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.

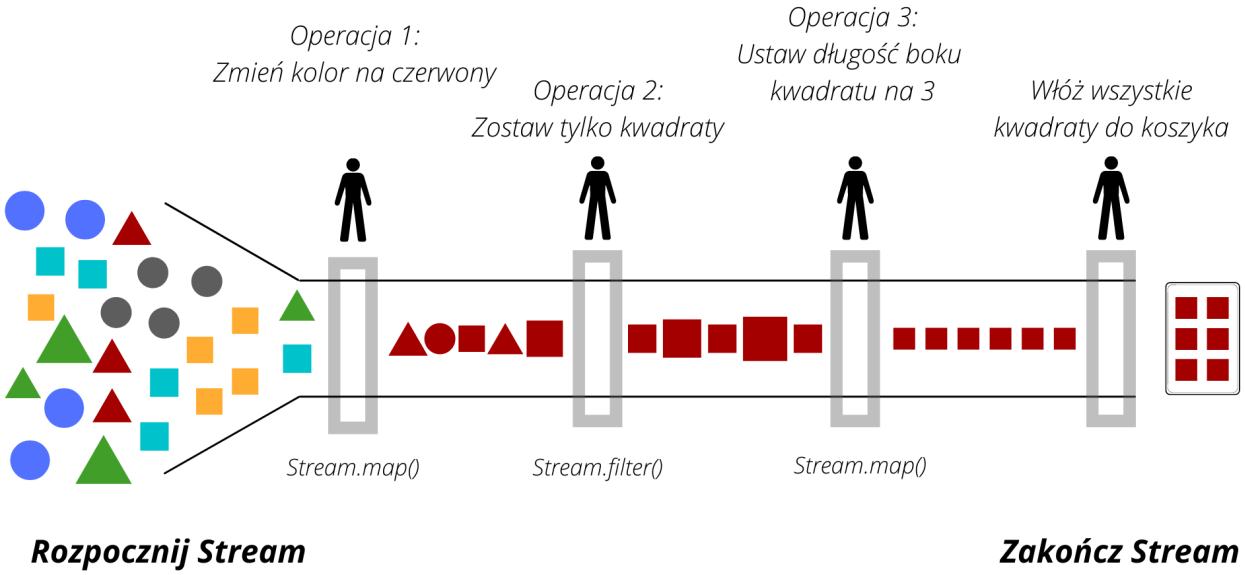
Wyobrażenie sobie takiej analogii jest o tyle istotne, że `Streamy` są ***Lazy***. Oznacza to, że operacje pośrednie nie zostaną wykonane dopóki nie uruchomimy operacji terminującej. Jeżeli wyobrazimy sobie, że to Java jest takim kierownikiem, to możemy też sobie wyobrazić, że jeżeli nie powiemy kierownikowi 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>>