumowanie w formie tabelki:

Pytanko	Operacja pośrednia	Operacja terminująca
Czy po jej wywołaniu, Stream nadal może być używany?	Tak	Nie
Czy typ zwracany z tej operacji jest Streamem?	Tak	Nie
Czy ta operacja może wystąpić w Streamie parę razy?	Tak	Nie
Czy jest konieczna, żeby wystąpić?	Nie	Tak
Czy operacja jest wywoływana od razu?	Nie	Tak

Cały czas rozmawiamy tutaj o `Streamach`, które są skończone. Da się też napisać `Streamy`, które będą się wykonywały w nieskończoność, natomiast w praktyce bardzo rzadko (o ile w ogóle) używałem czegoś takiego, zatem nie będę się zagłębiał w temat.

Przejdźmy przez kolejną analogię dla lepszego zrozumienia zagadnienia. W fabryce mamy często nadzorcę/kierownika, czyli osobę, która pilnuje jak wykonywana jest praca na linii montażowej. Możemy sobie wyobrazić, że tworzenie kolejnych kroków `Streama` może wyglądać jak instrukcje, które są wydawane kierownikowi. Instrukcje takie opisują w jaki sposób ma odbywać się praca na linii montażowej i co należy nadzorować. Kierownik taki następnie może podzielić pracę na poszczególnych pracowników i przydzielić im zadania. Natomiast nie mogą oni zacząć wykonywać tych zadań do momentu aż kierownik wyda im polecenie.

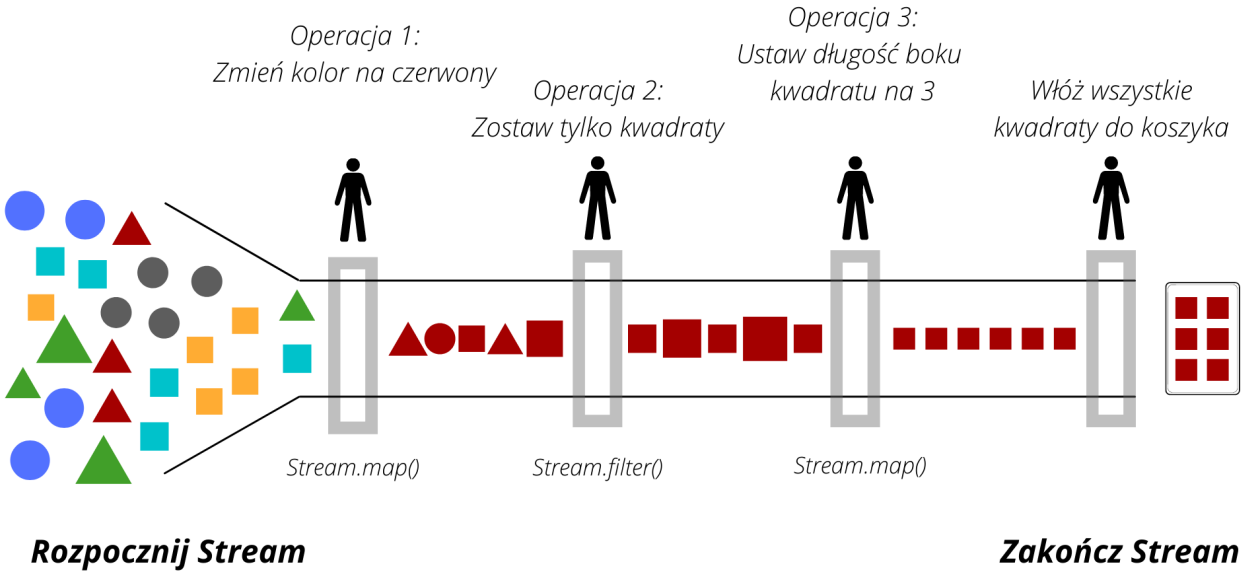
Wyobrażenie sobie takiej analogii jest o tyle istotne, że `Streamy` są ***Lazy***. Oznacza to, że operacje pośrednie nie zostaną wykonane dopóki nie uruchomimy operacji terminującej. Jeżeli wyobrazimy sobie, że to Java jest takim kierownikiem, to możemy też sobie wyobrazić, że jeżeli nie powiemy kierownikowi co jest operacją terminującą to nie zostanie uruchomiony proces produkcyjny. Jednocześnie też kierownik taki ma możliwość zatrzymania całej produkcji, jeżeli po skończeniu drugiego samochodu dostanie on SMS, że jednak tylko 2 samochody są potrzebne. Produkcja reszty byłaby w tym momencie marnotrawstwem zasobów.

Analogia

Podsumowując, Ty jako osoba pisząca kod, możesz się poczuć jak osoba tworząca linię montażową. Ty pisząc decydujesz jak taka linia ma się zaczynać (najczęściej będziemy wychodzić od jakiejś kolekcji obiektów), Ty decydujesz jakie kroki mają się odbyć po drodze, który ma być pierwszy, a który kolejny. Jednocześnie Ty decydujesz, jak taka linia montażowa ma się zakończyć. Możesz nawet określić, że po przeprocesowaniu 4 elementów chcesz taką linię montażową zamknąć, nawet jeżeli są jeszcze na niej elementy do przetworzenia, ale do tego przejdziemy.

Patrząc na grafikę poniżej, wyobraź sobie linię montażową, która przetwarza figury geometryczne, trójkąty, koła i kwadraty w różnych kształtach i kolorach. Rozpoczynamy przetwarzaie takich elementów od pobierania ich pojedyńczo. Następnie możemy przemalować wszystkie figury na kolor czerwony. Następnie zostawiamy na naszej linii montażowej tylko kwadraty, a następnie zmieniamy im wszystkich rozmiar. W ostatnim kroku, na zakończenie przetwarzania naszych elementów wkładamy je

wszystkie do koszyka. To kończy naszą linię montażową 😁.



Obraz 1. Abstrakcyjne spojrzenie na Streamy

Tworzenie Streamów

Zacznijmy od tego, że `Stream`, który utworzymy może być **skończony** albo **nieskończony**. Czyli możemy stworzyć `Stream`, który z założenia ma skończoną ilość elementów, albo `Stream`, który ma nieskończoną ilość elementów.

Finite Streams

Pierwszy z nich może zostać utworzony w sposób pokazany poniżej.

```
Stream<String> emptyStream = Stream.empty();
Stream<String> multipleElements = Stream.of("1", "2", "3", "4");
```

Pierwszy z pokazanych `Streamów` prezentuje pustą sekwencję danych, drugi natomiast sekwencję danych typu `String`.

Mając zdefiniowane `Streamy` jak w kroku powyżej, możemy np. zliczyć ilość elementów w każdym z nich.

```
System.out.println(emptyStream.count()); // 0
System.out.println(multipleElements.count()); // 4
```

Używając `Streamów` należy pamiętać, że `Stream`, który został raz zakończony, nie może zostać ponownie użyty. Pisząc zakończony odwołuję się do operacji terminujących `Stream`. Operacja `count()` jest przykładem operacji terminującej, czyli kończącej działanie `Streama`.

```
Stream<String> emptyStream = Stream.empty();
Stream<String> multipleElements = Stream.of("1", "2", "3", "4");
```

```
System.out.println(emptyStream.count());
System.out.println(multipleElements.count());

System.out.println(emptyStream.count()); ①
System.out.println(multipleElements.count()); ②
```

① Wywołanie tej linijki spowoduje wyrzucenie wyjątku: java.lang.IllegalStateException: stream has already been operated upon or closed.

② Wywołanie tej linijki spowoduje wyrzucenie wyjątku: java.lang.IllegalStateException: stream has already been operated upon or closed. Należy jednak pamiętać o tym, że linijka ta zostanie wywołana dopiero jak zakomentujemy linijkę 1, inaczej błąd zostanie wyrzucony w linijce 1 i linijka 2 nie zostanie wywołana.

Najczęstszym sposobem stworzenia Streama będzie wywołanie metody `.stream()` na jakiejś kolekcji.

```
List<String> stringList = Arrays.asList("1", "2", "3", "4");
Stream<String> stringStream = stringList.stream();
System.out.println(stringStream); ①
```

① Swoją drogą to Stream nie ma "ładnie" drukującej metody, która wydrukuje jego elementy. Wydrukowane zostanie coś w stylu java.util.stream.ReferencePipeline$Head@6a5fc7f7.

Infinite Streams

Drugim rodzajem Streamów są Streamy **infinite**, czyli Streamy nieskończone. Najprostszym sposobem na stworzenie **infinite** Stream jest kod poniżej. Kod ten nie robi nic innego jak generuje w nieskończoność String o wartości 1 i drukuje go na ekranie.

```
Stream<String> generatedWithOne = Stream.generate(() -> "1");
generatedWithOne.forEach(a -> System.out.println(a));
```

Możemy urozmaicić ten przykład w taki sposób, żeby zawsze generować wartości losowe z przedziału od 0.0 do 1.0.

```
Stream<Double> randomStream = Stream.generate(() -> Math.random());
randomStream.forEach(System.out::println);
```

Innym ciekawym sposobem na stworzenie **infinite** Stream jest wykorzystanie metody `iterate()`, która pozwala nam stworzyć Stream wartości zaczynając od jakiejś wartości i modyfikując poprzednią wartość zgodnie z podanym warunkiem. Przykład poniżej.

```
Stream<Integer> wordsStream = Stream.iterate(1, previous -> previous + 1);
wordsStream.forEach(word -> System.out.println(word));
```

Powyższy fragment kodu stworzy Stream, który będzie zawierał wartości zaczynające się od 1 i zwiększające się o 1 aż do nieskończoności. Bo w końcu nigdzie nie napisaliśmy żadnego warunku kiedy

taki Stream ma się zatrzymać.

Operacje terminujące

Operacja terminująca jest krokiem, który faktycznie pozwala wyprodukować jakiś rezultat. Można wywołać operację terminującą, bez napisania jakiejkolwiek operacji pośredniej. Odwrotnie nie można. Oczywiście można napisać taki Stream, gdzie będziemy mieli operacje pośrednie, ale nie będziemy mieli terminującej i kod się skompiluje, ale nie ma to sensu, bo taki Stream się nie uruchomi. Czyli, aby Stream się w ogóle uruchomił, musimy zdefiniować operację terminującą.

Także operacja terminująca musi być zawsze. Operacja terminująca powoduje też, że dany Stream nie może być wykorzystany ponownie. Czyli nie możemy wywołać dwukrotnie operacji terminującej na tym samym Streamie.

Poniżej umieszczam definicję klasy Dog, która będzie używana w przykładach.

Klasa Dog

```
class Dog {

    private String name;

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

    public String getName() {
        return name;
    }

    public static Dog of(String name) {
        return new Dog(name);
    }

    @Override
    public String toString() {
        return "Dog{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

count

Poniżej znajdziesz przykład wykorzystania operacji terminującej count(). Operacja ta jest wykorzystywana do zliczenia ilości elementów w Stream. Oczywiście zastosowanie jej ma sens gdy wykorzystujemy ją ze Streamami skończonymi. W przypadku Streamów nieskończonych program będzie działał w nieskończoność. Jak zastanawiasz się czemu to spróbuj policzyć od 1 do nieskończoności i daj nam znać jak skończysz 😁.

```
private static void countExmple() {
    List<String> fruits = Arrays.asList("banana", "mango", "raspberry", "apple", "blueberry");
```

```
    Stream<String> fruitsStream = fruits.stream();
    System.out.println(fruitsStream.count());
}
```

findFirst i findAny

Poniżej znajdziesz przykład wykorzystania operacji terminujących `findFirst()` oraz `findAny()`. Operacja `findFirst()` jest wykorzystywana do znalezienia pierwszego elementu w `Stream`. Operacja `findAny()` jest wykorzystywana do znalezienia jakiegokolwiek elementu w `Stream`.

```
private static void findExmple() {
    List<String> fruits = Arrays.asList("banana", "mango", "raspberry", "apple", "blueberry");
    Stream<String> fruitsStream2 = fruits.stream();
    Optional<String> firstFruit = fruitsStream2.findFirst(); ①
    String optionalFruit = firstFruit.orElse("non existing fruit"); ②
    System.out.println(optionalFruit);

    List<String> fruits2 = Arrays.asList("banana", "mango", "banana", "apple", "banana");
    Optional<String> any = fruits2.stream() ③
        .findAny();
    System.out.println(any);

    List<String> fruits3 = Arrays.asList("raspberry", "apple", "blueberry", "banana", "mango");
    Optional<String> first = fruits3.stream()
        .findFirst();
    System.out.println(first);
}
```

① Wynikiem wywołania `findFirst()` jest `Optional`, gdyż nie mamy pewności, że taki element istnieje.

② Możemy wyjść z `Optionala` albo wyrzucając wyjątek, albo zapewniając wartość domyślną w przypadku pustego `Optionala`.

③ Wynikiem wywołania `findAny()` jest `Optional`, gdyż nie mamy pewności, że taki element istnieje.

min i max

Poniżej znajdziesz przykład wykorzystania operacji terminujących `min()` oraz `max()`. Operacja `min()` jest wykorzystywana do znalezienia najmniejszego elementu w `Stream` przy wykorzystaniu określonego `Comparatora`. Analogicznie działa operacja `max()` do znalezienia największego elementu.

```
private static void minMaxExample() {
    List<String> fruits = Arrays.asList("banana", "mango", "raspberry", "apple", "blueberry");
    Optional<String> min = fruits.stream()
        .min(Comparator.naturalOrder()); ①
    System.out.println(min);

    List<Dog> dogs = Arrays.asList(Dog.of("bob"), Dog.of("apple"), Dog.of("banana"), Dog.of("mango"));
    Optional<Dog> max = dogs.stream()
        .max(Comparator.comparing(Dog::getName)); ②
    System.out.println(max);
}
```

① Zarówno metoda `min()` jak i `max()` wymuszają podanie implementacji interfejsu `Comparator`.

② Zarówno metoda `min()` jak i `max()` wymuszają podanie implementacji interfejsu `Comparator`.

allMatch, anyMatch i noneMatch

Poniżej znajdziesz przykład wykorzystania operacji terminujących `allMatch()`, `anyMatch()` oraz `noneMatch()`. Operacje te służą do sprawdzenia, czy wszystkie/którykolwiek/żaden element `Streama` spełnia przekazany `Predicate`.

```
private static void allMatch_anyMatch_noneMatch() {
    List<String> fruits1 = Arrays.asList("banana", "mango", "raspberry", "apple", "blueberry");
    boolean any = fruits1.stream()
        .anyMatch(fruit -> fruit.contains("ban")); ①
    System.out.println(any);

    List<String> fruits2 = Arrays.asList("banana", "mango", "raspberry", "apple", "blueberry");
    boolean all = fruits2.stream()
        .allMatch(fruit -> fruit.length() > 6); ②
    System.out.println(all);

    List<Dog> dogs = Arrays.asList(Dog.of("bob"), Dog.of("apple"), Dog.of("banana"), Dog.of("mango"));
    boolean none = dogs.stream()
        .noneMatch(dog -> dog.getName().length() == 0); ③
    System.out.println(none);
}
```

① Sprawdź czy jakikolwiek element w `Stream` spełnia podany `Predicate`.

② Sprawdź czy wszystkie elementy w `Stream` spełniają podany `Predicate`.

③ Sprawdź czy żaden element w `Stream` nie spełnia podanego `Predicate`.

forEach

Poniżej znajdziesz przykład wykorzystania operacji terminującej `forEach()`. Operacja `forEach()` kończy działanie `Streama` (czyli jest terminująca). Może być używana zamiast stosowania pętli `foreach`. Po jej użyciu, nie jest możliwe ponowne użycie `Streama`, gdyż `forEach()` jest terminujące.

```
private static void forEach() {
    List<Dog> dogs1 = Arrays.asList(Dog.of("bob"), Dog.of("apple"), Dog.of("banana"), Dog.of("mango"));
    dogs1
        .forEach(dog -> System.out.println(dog + "123"));

    List<Dog> dogs2 = Arrays.asList(Dog.of("bob"), Dog.of("apple"), Dog.of("banana"), Dog.of("mango"));
    dogs2.stream() ①
        .forEach(dog -> System.out.println(dog + "345"));

    List<Dog> dogs3 = Arrays.asList(Dog.of("bob"), Dog.of("apple"), Dog.of("banana"), Dog.of("mango"));
    Stream<Dog> stream = dogs3.stream();
    // for (String s : stream) {} ②
}
```

① Intellij nam tutaj podpowie, że niepotrzebnie dodajemy `.stream()`.

② Ten fragment kodu się nie kompiluje, `Stream` nie może być używany do iterowania w pętli `foreach`.

Jeżeli ktoś chce poczytać, czy lepiej jest używać pętli "enhanced" `for`, czy operacji `forEach()` w odniesieniu do kolekcji, to zostawiam ten wątek ze Stackoverflow.

reduce

Poniżej znajdziesz przykład wykorzystania operacji terminującej `reduce()`. Operacja `reduce()` jest bardzo dobrym przykładem redukcji. Jest to taki rodzaj operacji terminującej, który "skleja", (z angielska "combines") rezultat `Streama` do jednego obiektu, albo prymitywa. Rezultatem może też być jakaś kolekcja. Bo przecież w Javie wszystko (oprócz prymitywów) jest obiektem, więc kolekcje też. Operacja `reduce()` jest przykładem redukcji `Streama` gdyż służy ona do zamiany `Streama` w jeden wynik końcowy - np. String. Kolokwialnie rzecz nazywając, operacja `reduce()` służy do sklejenia `Streama` do jednej wartości końcowej. Możemy przy tym również podać wartość początkową.

Najpierw pokażmy przykład bez wykorzystania podejścia funkcyjnego.

```
private static void reduceNonFunctional() {
    String[] someChars = new String[]{"z", "a", "j", "a", "v", "k", "a", " ", "j", "a", "v", "k", "a"};
    String concat = "";
    for (String someChar : someChars) {
        concat = concat + someChar;
    }
    System.out.println("1. " + concat);
}
```

A teraz analogiczny przykład z wykorzystaniem podejścia funkcyjnego.

```
private static void reduceFunctional() {
    String reduced = Stream.of(someChars)
        .reduce("initial", String::concat);
        //.reduce("", (l, r) -> l + r); ①
    System.out.println("2. " + reduced);

    // inny przykład
    List<Integer> integers = List.of(1, 3, 4, 5);
    Integer weirdSum = integers.stream()
        .reduce(0, (a, b) -> a + b + 1); ②
    System.out.println(weirdSum);
}
```

① Można to zapisać również w ten sposób. Operacja `reduce()` przyjmuje jako argument `BinaryOperator`. Jako pierwszy argument wywołania możemy natomiast podać wartość początkową, od której zaczniemy sklejanie wartości `Streama`.

② Tutaj natomiast "redukujemy" `Stream` zaczynając od `0` i dodając do siebie kolejne wartości, przy czym przy każdym dodawaniu dodajemy też `1`.

Jeżeli natomiast nie podamy wartości początkowej, to wynikiem takiego "sklejania" jest `Optional`. Wynika to z tego, że jeżeli nie mamy wartości początkowej i przekażemy pusty `Stream` to wartość "sklejona" jest `Optional`. Jeżeli podamy wartość początkową i przekażemy pusty `Stream`, to wynik takiego sklejenia będzie tą wartością początkową.

```
private void noInitValue() {
    BinaryOperator<Integer> concatenator = (a, b) -> a + b;

    Stream<Integer> emptyStream = Stream.empty();
    Stream<Integer> oneElementStream = Stream.of(2);
    Stream<Integer> multipleElementsStream = Stream.of(2, 6, 3);

    Optional<Integer> reduce1 = emptyStream.reduce(concatenator);
    Optional<Integer> reduce2 = oneElementStream.reduce(concatenator);
    Optional<Integer> reduce3 = multipleElementsStream.reduce(concatenator);

    System.out.println(reduce1);
    System.out.println(reduce2);
    System.out.println(reduce3);
}
```

collect

Poniżej znajdziesz przykład wykorzystania operacji terminującej `collect()`. Operacja `collect()` najczęściej służy do przetransformowania `Streama` do postaci kolekcji (np. `List`, `Set`, `Map` itp.), ale możemy ją też wykorzystać aby przetransformować `Stream` do wartości końcowej w postaci np. String. Z operacją `collect()` bardzo blisko jest związana klasa `Collectors`, której zastosowania pokażemy teraz. Natomiast do bardziej skomplikowanych przypadków przejdziemy później.

W każdym z poniższych przypadków wykorzystamy listę `chars`, która jest pokazana poniżej.

```
List<String> chars = List.of("z", "a", "j", "a", "v", "k", "a", " ", "j", "a", "v", "k", "a");
```

toSet

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `Set`. Zwróć uwagę, że nie wiemy tutaj jaka konkretnie implementacja interfejsu `Set` to jest. Wiemy tylko, że będzie to `Set`. Nawet dokumentacja wspomina, że nie mamy gwarancji jakiego rodzaju `Set` to będzie. Nie możemy się zatem na tym opierać.

```
private void collectExample(final List<String> chars) {
    Set<String> collect1 = chars.stream()
        .collect(Collectors.toSet());
    System.out.println(collect1);
}
```

toList

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `List`. Zwróć uwagę, że nie wiemy tutaj jaka konkretnie implementacja interfejsu `List` to jest. Wiemy tylko, że będzie to `List`. Nawet dokumentacja wspomina, że nie mamy gwarancji jakiego rodzaju `List` to będzie. Nie możemy się zatem na tym opierać.

```
private void collectExample(final List<String> chars) {
    List<String> collect2 = chars.stream()
```

```
        .collect(Collectors.toList());
    System.out.println(collect2);
}
```

toSet - LinkedHashSet

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `Set`. Zwróć uwagę, że określamy tutaj już konkretnie, że chcemy aby wynikowy `Set` to był `LinkedHashSet`.

```
private void collectExample(final List<String> chars) {
    Set<String> collect3 = chars.stream()
        .collect(Collectors.toCollection(() -> new LinkedHashSet<>()));
    System.out.println(collect3);
}
```

toSet - TreeSet

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `Set`. Określamy tutaj, że końcowy `Set` to ma być `TreeSet`, natomiast wykorzystujemy tym razem **method reference**.

```
private void collectExample(final List<String> chars) {
    Set<String> collect4 = chars.stream()
        .collect(Collectors.toCollection(TreeSet::new));
    System.out.println(collect4);
}
```

joining

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `String`. W tym celu wykorzystywany jest kolektor `Collectors.joining()`. Możemy również podać, że podane elementy mają być w wynikowym `Stringu` oddzielone jakimiś znakami, czyli możemy podać "łącznik". Przykład taki zostanie pokazany w następnej kolejności.

```
private void collectExample(final List<String> chars) {
    String collect5 = chars.stream()
        .collect(Collectors.joining());
    System.out.println(collect5);
}
```

joining z łącznikiem

Przykład poniżej terminuje `Stream` i tworzy na jego podstawie `String`. W tym celu wykorzystywany jest kolektor `Collectors.joining()`. W tym przykładzie określamy, że podane elementy mają być w wynikowym `Stringu` oddzielone "łącznikiem", którym w tym przypadku jest znak `;`.

```
private void collectExample(final List<String> chars) {
    String collect6 = chars.stream()
        .collect(Collectors.joining(";"));
```

```
    System.out.println(collect6);
}
```

Bez klasy Collectors

Możemy również napisać swój własny `Collector`, albo inaczej mówiąc, wywołać operację `collect()` bez wykorzystywania klasy `Collectors`. Przykład taki jest pokazany poniżej. W tym celu musimy przekazać 3 argumenty do wywołania metody `collect()` (odniesienia wymienionych nazw do kodu znajdziesz poniżej przykładu).

- **supplier** - w tym argumencie określamy implementację kolekcji, która nas interesuje.
- **accumulator** - w tym argumencie określamy co ma się stać z pojedynczym elementem w `Stream` w odniesieniu do istniejącej już kolekcji.
- **combiner** - ten argument będzie tak na prawdę potrzebny gdy zaczniemy przetwarzać `Streamy` równolegle, ale na razie nie poruszamy tej tematyki. Jeżeli będziemy przetwarzać `Streamy` równolegle to pod spodem może wystąpić sytuacja, gdzie zostanie utworzonych kilka mniejszych kolekcji i będziemy musieli określić sposób aby je ze sobą połączyć. Dlatego właśnie ten krok łączy ze sobą dwa `Sety`.

```
private void collectExample(final List<String> chars) {
    Set<String> collect = chars.stream()
        .collect(
            () -> new TreeSet<>(), ①
            (existingSet, nextElement) -> existingSet.add(nextElement), ②
            (leftColl, rightColl) -> leftColl.addAll(rightColl) ③
        );
        //.collect( ④
        //  TreeSet::new,
        //  TreeSet::add,
        //  TreeSet::addAll
        //);
    System.out.println(collect);
}
```

① **supplier**

② **accumulator**

③ **combiner**

④ Taką zamianę na **method reference** proponuje nam Intellij.

Podsumowanie

Poniżej znajdziesz tabelkę podsumowującą poruszone operacje terminujące.

Metoda	Typ zwracany	Redukcja
count()	long	Tak
findFirst()	Optional<T>	Nie
findAny()	Optional<T>	Nie

Metoda	Typ zwracany	Redukcja
min()	Optional<T>	Tak
max()	Optional<T>	Tak
allMatch()	boolean	Nie
anyMatch()	boolean	Nie
noneMatch()	boolean	Nie
forEach()	void	Nie
collect()	to zależy	Tak
reduce()	to zależy	Tak

Operacje pośrednie

Wynikiem wykonania operacji pośredniej jest kolejny stream. W przypadku operacji terminujących wyniki tych operacji zostały podane w tabelce. Jeżeli natomiast wykonamy operację pośrednią, to dostaniemy znowu `Stream`. Można później wykonać kolejną operację pośrednią i znowu jej wynikiem będzie `Stream`. I tak dalej i tak dalej. Trzeba pamiętać żeby na końcu była operacja terminująca, bo inaczej `Stream` się nie uruchomi. `Streamy` są **lazy**, pamiętasz? Każda operacja pośrednia skupia się na wykonaniu swojej czynności (porównując operacje do linii montażowej). Pracownik, który składa koła nie idzie w tym samym czasie malować, robi jedną rzecz naraz… składa koła.

Poniżej znajdziesz wypisane poruszone w materiałach operacje pośrednie razem z przykładami. W przykładach będziemy wykorzystywać dwie klasy, `City` oraz `Person`. Ich definicje znajdziesz poniżej.

Klasa Person

```
class Person {

    private final String name;

    private final City city;

    public Person(final String name, final City city) {
        this.name = name;
        this.city = city;
    }

    public String getName() {
        return name;
    }

    public City getCity() {
        return city;
    }

    @Override
    public String toString() {
        return "Person{" +
            "name='" + name + '\'' +
            ", city=" + city +
            '}';
```

```
    }
}
```

Klasa City

```
class City {

    private final String name;

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

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "City{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

filter

Operacja `filter()` służy do tego aby odsiać dane, które nie spełniają warunku, który jest określony jako implementacja interfejsu `Predicate`. Inaczej mówiąc, jeżeli `Predicate` jest spełnione, to elementy pozostaną w `Streamie`, jeżeli nie jest to zostaną odrzucone. Operacja `filter()` przyjmuje jako argument interface funkcyjny `Predicate`, a jej sygnatura wygląda w ten sposób:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Do implementacji wspomnianego interfejsu funkcyjnego będziemy stosować lambdę, poniżej przykłady wykorzystania.

```
private static void filter() {
    String someString = Optional.of("someValue")
        .filter(value -> value.startsWith("some")) ①
        .orElseThrow(() -> new RuntimeException());
}
```

① W `Optional` wartość będzie nadal dostępna (czyli `Optional` nie będzie empty) tylko jeżeli ta wartość `String` zaczyna się od `"some"`.

```
private static void filter() {
    Stream<String> someStream = Stream.of("val1", "val2", "val3", "val4");
    List<String> collect = someStream
        .filter(value -> value.contains("3") || value.contains("2")) ①
        .collect(Collectors.toList());
    System.out.println(collect);
```

```
}
```

① Przykład pokazuje `Stream Stringów`, na którym próbujemy zostawić tylko wartości, które zawierają w sobie `"2"` lub `"3"`, zatem na ekranie zostanie wydrukowane tylko `val2` i `val3`.

```
private static void filter() {
    Stream<String> someStream = Stream.of("val1", "val2", "val3", "val4");
    List<String> collect = someStream
        .filter(value -> value.equals("123")) ①
        .collect(Collectors.toList());
    System.out.println(collect);
}
```

① Przykład pokazuje `Stream Stringów`, na którym próbujemy zostawić tylko wartości, które są równe `"123"`, zatem na ekranie zostanie wydrukowana pusta lista, gdyż żadna z wartości ze `Stream` nie przejdzie przez operację `filter()` dalej.

```
private static void filter() {
    Stream<String> someStream = Stream.of("abc1", "cde", "efg", "ghi");
    List<String> collect = someStream
        .filter(value -> !value.contains("a")) ①
        .collect(Collectors.toList());
    System.out.println(collect);
}
```

① Przykład pokazuje `Stream Stringów`, na którym próbujemy zostawić tylko wartości, które **nie** są równe `"a"`, na ekranie zostaną wydrukowane tylko wartości, które nie zawierają w sobie litery `a`.

map

Map jest wykorzystywany wtedy gdy chcemy zmienić typ danych, na którym operujemy w naszym strumieniu danych. Czyli jeżeli `Stream` operuje na Samochodach i chcemy to zmienić, aby od następnego kroku operował na Kierownicach z tych samochodów to wykorzystamy operację `map()`. Operacja `map()` przyjmuje jako argument interfejs funkcyjny `Function`, a jej sygnatura wygląda w ten sposób:

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

Do implementacji wspomnianego interfejsu funkcyjnego będziemy stosować lambdę, poniżej przykłady wykorzystania.

```
private static void map() {
    List<Person> people = Arrays.asList(
        new Person("Roman", new City("Warszawa")),
        new Person("Agnieszka", new City("Gdańsk")),
        new Person("Adam", new City("Łódź")),
        new Person("Zbyszek", new City("Wrocław")),
        new Person("Stefania", new City("Gdańsk")),
        new Person("Gabriela", new City("Łódź"))
    );

    Integer sum = people.stream()
```

```
        .map(person -> person.getCity()) ①
        .map(city -> city.getName()) ②
        .map(name -> name.length()) ③
        .reduce(0, (a, b) -> a + b); ④
    System.out.println(sum);
}
```

① W pierwszej kolejności zmieniamy typ danych w `Stream` z `Person` na `City`. Czyli od tego momentu operujemy na miastach każdej z osób w `Streamie`, a nie na osobach.

② Następnie ze `Stream<City>` wyjmujemy nazwę tego miasta w postaci `String`, czyli zamiast `Stream<City>` będziemy teraz operować na `Stream<String>`.

③ Dla każdej z nazw tych miast przemapowujemy nazwę miasta na jego długość, czyli zamiast `Stream<String>` będziemy teraz operować na `Stream<Integer>`.

④ Dokonujemy operacji terminującej, w której zaczynając od `0` sumujemy długości nazw miast, które były zawarte w `Stream`. Wynikiem jest suma długości nazw wszystkich miast, które początkowo były zawarte w liście początkowej.

Poniżej przykład, który spowoduje błąd kompilacji. W przykładzie tym staramy się stworzyć osoby, które będą automatycznie miały przypisane miasta. Jednocześnie też natomiast chcemy nadać automatyczną numerację tym osobom.

```
private static void map() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Gdańsk", "Łódź");
    int counter = 0; ①
    cities.stream()
        .map(city -> new City(city))
        .map(city -> new Person("person" + ++counter, city)); ②
}
```

① Zmienna zdefiniowana poza lambdą, którą chcemy wykorzystać w środku lambdy musi być **final** lub **effectively final**. Zmienna jest **effectively final** jeżeli możemy dopisać do niej słówko final i nie dostaniemy błędu kompilacji, w tym przypadku tak nie jest, bo `counter` jest zmieniane w obrębie ciała lambdy.

② W tym miejscu we fragmencie `++counter` otrzymamy błąd kompilacji, gdyż lambda wymusza, aby zmienne (zdefiniowane poza lambdą), które są w niej używane były **effectively final**.

W przykładzie powyżej Intellij sam podpowiada, żeby zastąpić zmienną `int` typem `AtomicInteger`. Na ten moment wystarczy nam informacja, że `AtomicInteger` jest wrapperem (opakowaniem), na `Integer`, który pozwoli nam zmieniać wartości `Integera` bez zmiany referencji do obiektu. Czyli w przykładzie poniżej, nie ulegnie zmianie referencja `counter`, będzie ona cały czas wskazywała na ten sam obiekt, czyli będzie **effectively final**. Jednocześnie jednak możemy zmieniać wartość `Integer`, który jest opakowany w `AtomicInteger` przy wykorzystaniu metody `incrementAndGet()`.

```
private static void map() {
    AtomicInteger counter = new AtomicInteger(0);
    List<Person> collected = cities.stream()
        .map(city -> new City(city))
        .map(city -> new Person("person" + counter.incrementAndGet(), city))
        .collect(Collectors.toList());
    System.out.println(collected);
```

```
}
```

flatMap

Najczęściej używamy tej operacji, gdy chcemy "spłaszczyć" strukturę, czyli np. gdy mamy listę list. Czyli taka lista dwuwymiarowa (analogia do tablicy dwuwymiarowej). Podobne zastosowanie miało `flatMap()` w `Optional`, w tamtym przypadku gdy mieliśmy zagnieżdżenie - `Optional` w `Optional` i chcieliśmy to "spłaszczyć" to wykorzystywana była operacja `flatMap()`. Tutaj działa to w ten sam sposób, tylko, że możemy mieć np. `Stream<Stream<String>>` Operacja `flatMap()` przyjmuje jako argument interfejs funkcyjny `Function`, a jej sygnatura wygląda w ten sposób:

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

Do implementacji wspomnianego interfejsu funkcyjnego będziemy stosować lambdę, poniżej przykłady wykorzystania.

```
private static void flatMap() {
    List<String> cities1 = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Gdańsk", "Łódź");
    List<String> cities2 = Arrays.asList("Białystok", "Szczecin", "Łódź", "Zakopane", "Gdańsk", "Łódź");
    List<String> cities3 = Arrays.asList("Warszawa", "Lublin", "Wrocław", "Wrocław", "Kraków", "Poznań");

    Stream<List<String>> stream = Stream.of(cities1, cities2, cities3);
    var collected = stream
        .flatMap(a -> a.stream()) ①
        .collect(Collectors.toList());
    System.out.println(collected);
}
```

① Ważne jest to, że jeżeli chcemy "spłaszczyć" taką strukturę, to we `flatMap()` musimy doprowadzić do sytuacji, gdzie będziemy mieli `Stream<Stream<String>>`, stąd wywołanie `.stream()`.

peek

Dochodzimy nareszcie do operacji, która pozwoli nam podejrzeć przebieg wykonania `Streama`. Jest ona bardzo przydatna gdy chcemy się zorientować co się dzieje w środku naszej linii produkcyjnej. Czyli po każdej wykonanej operacji możemy dodać `peek()` żeby zerknąć na aktualny stan `Streama`. Operacja `peek()` przyjmuje jako argument interfejs funkcyjny `Consumer`, a jej sygnatura wygląda w ten sposób:

```
Stream<T> peek(Consumer<? super T> action)
```

Do implementacji wspomnianego interfejsu funkcyjnego będziemy stosować lambdę, poniżej przykłady wykorzystania.

```
private static void peek() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Gdańsk", "Łódź");
    AtomicInteger counter = new AtomicInteger(0);
    List<Person> collected = cities.stream()
        .peek(value -> System.out.println("Step1: " + value))
```

```
        .map(city -> new City(city)) ①
        .peek(value -> System.out.println("Step2: " + value))
        .map(city -> new Person("person" + counter.incrementAndGet(), city)) ②
        .peek(value -> System.out.println("Step3: " + value))
        .collect(Collectors.toList());
}
```

Operacja `peek()` pozwala nam zauważyć jaka jest kolejność powyżej wykonywanych operacji.

```
Step1: Warszawa
Step2: City{name='Warszawa'}
Step3: Person{name='person1', city=City{name='Warszawa'}}
Step1: Gdańsk
Step2: City{name='Gdańsk'}
Step3: Person{name='person2', city=City{name='Gdańsk'}}
Step1: Łódź
Step2: City{name='Łódź'}
Step3: Person{name='person3', city=City{name='Łódź'}}
Step1: Wrocław
Step2: City{name='Wrocław'}
Step3: Person{name='person4', city=City{name='Wrocław'}}
Step1: Gdańsk
Step2: City{name='Gdańsk'}
Step3: Person{name='person5', city=City{name='Gdańsk'}}
Step1: Łódź
Step2: City{name='Łódź'}
Step3: Person{name='person6', city=City{name='Łódź'}}
```

Zwróć uwagę, że każda z wartości wejściowych z listy `cities` jest przetwarzana sekwencyjnie. Czyli krok oznaczony numerem 1 i 2 nie czeka na wszystkie elementy i dopiero przepuszcza je dalej, tylko wszystkie elementy są przetwarzane w sekwencji. Czyli `Warszawa`, krok 1, 2, 3, następnie `Gdańsk`, krok 1, 2, 3 i tak dalej.

distinct

Operacja `distinct()` służy do usuwania wartości zduplikowanych. Jak możesz się domyślić, żeby w poprawny sposób Java mogła odróżnić duplikaty, musimy zadeklarować metodę `equals()`. Operacja `distinct()` nie przyjmuje żadnego argumentu, a jej sygnatura wygląda w ten sposób:

```
Stream<T> distinct()
```

Poniżej przykłady wykorzystania.

```
private static void distinct() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Gdańsk", "Łódź");
    String collected = cities.stream()
        .distinct()
        .collect(Collectors.joining(","));
    System.out.println(collected);
}
```

W tym przykładzie odsiewamy duplikaty, a następnie łączymy miasta jako wynik oddzielając nazwy przy pomocy przecinka.

```
private static void distinct() {
    List<City> cities = Arrays.asList(
        new City("Warszawa"),
        new City("Gdańsk"),
        new City("Łódź"),
        new City("Wrocław"),
        new City("Gdańsk"),
        new City("Łódź"));

    List<City> collected = cities.stream()
        .distinct()
        .collect(Collectors.toList());
    System.out.println(collected);
}
```

W tym przykładzie odsiewamy duplikaty z listy, aby w następnej kolejności uzyskać listę bez duplikatów.

limit

Operacja `limit()` służy do tego aby ograniczyć ilość elementów w `Streamie`. Operacja ta jest jak SMS, który przyszedł do kierownika, że po 4 elementach kończymy. Operacja `limit()` przyjmuje jako argument ilość elementów do jakiej ma zostać ograniczony `Stream`, a jej sygnatura wygląda w ten sposób:

```
Stream<T> limit(int maxSize)
```

Poniżej przykłady wykorzystania.

```
private static void limit() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Zakopane", "Szczecin");
    cities.stream()
        .peek(value -> System.out.println("Step1: " + value))
        .map(value -> value.length())
        .peek(value -> System.out.println("Step2: " + value))
        .limit(4)
        .peek(value -> System.out.println("Step3: " + value))
        .forEach(System.out::println);
    System.out.println();
}
```

W przykładzie powyżej ograniczamy ilość wydrukowanych elementów do 4. Jeżeli teraz uruchomimy ten kod to na ekranie zobaczymy poniższy wydruk. Widać, że ograniczenie ilości elementów do 4 powoduje, że kolejne miasta (Zakopane, Szczecin) nie zostały nawet wpuszczone na linię produkcyjną.

```
Step1: Warszawa
Step2: 8
Step3: 8
8
Step1: Gdańsk
```

```
Step2: 6
Step3: 6
6
Step1: Łódź
Step2: 4
Step3: 4
4
Step1: Wrocław
Step2: 7
Step3: 7
7
```

skip

Operacja `skip()` służy do pominięcia określonej ilości elementów z początku `Streama`. Możemy ją sobie wyobrazić jako wyrzucenie kilku pierwszych wyprodukowanych samochodów ze względu na problemy z początku produkcji. Operacja `skip()` przyjmuje argument określający ilość elementów do pominięcia, a jej sygnatura wygląda w ten sposób:

```
Stream<T> skip(int n)
```

Poniżej przykłady wykorzystania.

```
private static void skip() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Zakopane", "Szczecin");
    cities.stream()
        .peek(value -> System.out.println("Step1: " + value))
        .map(value -> value.length())
        .peek(value -> System.out.println("Step2: " + value))
        .skip(4)
        .peek(value -> System.out.println("Step3: " + value))
        .forEach(System.out::println);
    System.out.println();
}
```

W przykładzie powyżej pomijamy pierwsze 4 elementy. Jeżeli teraz uruchomimy ten kod to na ekranie zobaczymy poniższy wydruk. Widać, że pominięcie pierwszych 4 elementów powoduje, że pierwsze miasta (Warszawa, Gdańsk, Łódź, Wrocław) zostały wpuszczone na linię produkcyjną, ale w trakcie zostały z niej usunięte. Nie widać dla nich kroku `Step3`.

```
Step1: Warszawa
Step2: 8
Step1: Gdańsk
Step2: 6
Step1: Łódź
Step2: 4
Step1: Wrocław
Step2: 7
Step1: Zakopane
Step2: 8
Step3: 8
8
```

```
Step1: Szczecin
Step2: 8
Step3: 8
8
```

sorted

Operacja `sorted()` jak sama nazwa wskazuje, służy do sortowania elementów, które mamy dostępne w `Stream`. Możemy przekazać do niej `Comparator` lub wywołać operację `sorted()` na `Streamie` obiektów, które implementują interface `Comparable`. Jeżeli nie zrobimy przynajmniej jednego z wymienionych, otrzymamy błąd w trakcie działania programu. Operacja `sorted()` może albo nie przyjąć żadnego argumentu, albo przyjąć `Comparator`. Jej sygnatura wygląda w ten sposób:

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

Poniżej przykłady wykorzystania.

```
private static void sorted() {
    List<City> cities = Arrays.asList(
        new City("Warszawa"),
        new City("Gdańsk"),
        new City("Łódź"),
        new City("Wrocław"),
        new City("Gdańsk"),
        new City("Łódź")
    );
    cities.sort(Comparator.comparing(City::getName)); ①

    cities.stream()
        .sorted(Comparator.comparing(element -> element.getName().length())) ②
        .forEach(System.out::println);

    // cities.stream()
    //     .sorted() ③
    //     .forEach(System.out::println);
}
```

① W podejściu, gdzie nie stosujemy `Stream`, tylko wywołujemy metodę `sort()` z interfejsu `List` musimy przekazać w wywołaniu `Comparator`.

② Wykorzystując operację `sorted()` dostępną na `Stream`, możemy przekazać `Comparator` w wywołaniu, albo klasa, na której operujemy w `Stream` musi implementować interface `Comparable`.

③ W przykładach klasa `City` nie implementuje interfejsu `Comparable`, zatem ta linijka spowoduje błąd w trakcie działania programu.

Operacja `sorted()` musi poczekać w swoim kroku na wszystkie elementy, które są zawarte w `Stream`. Nie może przecież posortować wszystkich elementów nie gromadząc ich w jednym miejscu. Przykład z tym związany zostanie pokazany później.

Podsumowanie

Operacja	Co to robi?
filter()	Zostawia tylko te wartości, dla których predykat przekazany w argumencie zwraca `true`
map()	Zmienia typ danych w Streamie na inny. Formalnie może też "zmienić" na ten sam. Ten "map" służy do transformowania elementów (mapowania elementów), nie mylić z kolekcją "Map"
flatMap()	Robi to samo co `map()`, ale pozbywa się zagnieżdżenia, tzn. jak mamy `Stream` w Streamie i chcemy się pozbyć tego zagnieżdżenia to stosujemy `flatMap()`. Analogiczne metody `map()` i `flatMap()` są dostępne w `Optional`
peek()	Zerknij - służy do podglądu tego co jest obecnie w Streamie. Nie zaleca się tutaj modyfikowania danych. Nie mylić z `peek()` w kolejkach
distinct()	Zwraca Stream z usuniętymi duplikatami. A od razu zapytam, skąd Java wie jak odróżnić duplikaty?
limit()	Ogranicza Stream do podanej ilości elementów
skip()	Pomija podaną początkową ilość elementów w Streamie
sorted()	A to jak myślisz co robi? No sortuje 😄

Jak streamy upraszczają życie

Aby pokazać jak `Stream` upraszcza życie w przypadku procesowania kolekcji, napiszmy kod, który na podstawie listy `Stringów`, stworzy listę z długościami tych `Stringów`, następnie je posortuje malejąco. Dalej zostawi tylko wartości większe od 5 i wydrukuje na ekranie tylko element 2 i 3 (licząc od 1). Najpierw napiszmy to tak jak dotychczas byśmy to napisali, a później spróbujmy podejścia funkcyjnego.

Podejście "klasyczne"

```
public void oldWay() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Gdańsk", "Łódź");
    List<Integer> lengths = new ArrayList<>();
    for (String city : cities) {
        lengths.add(city.length());
    }
    Collections.sort(lengths, Comparator.<Integer>naturalOrder().reversed());
    List<Integer> lengthsFiltered = new ArrayList<>();
    for (Integer length : lengths) {
        if (length > 5) {
            lengthsFiltered.add(length);
        }
    }
    System.out.println(lengthsFiltered.get(1));
    System.out.println(lengthsFiltered.get(2));
    System.out.println(lengthsFiltered);
}
```

Podejście funkcyjne

```
public void functionalWay() {
    cities.stream()
        .map(String::length)
        .sorted(Comparator.<Integer>naturalOrder().reversed())
        .filter(element -> element > 5)
        .skip(1)
        .limit(2)
        .forEach(System.out::println);
}
```

Powyżej znajdziesz kod realizujący to samo zadanie, ale w sposób funkcyjny. Prawda, że ilość kodu się zmniejszyła? Wizualnie skomplikowanie na pierwszy rzut oka również. Jedyny minus to taki, że aby ten kod zrozumieć, to trzeba się nauczyć tych mechanizmów. Ale ze "starymi" było tak samo 😁.

Przykłady wykorzystania Streamów

Poniżej umieszczam kilka przykładów, żebyś mógł/mogła sprawdzić we własnym zakresie, czy rozumiesz poruszone zagadnienia i ich działanie.

Przykład 1

Zanim uruchomisz ten przykład, postaraj się napisać sobie na kartce co i w jakiej kolejności zostanie wydrukowane na ekranie. Przypomnę, że operacja `sorted()` musi poczekać w swoim kroku na wszystkie elementy, które są zawarte w `Stream`. Nie może przecież posortować wszystkich elementów nie gromadząc ich w jednym miejscu.

```
public void doYouGetIt1() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Szczecin", "Zakopane");
    List<String> byStream = cities.stream()
        .peek(e -> System.out.println("Step 1. peek: " + e))
        .filter(element -> element.length() > 5)
        .peek(e -> System.out.println("Step 2. peek: " + e))
        .sorted(Comparator.<String>naturalOrder().reversed())
        .peek(e -> System.out.println("Step 3. peek: " + e))
        .skip(1)
        .peek(e -> System.out.println("Step 4. peek: " + e))
        .limit(2)
        .peek(e -> System.out.println("Step 5. peek: " + e))
        .collect(Collectors.toList());
}
```

Informacje na ekranie drukują się w następującej kolejności:

```
Step 1. peek: Warszawa
Step 2. peek: Warszawa
Step 1. peek: Gdańsk
Step 2. peek: Gdańsk
Step 1. peek: Łódź
Step 1. peek: Wrocław
Step 2. peek: Wrocław
```

```
Step 1. peek: Szczecin
Step 2. peek: Szczecin
Step 1. peek: Zakopane
Step 2. peek: Zakopane ①
Step 3. peek: Zakopane ②
Step 3. peek: Wrocław
Step 4. peek: Wrocław
Step 5. peek: Wrocław
Step 3. peek: Warszawa
Step 4. peek: Warszawa
Step 5. peek: Warszawa ③
```

① Z racji, że `sorted()` musi poczekać na wszystkie elementy w `Stream`, to wszystkie elementy dochodzą do kroku `sorted()` i tam czekają. Dlatego zanim zostanie wykonany krok `Step 3.`, wszystkie wartości czekają na kroku `sorted()`. Dopiero jak wszystkie elementy w `Stream` dojdą do tego kroku, to zostają wypuszczone dalej, do kroków 3, 4, 5.

② Zakopane zostaje pominięte w tym miejscu, ze względu na `skip(1)`, Zakopane jest na tym etapie pierwszym elementem.

③ Zauważ, że dalej ograniczamy ilość do 2, dlatego do końca dociera tylko Wrocław i Warszawa. Jednocześnie są one przetwarzane sekwencyjnie, czyli w krokach 3, 4, 5.

Przykład 2

Zanim uruchomisz ten przykład, postaraj się napisać sobie na kartce co i w jakiej kolejności zostanie wydrukowane na ekranie.

```
public void doYouGetIt2() {
    Stream.generate(() -> "someString") ①
        .peek(e -> System.out.println("1. peek: " + e))
        .sorted() ②
        .peek(e -> System.out.println("2. peek: " + e))
        .limit(5)
        .forEach(System.out::println);
}
```

① Operacja `generate()` będzie generowała `String` o wartości `someString` w nieskończoność.

② Operacja `sorted()` natomiast czeka aż przyjdą do niej wszystkie elementy, które są zdefiniowane w `Stream`. Z racji, że `generate()` generuje elementy w nieskończoność to `sorted()` nigdy się nie doczeka. Zatem `Stream` będzie wykonywał się w nieskończoność.

Przykład 3

Zanim uruchomisz ten przykład, postaraj się napisać sobie na kartce co i w jakiej kolejności zostanie wydrukowane na ekranie.

```
public void doYouGetIt3() {
    Stream.generate(() -> "someString") ①
        .peek(e -> System.out.println("1. peek: " + e))
        .limit(5) ②
        .peek(e -> System.out.println("2. peek: " + e))
        .sorted() ③
```

```
        .forEach(System.out::println);
}
```

① Operacja `generate()` będzie generowała `String` o wartości `someString` w nieskończoność.

② Operacja `limit()` ogranicza ilość elementów w `Stream` do 5.

③ Operacja `sorted()` natomiast czeka aż przyjdą do niej wszystkie elementy, które są zdefiniowane w `Stream`. Z racji, że po drodze ilość tych elementów została ograniczona do 5, to `sorted()` poczeka na 5 elementów, posortuje je i puści `Stream` dalej.

Przykład 4

Zanim uruchomisz ten przykład, postaraj się napisać sobie na kartce co i w jakiej kolejności zostanie wydrukowane na ekranie.

```
public void doYouGetIt4() {
    List<String> cities = Arrays.asList("Warszawa", "Gdańsk", "Łódź", "Wrocław", "Szczecin", "Zakopane");
    List<String> citiesAfterLimit = cities.stream()
        .filter(a -> a.length() > 100)
        .limit(120)
        .skip(120)
        .collect(Collectors.toList());
    System.out.println(citiesAfterLimit);
}
```

W tym przykładzie na ekranie jest drukowana pusta lista. W tym przypadku `Stream` nie działa w nieskończoność bo jest finite (skończony), zatem Java jest w stanie "wywnioskować", że taki `Stream` ma koniec i finalnie będziemy mieli 0 elementów. Jakby zrobić coś takiego na infinite `Stream`, to program działałby w nieskończoność.

Streamy a typy prymitywne

Można używać czegoś takiego jak `Stream<Integer>`, nie można zapisać `Stream<int>`, bo w generykach można używać tylko klas, pamiętasz? Natomiast są jeszcze takie konstrukcje jak:

- `IntStream`
- `LongStream`
- `DoubleStream`

W jaki sposób można z tych konstrukcji korzystać? Oczywiście można stosować zapisy korzystające z `Stream<Integer>` w sposób pokazany poniżej.

```
public void example() {
    List<Integer> numbers = List.of(1, 2, 3, 4, 5);
    System.out.println(numbers.stream()
        .reduce(1, (left, right) -> left + right));
}
```

Możemy natomiast spróbować skorzystać ze wspomnianego IntStream.

```
public void example() {
    System.out.println(numbers.stream().mapToInt(a -> a).sum());
    IntStream toInt = numbers.stream().mapToInt(a -> a);
    System.out.println(toInt.average());
    //System.out.println(toInt.max()); ①
}
```

① Tak jak w przypadku Stream, nie możemy kilka razy skorzystać z operacji terminującej na IntStream i pozostałych Streamach.

Finite

IntStream, LongStream oraz DoubleStream mogą być utworzone w sposób podobny do interfejsu Stream gdy mówimy o Streamach **finite**.

```
IntStream intStream = IntStream.of(1, 2, 3);
LongStream longStream = LongStream.of(1, 3, 4);
DoubleStream doubleStream = DoubleStream.of(7.5, 3, 2.2);
```

Infinite

I tak samo analogicznie mamy możliwość inicjowania Streamów **infinite**.

```
DoubleStream.generate(() -> Math.random())
    .limit(4)
    .forEach(a -> System.out.println(a));

IntStream.iterate(2, previous -> previous * previous)
    .limit(4)
    .forEach(x -> System.out.println(x));
```

Tworzenie Streamów z zakresem danych

IntStream oraz LongStream (ale DoubleStream już nie) posiadają metody range() oraz rangeClosed(), które pozwalają stworzyć Stream z określonym zakresem danych. Metody te różnią się od siebie tym, że rangeClosed() uwzględnia drugi argument jako wartość, która zostanie dodana do Stream, natomiast range() tego nie robi. Poniżej znajdziesz przykłady.

```
LongStream.range(1, 5)
    .mapToObj(a -> "a" + a)
    .forEach(x -> System.out.println(x));
```

Wykonanie kodu powyżej wydrukuje na ekranie rezultat pokazany poniżej. Zwróć uwagę, że wartość 5 **nie** została uwzględniona przy rezultacie, bo tak właśnie działa operacja range().

```
a1
a2
a3
a4
```

W kolejnym przykładzie używamy operacji `rangeClosed()`, która uwzględni wartość `5`.

```
LongStream.rangeClosed(1, 5)
    .mapToDouble(a -> a * 3.0)
    .forEach(x -> System.out.println(x));
```

Wykonanie kodu powyżej wydrukuje na ekranie rezultat pokazany poniżej. Zwróć uwagę, że wartość `5` została uwzględniona przy rezultacie, bo tak właśnie działa operacja `rangeClosed()`.

```
3.0
6.0
9.0
12.0
15.0
```

W podsumowaniu tego fragmentu materiału znajdziesz rozpisane przejścia między różnego rodzaju `Streamami`. Możesz tam znaleźć operacje, pokroju `mapToDouble()`, które pozwalają przejść np. z `LongStream` na `DoubleStream`, lub z `LongStream` na `Stream<String>`.

Dedykowane klasy Optional

Razem z interfejsami `IntStream`, `LongStream` oraz `DoubleStream` dostajemy klasy `OptionalInt`, `OptionalLong` oraz `OptionalDouble`, które działają analogicznie do `Optional`. Poniżej znajdziesz przykład wykorzystania.

```
OptionalDouble max = DoubleStream.empty().max();
System.out.println(max);
```

Co ciekawe, gdy użyjemy np. `DoubleStream.empty()`, operacje `average()` i `max()` zwracają `Optional`. Jeżeli natomiast na `DoubleStream.empty()` wykonamy operację `sum()` to rezultatem będzie `double` z wartością `0.0`.

boxed

Bardzo przydatną operacją, gdy działamy na `IntStream`, `LongStream` oraz `DoubleStream` jest operacja `boxed()`. Pozwala ona przejść np. z `IntStream` na `Stream<Integer>`. Poniżej znajdziesz przykład.

```
IntStream intStream = IntStream.of(1, 2, 3);
Stream<Integer> boxed = intStream.boxed();
List<Integer> collected = boxed.collect(Collectors.toList());
```

Podsumowanie

Poniżej znajdziesz tabelkę, w której rozpisane zostały mapowania (przejścia) pomiędzy różnymi klasami Streamów.

Stream źródłowy	Map do LongStream	Map do IntStream	Map do DoubleStream	Map do Stream
Stream	mapToLong	mapToInt	mapToDouble	map
DoubleStream	mapToLong	mapToInt	map	mapToObj
IntStream	mapToLong	map	mapToDouble	mapToObj
LongStream	map	mapToInt	mapToDouble	mapToObj

Streamy - Advanced Collectors

W tym miejscu chciałbym wrócić do zagadnienia kolektorów, które były stosowane w operacjach terminujących. Poniżej znajdziesz przykłady kodu z ciekawszymi kolektorami jakie możemy zastosować razem z operacjami terminującymi. Każdy z poniższych przykładów zostanie przedstawiony w formie metody, która przyjmuje na wejściu listę List<String>. Załóżmy, że w każdym z przypadków ta lista będzie wyglądała w ten sposób.

```
List<String> input = Arrays.asList("Warszawa", "Lublin", "Zakopane", "Wrocław", "Kraków", "Poznań");
```

counting

Collectors.counting() służy do zliczenia ilości elementów w Streamie. Poniżej przykład wykorzystania.

```
private void counting(List<String> cities) {
    Long collect1 = cities.stream()
        .collect(Collectors.counting());
    System.out.println(collect1);
}
```

joining

Collectors.joining() służy do złączenia wszystkich elementów Streama w jeden. Ważne jest tutaj, że kolektor ten jest używany do złączenia elementów w String, czyli nie możemy wykorzystać go do tego, aby wynikiem jego wywołania był BigDecimal. Kolektor .joining() może również przyjąć argument określający jak mają być oddzielone od siebie kolejne elementy ze Streama, które zostaną złączone do Stringa.

```
private static void joining(List<String> cities) {
    String result1 = cities.stream()
        .collect(Collectors.joining()); ①
    System.out.println(result1);
```

```
    String result2 = cities.stream()
        .collect(Collectors.joining("== + ==")); ②
    System.out.println(result2);
}
```

① Przykład wykorzystania kolektora bez użycia `Stringa`, który oddziela od siebie łączone elementy.

② Przykład wykorzystania kolektora z użyciem `Stringa`, który oddziela od siebie łączone elementy.

toCollection

Przykład wykorzystania kolektora `Collectors.toCollection()` widzieliśmy już wcześniej. Służy on do określenia konkretnej implementacji kolekcji jaka ma zostać zwrócona po zakończeniu działania `Streama`. Przykład kodu poniżej.

```
private static void toCollection(List<String> cities) {
    Set<String> result = cities.stream()
        .filter(s -> s.startsWith("W"))
        .collect(Collectors.toCollection(TreeSet::new));
    System.out.println(result);
}
```

maxBy oraz minBy

Kolektor `Collectors.maxBy()` oraz `Collectors.minBy()` jak nazwa może wskazywać, służy do zakończenia `Streama` wartością maksymalną lub minimalną określoną na podstawie `Comparatora`, który jest przekazany jako argument wywołania tych operacji. Poniżej przykład wykorzystania `Collectors.maxBy()`.

```
private static void maxBy(List<String> cities) {
    Optional<String> collect1 = cities.stream()
        .collect(Collectors.maxBy(Comparator.naturalOrder()));
    System.out.println(collect1);

    Optional<String> collect2 = cities.stream()
        .max(Comparator.comparing(String::length));
    System.out.println(collect2);
}
```

Pokazane wyżej przykłady `Collectors.maxBy()` oraz `.max()` służą do tego samego - zakończenia `Streama` wartością maksymalną. Wspólne dla nich jest to, że rezultatem wywołania `Streama` jest `Optional`.

A w następnej kolejności znajdziesz przykład wywołania operacji `Collectors.minBy()`. Różnica między przykładami jest taka, że implementacja interfejsu `Comparator` narzuca porównanie na podstawie innych wartości.

```
private static void minBy(List<String> cities) {
    Optional<String> collect3 = cities.stream()
        .collect(Collectors.minBy(Comparator.naturalOrder()));
    System.out.println(collect3);
```

```
    Optional<String> collect4 = cities.stream()
        .min((one, two) -> one.length() - two.length());
    System.out.println(collect4);
}
```

Pokazane wyżej przykłady `Collectors.minBy()` oraz `.min()` służą do tego samego - zakończenia `Streama` wartością minimalną. Wspólne dla nich jest to, że rezultatem wywołania `Streama` jest `Optional`. Różnica między przykładami jest taka, że implementacja interfejsu `Comparator` narzuca porównanie na podstawie innych wartości.

mapping

Kolektor `Collectors.mapping()` jest o tyle ciekawą operacją, że pozwala on jednocześnie dokonać operacji pośredniej `.map()`, a w następnej kolejności wywołać kolektor określony jako drugi argument wywołania. Czyli można powiedzieć, że dodaje on następny poziom kolektora w swojej definicji. Definicja tego kolektora wygląda w ten sposób:

```
mapping(Function function, Collector collector)
```

Przykład wywołania tego kolektora znajdziesz poniżej.

```
private static void mapping(List<String> cities) {
    Integer result = cities.stream()
        .collect(Collectors.mapping(city -> city.length(), Collectors.reducing(0, (a, b) -> a + b)));
    System.out.println(result);
}
```

Czyli zapis, który widzisz powyżej można zapisać inaczej w ten sposób:

```
private static void mapping(List<String> cities) {
    Integer result2 = cities.stream()
        .map(city -> city.length())
        .reduce(0, (a, b) -> a + b);
    System.out.println(result2);
}
```

Czyli `Collectors.mapping()` w pierwszej kolejności wykona operację `.map(city → city.length())`, a w następnej kolejności `.reduce(0, (a, b) → a + b)`. Różnica jest taka, że możemy to zapisać w jednej linijce.

toMap

Dochodzimy do ciekawszych wariantów wywołań klasy `Collectors`. Teraz przechodzimy do kolektora `Collectors.toMap()`, jak możesz się domyślić, jest to kolektor, którego celem jest zakończenie `Streama` z rezultatem w postaci implementacji `Map`.

Zanim przejdziemy do przykładu poniżej, przypomnijmy sobie jak wyglądała lista `cities`.

```
List<String> input = Arrays.asList("Warszawa", "Lublin", "Zakopane", "Wrocław", "Kraków", "Poznań");
```

Jest to bardzo istotne w przykładach, które zostaną pokazane poniżej.

Kolektor `Collectors.toMap()` jest zdefiniowany w kilku wariantach wywołań. Pierwszy wariant wygląda w ten sposób:

```
toMap(Function key, Function value)
```

Przykłady wykorzystania kolektora `toMap()` w tym wariancie są przedstawione poniżej. Argument `k` spodziewa się lambdy określającej klucz zwracanej mapy, natomiast argument `v` oczekuje lambdy określającej wartość zwracanej mapy.

```
private static void toMap(List<String> cities) {
    Map<String, Integer> result1 = cities.stream()
        .collect(Collectors.toMap(key -> key, String::length)); ①
    System.out.println(result1);

    Map<Integer, String> result2 = cities.stream()
        .collect(Collectors.toMap(String::length, value -> value)); ②
    System.out.println(result2);
}
```

① Wywołanie kolektora `Collectors.toMap()` w tym przypadku spowoduje zwrócenie `Map<String, Integer>`. Jako pierwszy argument wywołania `toMap()` określona została lambda, która przyjmuje wartości streama `Stream<String>` gdzie określamy, że kluczem w tej mapie mają być wartości `String`, które były zawarte w `Stream`. Wartościami w `Map<String, Integer>` są natomiast długości tych `String`ów, co jest określone w **method reference** `String::length`.

② Ten przykład jest ciekawszy. Lista danych wejściowych została wcześniej przypomniana nie bez powodu. W przykładzie 2, kluczem w `Map` ma być długość `String`ów, które są przetwarzane w `Stream`. Zarówno `Warszawa` jak i `Zakopane` mają długość `8`. Oznacza to, że będziemy mieli konflikt wartości `Mapy` dla klucza `8`. Co się wtedy stanie? Zostanie wyrzucony wyjątek: "Duplicate key 8 (attempted merging values Warszawa and Zakopane)".

Pokazany wariant kolektora `toMap()` zadziała w momencie, gdy nie będziemy mieli konfliktu wartości `Mapy` dla tej samej wartości klucza. Co natomiast zrobić, jeżeli taki konflikt może wystąpić? Od tego są kolejne warianty. Następnym wariantem jest:

```
toMap(Function key, Function value, BinaryOperator merge)
```

Wariant wywołania kolektora z argumentem `merge` zapewnia nam rozwiązanie tego problemu. W tym miejscu określamy co zrobić gdy wystąpi konflikt pokazany wcześniej.

```
private static void toMap(List<String> cities) {
    Map<Integer, String> result3 = cities.stream()
        .collect(Collectors.toMap(String::length, value -> value, (left, right) -> left + "," + right));
①
    System.out.println(result3);
```

```
    System.out.println(result3.getClass()); ②
}
```

① Dodanie implementacji interfejsu `BinaryOperator` daje nam możliwość określenia co mamy zrobić gdy wystąpi nam konflikt wartości dla tego samego klucza. W tym przypadku sklejamy ze sobą te wartości separując je przecinkiem.

② Ta linijka została wspomniana gdyż wywołując kolektor `toMap()` nie mamy gwarancji jaka implementacja mapy zostanie wykorzystana. Ten fragment kodu ma za zadanie pokazać co to będzie za implementacja. Dokumentacja również o tym wspomina "There are no guarantees on the type".

Jak już się domyślasz, jest też trzeci wariant, który pozwoli określić typ mapy jaka ma zostać zwrócona. Definicja tego wariantu wygląda w sposób pokazany poniżej, gdzie parametr `supplier` odpowiada za przekazanie konkretnej implementacji `Mapy`, która ma zostać wykorzystana.

```
toMap(Function key, Function value, BinaryOperator merge, Supplier supplier)
```

Przykład kodu znajdziesz poniżej.

```
private static void toMap(List<String> cities) {
    TreeMap<Integer, String> result4 = cities.stream()
        .collect(Collectors
            .toMap(String::length, value -> value, (left, right) -> left + "," + right, TreeMap::new));
①
    System.out.println(result4);
    System.out.println(result4.getClass()); ②
}
```

① W tej linijce określamy na końcu przy wykorzystaniu **method reference**, że interesuje nas konkretna implementacja - `TreeMap`.

② Tutaj możemy sprawdzić, czy faktycznie linijka 1 i przekazanie `TreeMap` odniosło oczekiwany efekt.

partitioningBy

Kolektor `Collectors.partitioningBy()` jest bardzo ciekawym kolektorem, bo jego rezultatem zawsze jest `Map<Boolean, List<T>>`, gdzie `T` jest klasą obiektów, na których operujemy w `Stream`. Kolektor ten służy do tego, aby rozdzielić `Stream`, na listy obiektów, które spełniają jakieś założenie i drugą listę obiektów, które tego założenia nie spełniają. Możemy w ten sposób podzielić miasta na listę miast, których długość nazwy jest mniejsza od 4 i listę pozostałych. Kolektor `Collectors.partitioningBy()` występuje w dwóch wariantach, które znajdziesz poniżej.

```
partitioningBy(Predicate predicate)
partitioningBy(Predicate predicate, Collector collector)
```

Pierwszy wariant pozwala nam określić `Predicate`, na podstawie którego nastąpi rozdział na wspomniane listy. Drugi wariant pozwala nam jednocześnie określić jakiego kolektora chcemy użyć do otrzymania wartości dla danego klucza mapy. W pierwszym wariancie będzie wartością mapy będzie lista.

Natomiast przykład jego wykorzystania znajdziesz w kodzie poniżej.

```
private static void partitioningBy(List<String> cities) {
    Map<Boolean, List<String>> result1 = cities.stream()
        .collect(Collectors.partitioningBy(city -> city.length() < 4));
    System.out.println(result1);

    Map<Boolean, List<String>> result2 = cities.stream()
        .collect(Collectors.partitioningBy(city -> city.length() < 10)); ①
    System.out.println(result2);

    Map<Boolean, Set<String>> result3 = cities.stream()
        .collect(Collectors
            .partitioningBy(city -> city.length() < 10, Collectors.toCollection(TreeSet::new))); ②
    System.out.println(result3);
}
```

① Ten wariant zwraca nam mapę `Map<Boolean, List<String>>`, gdzie znajdziemy listy wartości spełniające podany `Predicate` - wtedy kluczem w mapie będzie `true`. Drugi klucz to będzie `false` i tam znajdziemy listę wartości, które tego `Predicate` nie spełniają.

② Przykład jest analogiczny, przy czym tutaj określamy kolektor, który zamiast listy (będącą wartością mapy w przykładzie poprzednim) zwróci `TreeSet`. Dlatego definicja wynikowej mapy to `Map<Boolean, Set<String>>`.

groupingBy

Moim zdaniem najciekawszy z dostępnych kolektorów `Collectors.groupingBy()` - ze względu na ilość możliwości jakie nam daje. Jego wykorzystanie sprowadza się do otrzymania końcowej mapy, która pozwala nam podzielić elementy przetwarzanego `Streama` w grupy na podstawie przekazanych przez nas kryteriów. Typem zwracanym ze `Streama`, który wykorzysta ten kolektor będzie najczęściej `Map<K, List<T>>`, gdzie `K` określa typ klucza, a `T` określa typ wartości. Możliwe jest natomiast użycie innego kolektora, aby wartością mapy zamiast `List<T>` był np. `String`. Poniżej znajdziesz możliwe definicje tego kolektora.

```
groupingBy(Function function) ①
groupingBy(Function function, Collector collector) ②
groupingBy(Function function, Supplier supplier, Collector collector) ③
```

① Wariant pierwszy pozwala nam na określenie funkcji na podstawie której wynikiem wywołania kolektora będzie np, mapa `Map<Integer, List<String>>`.

② Wariant drugi pozwala nam za pomocą parametru `collector` określić kolektor, który ma zostać użyty do określenia wartości w wynikowej mapie. Czyli, np. może to być `Collectors.toCollection(TreeSet::new)`, żeby otrzymać `TreeSet` zamiast `List`.

③ Wariant trzeci natomiast pozwala nam za pomocą parametru `supplier` określić jakiego rodzaju mapa ma być rezultatem wywołania, czyli np. `TreeMap::new`. Parametr `collector` działa analogicznie jak w poprzednim przykładzie.

Przykłady wywołań w kodzie znajdziesz poniżej.

```
private static void groupingBy(List<String> cities) {
    Map<Integer, List<String>> result1 = cities.stream()
        .collect(Collectors.groupingBy(String::length));
    System.out.println(result1);
}
```

W powyższym przykładzie wykorzystujemy wariant 1, czyli przekazujemy tylko `function` i wynikiem wywołania takiego kolektora jest `Map<Integer, List<String>>`.

```
private static void groupingBy(List<String> cities) {
    Map<Integer, Set<String>> result2 = cities.stream()
        .collect(Collectors.groupingBy(String::length, Collectors.toCollection(TreeSet::new)));
    System.out.println(result2);
}
```

W powyższym przykładzie określamy, że wynikiem wywołania ma być `Map<Integer, Set<String>>`, ale interesuje nas konkretnie, że ma to być `TreeSet`, stąd też stosujemy parametr `collector`. Jest to jednocześnie przykład wywołania operacji `groupingBy()` w 2 pokazanym wariancie.

```
private static void groupingBy(List<String> cities) {
    Map<Integer, Set<String>> result3 = cities.stream()
        .collect(Collectors
            .groupingBy(String::length, TreeMap::new, Collectors.toCollection(TreeSet::new)));
    System.out.println(result3);

    TreeMap<Integer, List<String>> result4 = cities.stream()
        .collect(Collectors.groupingBy(String::length, TreeMap::new, Collectors.toList()));
    System.out.println(result4);
}
```

W powyższym przykładzie wykorzystujemy przykłady pokazane wcześniej, tylko, że tym razem dokładamy do tego jeszcze `supplier`, który określa jakiego rodzaju mapa interesuje nas jako rezultat wykonania operacji terminującej `Stream`. Stąd też jako `supplier` przekazujemy `TreeMap::new`.

Kolejne dwa przykłady pokazują, że argument wywołania `collector` to wcale nie musi być np. `Collectors.toList()`. Możemy równie dobrze wykorzystać kolektory takie jak `counting()`, albo `joining()`. W takim przypadku, wynikowa mapa dla danego klucza zliczy ilość wystąpień danych elementów lub złączy wartości w `String`.

```
private static void groupingBy(List<String> cities) {
    Map<Integer, Long> result5 = cities.stream()
        .collect(Collectors.groupingBy(String::length, Collectors.counting()));
    System.out.println(result5);

    Map<Integer, String> result6 = cities.stream()
        .collect(Collectors.groupingBy(String::length, Collectors.joining()));
    System.out.println(result6);
}
```

Poniższy przykład pokazuje natomiast, że możemy również wykorzystać kolektor `Collectors.mapping()`.

W tym przypadku dodajemy kolejną operację mapowania, która zostanie wykonana zanim wywołamy kolektor na wartościach w końcowej mapie. Efektem wywołania kodu poniżej będzie mapa, która jako klucze przetrzymuje długości nazw miast, natomiast jako wartości otrzymamy ostatnią wartość `String` (zgodnie z sortowaniem `Comparator.naturalOrder()`), która jeszcze w dodatku będzie przemapowana wykorzystując `toUpperCase()`, gdyż mapowanie `toUpperCase()` zostanie wywołane przed wywołaniem kolektora `Collectors.maxBy()`. Wynikiem wywołania `Collectors.maxBy()` jest `Optional`, dlatego też definicja wynikowej mapy to `Map<Integer, Optional<String>>`.

```
private static void groupingBy(List<String> cities) {
    Map<Integer, Optional<String>> result7 = cities.stream()
        .collect(Collectors
            .groupingBy(
                String::length,
                Collectors.mapping((String s) -> s.toUpperCase(), Collectors.maxBy(Comparator
.naturalOrder()))
            )
        );
    System.out.println(result7);
}
```

Podsumowanie

Poniżej znajdziesz tabelkę podsumowującą przedstawione kolektory.

Kolektor	Co robi	Co zwraca
counting()	Zlicza ilość elementów w Stream	Long
joining()	Tworzy pojedynczego Stringa, ewentualnie można oddzielić elementy przy pomocy podanego parametru	String
toList() lub toSet()	Tworzy `List` albo `Set`, nie jest powiedziane jaka konkretnie będzie to implementacja, więc nie należy tego zakładać	List<T> lub Set<T>
toCollection(Supplier s)	Tworzy kolekcję podanego typu kolekcji	Collection
maxBy(Comparator c) lub minBy(Comparator c)	Znajduje największy/najmniejszy element	Optional<T>

mapping()	Ta metoda jest takim ciekawym tworem, który najpierw przemapowuje kolekcję, a potem stosuje na niej kolektor, np:
	`elements.stream()`
	`.collect(Collectors`
	`.mapping(function, collector));`
	co jest tożsame z:
	`elements.stream()`
	`.map(function)`
	`.collect(collector);`
		Collector type

toMap()	Tworzy mapę przy wykorzystaniu podanej funkcji żeby mapować klucze i wartości	Map

Kolektor	Co robi	Co zwraca
partitioningBy()	Tworzy mapę zgrupowaną na podstawie podanego predykatu, gdzie klucze są true/false	Map<Boolean, List<V>>
groupingBy()	Grupuje elementy na podstawie podanej funkcji i zwraca mapę z podziałem na grupy	Map<K, List<V>>

Programowanie funkcyjne, interfejsy funkcyjne - projekt

Napisz program, w którym zdefiniujesz klasę `Producer`, która będzie w stanie dostarczać implementacje interfejsów funkcyjnych takich jak: `Supplier`, `Consumer` oraz `Function`. Zaimplementuj każdy ze wspomnianych interfejsów funkcyjnych przy wykorzystaniu lambdy oraz wykonaj na nich dedykowaną dla nich metodę.

Utwórz klasę `Transformer`, która będzie w stanie zwrócić implementacje interfejsów funkcyjnych (`Function`, `UnaryOperator`), które pozwolą nam transformować dane w metodach `.map()`, które są dostępne w klasie `Optional`.

Na koniec stwórz klasę `MyConsumer`, która zdefiniuje metody, będące w stanie przyjąć jako argumenty interfejsy funkcyjne takie jak: `Consumer`, `Supplier` oraz `Function`, a następnie wydrukować na ekranie wartości zwrócone przez metody wywołane na tych interfejsach funkcyjnych.

Streamy - projekt

Wyobraźmy sobie, że mamy dane ze sklepu internetowego nazwanego... (w sumie to nazwij go sobie jak masz ochotę ☺).

Dane te są dostarczone w postaci obiektu klasy `DataFactory`, która fabrykuje takie dane.

Fabrykujemy te dane w taki sposób, bo nie poznaliśmy jeszcze metod na odczyt danych z pliku, bądź też z bazy danych.

W naszym sklepie będziemy operować na schemacie klas, który jest przedstawiony poniżej.

Klasa Purchase oraz enumy Delivery, Payment oraz Status

```
public class Purchase {

    private final Client buyer;

    private final Product product;

    private final long quantity;

    private final Delivery delivery;

    private final Payment payment;

    private final LocalDate when;

    private Status status = Status.PAID;

    // konstruktory, gettery itp

    public enum Delivery {
        IN_POST,
        UPS,
        DHL
    }

    public enum Payment {
        CASH,
        BLIK,
        CREDIT_CARD
    }

    public enum Status {
        PAID,
        SENT,
        DONE
    }
}
```

Klasa Client

```
public class Client implements Comparable<Client> {
```

```
    private final String id;

    private final String name;

    private final String surname;

    private final BigInteger pesel;

    private final String city;

    // konstruktory, gettery itp

    @Override
    public int compareTo(final Client o) {
        return this.id.compareTo(o.id);
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        final Client client = (Client) o;
        return Objects.equals(pesel, client.pesel);
    }

    @Override
    public int hashCode() {
        return Objects.hash(pesel);
    }
}
```

Klasa Product oraz enum Category

```
public class Product implements Comparable<Product> {

    private final String id;

    private final String name;

    private final Category category;

    private final Money price;

    // konstruktory, gettery itp

    @Override
    public int compareTo(final Product o) {
        int thisNumber = Integer.parseInt(id.substring(7));
        int otherNumber = Integer.parseInt(o.id.substring(7));
        return thisNumber - otherNumber;
    }

    public enum Category {
        HOBBY,
        CLOTHES,
        GARDEN,
        AUTOMOTIVE
    }
}
```

Klasa Money

```
public class Money {

    private final BigDecimal value;

    private final Currency currency;

    // konstruktory, gettery itp

    public enum Currency {
        PLN,
        EUR
    }
}
```

Mając dane jakie dostarcza nam klasa `DataFactory`, wykonaj polecenia wymienione poniżej. Wszędzie stosuj podejście funkcyjne i Streamy.

Najpierw zapoznaj się z poleceniami do wykonania. Kod klasy `DataFactory` tworzącej dane do wykorzystania znajdziesz w formie kodu źródłowego umieszczonego w poście z zadaniem.

W poleceniach wyróżniamy 3 poziomy skomplikowania zagadnienia: pierwszy, drugi oraz trzeci 😉.

Zadania z poziomu pierwszego:

1. Oblicz jaka ilość klientów dokonała zakupu w naszym sklepie.
2. Oblicz jaka ilość klientów płaciła Blikiem.
3. Oblicz jaka ilość klientów płaciła kartą kredytową.
4. Oblicz jaka ilość zakupów została wykonana w walucie `EUR`.
5. Oblicz ile unikalnych kupionych produktów zostało zakupionych w `EUR`.

Zadania z poziomu drugiego:

1. Oblicz ile PLN wydała w sklepie każda z osób, które dokonały u nas zakupu. Uwzględnij tylko zakupy dokonane w PLN.
2. Przygotuj metodę, która przyjmie konkretną kategorię i dla tej kategorii zwróci mapę, gdzie kluczem będzie `id` klienta, a wartością ilość kupionych przez niego produktów z podanej kategorii (weź pod uwagę tylko te transakcje, w których ilość kupionych produktów jest większa niż 1).
3. Każde zamówienie początkowo ma status `PAID`. Zaktualizuj status wszystkich zamówień, wykorzystując sprawdzenie statusu każdego konkretnego zamówienia poprzez kod klasy `OrderService` podany poniżej. Aby sprawdzić status każdego zamówienia wykorzystaj kod klasy `OrderService` podany poniżej. Na koniec oblicz ile zamówień zostało przetworzonych, czyli mają status `DONE`.

W rzeczywistości, takie rzeczy sprawdzałoby się przykładowo w innym systemie zewnętrznym wywołując jego API. Tutaj natomiast, zmiana takiego statusu zamówienia jest "na sztywno" określana przez metodę `checkOrderStatus()`.

Klasa OrderService

```
import java.util.Set;

public class OrderService {

    private static final Set<String> PAID_STATUSES
        = Set.of("0", "5", "12", "15", "18");

    private static final Set<String> SENT_STATUSES
        = Set.of("1", "2", "4", "7", "8", "11", "13", "16", "19", "21", "25");

    public static Purchase.Status checkOrderStatus(final Purchase purchase) {
        if (PAID_STATUSES.contains(purchase.getProduct().getId().replace("product", ""))) {
            return Purchase.Status.PAID;
        } else if (SENT_STATUSES.contains(purchase.getProduct().getId().replace("product", ""))) {
            return Purchase.Status.SENT;
        } else {
            return Purchase.Status.DONE;
        }
    }
}
```

4. Oblicz ilu unikalnych klientów kupiło produkt wyceniony w `EUR` (klienci nie mogą się powtarzać, pamiętaj, że jeden mógł kupić kilka produktów). Dodatkowo stwórz mapę w której kluczem jest id klienta, a wartością lista zakupów produktów tego klienta w `EUR`.

5. Przygotuj mapę, gdzie kluczem będzie rocznik klienta, a wartościami, lista wszystkich produktów jakie klient z danego rocznika kupił. Rocznik weź z numeru PESEL, nie musi być to pełne 1987, może być przykładowo 87. Posortuj mapę po kluczu rosnąco.
6. Stwórz mapę, gdzie kluczem będą roczniki, a wartością unikalny zestaw kategorii produktów kupionych przez dany rocznik.
7. Jaki jest drugi najczęściej kupowany produkt? Jeżeli kilka produktów jest kupionych w takiej samej ilości, posortuj je alfabetycznie po `id`, i nadal weź drugi. Czyli sortujesz najpierw po największej ilości wystąpień danego produktu, a potem po `id`.

Zadania z poziomu trzeciego:

1. Dla ludzi starszych niż 50 lat stwórz strukturę, w której zawrzesz informacje: rocznik, najmniej popularna kategoria dla danego rocznika, ilość transakcji dla danego rocznika w obrębie danej kategorii. Mówiąc najmniej popularna mamy na myśli, najmniejszą ilość dokonanych zakupów w obrębie danej kategorii. Np: "rocznik: 62, najmniej popularna kategoria: GARDEN, transakcje: 5".
2. Który rocznik kupił najwięcej produktów?

Lambda - zadania

1. Stwórz interface funkcyjny z metodą przyjmującą int i zwracającą String. Zaimplementuj ten interface przy wykorzystaniu lambdy. Spróbuj zapisać lambdę na parę pokazanych sposobów.
2. Stwórz interface funkcyjny z metodą przyjmującą int i String i zwracającą String. Zaimplementuj ten interface przy wykorzystaniu lambdy. Spróbuj zapisać lambdę na parę pokazanych sposobów.
3. Stwórz interface funkcyjny z metodą przyjmującą int, int oraz String i zwracającą String. Zaimplementuj ten interface przy wykorzystaniu lambdy. Spróbuj zapisać lambdę na parę pokazanych sposobów.
4. Dla porównania, zaimplementuj interface z poprzedniego ćwiczenia przy wykorzystaniu klasy implementującej interface, a nie lambdy.

Programowanie funkcyjne, Streamy - zadania

1. Napisz funkcyjną implementację silni, czyli taką, która wykorzystuje Stream.
2. Wykorzystując Streamy, na podanej tablicy liter wykonaj następujące operacje (w podanej kolejności):
   - zamień wszystkie litery na wielką literę,
   - pozbądź się litery 'X',
   - posortuj malejąco,
   - zwróć wynik jako jeden String.

```
String[] letters = {"z", "x", "a", "j", "a", "v", "x", "k", "a", "x"};
```

3. Znajdź drugi najmniejszy element w liście poniżej:

```
List<Integer> numbers = Arrays.asList(1, 5, 16, 18, 2, 5, 2, 6, 2, 1, 6, 1, 23, 64, 34);
```

4. Znajdź drugą największą wartość w liście poniżej:

```
List<Integer> numbers = Arrays.asList(1, 5, 16, 18, 2, 5, 2, 6, 2, 1, 6, 1, 23, 64, 34);
```

5. Jeżeli podpowiem, że String posiada metodę `.chars()`, która generuje IntStream, spróbuj zaimplementować sprawdzanie, czy słowo/zdanie jest palindromem, przy wykorzystaniu Streamów.