O'REILLY®

# Nowoczesne receptury w Javie

PROSTE ROZWIĄZANIA TRUDNYCH PROBLEMÓW

Helion

Ken Kousen

Tytuł oryginału: Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-4073-2



Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/noreja.zip*

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/noreja*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

ROZDZIAŁ 4.

# Komparatory i kolektory

W Javie 8 wzbogacono interfejs `Comparator` o kilka statycznych i domyślnych metod, które znacznie ułatwiają wykonywanie operacji sortowania. Teraz kolekcję zwykłych obiektów można posortować według jednej własności, następnie według drugiej, potem trzeciej itd. tylko za pomocą serii wywołań metod biblioteki.

Ponadto w Javie 8 dodano nową klasę pomocniczą o nazwie `java.util.stream.Collectors`, która udostępnia statyczne metody do konwertowania strumieni na różne typy kolekcji. Kolektory mogą być też stosowane „w dole strumienia", to znaczy można ich używać jako etapu przetwarzania końcowego po operacjach grupowania lub partycjonowania.

Wszystkie te koncepcje zilustrowałem przedstawionymi w tym rozdziale recepturami.

## 4.1 Sortowanie przy użyciu komparatora

### Problem

Chcemy posortować obiekty.

### Rozwiązanie

Należy użyć metody `sorted` z interfejsu `Stream` z komparatorem zaimplementowanym jako wyrażenie lambda lub wygenerowanym przez jedną ze statycznych metod `compare` z interfejsu `Comparator`.

### Omówienie

Metoda `sorted` z interfejsu `Stream` tworzy nowy posortowany strumień, stosując naturalne uporządkowanie dla danej klasy. Porządek naturalny określa się przez implementację interfejsu `java.util.Comparable`.

Zastanów się na przykład nad metodą sortowania kolekcji łańcuchów przedstawioną na listingu 4.1.

*Listing 4.1. Leksykograficzne sortowanie łańcuchów*

```
private List<String> sampleStrings =
    Arrays.asList("to", "jest", "lista", "łańcuchów", "tekstowych");

public List<String> defaultSort() {
    Collections.sort(sampleStrings); ❶
    return sampleStrings;
}

public List<String> defaultSortUsingStreams() {
    return sampleStrings.stream()
        .sorted() ❷
        .collect(Collectors.toList());
}
```

❶ Domyślne sortowanie w Javie 7 i wersjach starszych

❷ Domyślne sortowanie w Javie 8 i wersjach nowszych

W języku Java klasa pomocnicza o nazwie `Collections` była dostępna od dodania mechanizmu kolekcji w wersji 1.2. Statyczna metoda `sort` z klasy `Collections` pobiera jako argument listę `List`, ale ma typ zwrotny `void`. Metoda `sort` działa destrukcyjnie, czyli modyfikuje otrzymaną kolekcję. Taki sposób działania jest niezgodny z funkcyjnymi technikami przyjętymi w Javie 8, w których duży nacisk jest kładziony na niezmienność.

W Javie 8 taki sam efekt sortowania strumieni uzyskuje się przy użyciu metody `sorted`, która jednak zwraca nowy strumień zamiast modyfikować pierwotną kolekcję. W tym przykładzie po posortowaniu kolekcji zwrócona lista zostaje posortowana w naturalnym porządku klasy. Jeśli chodzi o łańcuchy, naturalny jest porządek leksykograficzny, który sprowadza się do uporządkowania alfabetycznego, gdy wszystkie litery są małe, tak jak w przedstawionym przykładzie.

Jeżeli trzeba posortować łańcuchy w inny sposób, to można skorzystać z przeciążonej wersji metody `sorted`, która pobiera komparator jako argument.

Na listingu 4.2 pokazane są dwa sposoby sortowania łańcuchów według długości.

*Listing 4.2. Sortowanie łańcuchów według długości*

```
public List<String> lengthSortUsingSorted() {
    return sampleStrings.stream()
        .sorted((s1, s2) -> s1.length() - s2.length()) ❶
        .collect(toList());
}

public List<String> lengthSortUsingComparator() {
    return sampleStrings.stream()
        .sorted(Comparator.comparingInt(String::length)) ❷
        .collect(toList());
}
```

❶ Użycie lambdy jako komparatora do sortowania według długości

❷ Użycie komparatora wykorzystującego metodę `comparingInt`

Argument metody `sorted` jest komparatorem typu `java.util.Comparator`, który jest interfejsem funkcyjnym. W metodzie `lengthSortUsingSorted` rolę implementacji metody `compare` interfejsu `Comparator`

stanowi wyrażenie lambda. Do Javy 7 taka implementacja miałaby postać klasy anonimowej, ale tutaj wystarczy wyrażenie lambda.

W Javie 8 dodano `sort(Comparator)` jako domyślną metodę obiektu w interfejsie `List`, która jest równoważna ze statyczną metodą `void sort(List, Comparator)` z interfejsu `Collections`. Obie te metody są destrukcyjnymi operacjami sortowania o typie zwrotnym `void`, wobec czego w odniesieniu do strumieni preferowane jest opisane powyżej podejście polegające na użyciu metody `sorted(Comparator)`, która zwraca nowy posortowany strumień.

Druga metoda, `lengthSortUsingComparator`, wykorzystuje jedną ze statycznych metod dodanych do interfejsu `Comparator`. Metoda `comparingInt` pobiera argument typu `ToIntFunction` przekształcający łańcuch w liczbę typu `int`, w dokumentacji zwaną `keyExtractor`, i generuje komparator sortujący kolekcję za pomocą tego klucza.

Domyślne metody dodane do interfejsu `Comparator` są niezwykle przydatne. Choć napisanie komparatora sortującego według długości jest bardzo łatwe, to gdy trzeba zastosować sortowanie według więcej niż jednego pola, sprawy się komplikują. Weźmy na przykład sortowanie łańcuchów najpierw według długości, a następnie tych, które mają taką samą długość — alfabetycznie. Dzięki statycznym metodom domyślnym interfejsu `Comparator` to zadanie jest prawie banalne, czego dowodem jest kod przedstawiony na listingu 4.3.

*Listing 4.3. Sortowanie według długości, a potem leksykograficznie łańcuchów o takiej samej długości*

```
public List<String> lengthSortThenAlphaSort() {
    return sampleStrings.stream()
        .sorted(comparing(String::length) ❶
                .thenComparing(naturalOrder()))
        .collect(toList());
}
```

❶ Sortowanie według długości, a potem alfabetycznie łańcuchów o takiej samej długości

Interfejs `Comparator` udostępnia domyślną metodę o nazwie `thenComparing`. Tak jak metoda `comparing`, ta metoda również jako argument pobiera funkcję zwaną `keyExtractor`. Dołączenie jej do metody `comparing` pozwala uzyskać porównywanie polegające na tym, że najpierw zostaje zwrócony komparator porównujący według pierwszej wielkości, a następnie równe elementy według drugiej wielkości itd.

Statyczne importy często sprawiają, że kod źródłowy staje się bardziej czytelny. Gdy programista przyzwyczai się do statycznych metod z interfejsów `Comparator` i `Collectors`, bardzo ułatwi mu to pisanie kodu źródłowego. W tym przypadku statycznie zostały zaimportowane metody `comparing` i `naturalOrder`.

Technikę tę można stosować w odniesieniu do każdej klasy, nawet jeśli nie implementuje interfejsu `Comparable`. Spójrz na klasę `Golfer` z listingu 4.4.

*Listing 4.4. Klasa dla golfistów*

```
public class Golfer {
    private String first;
    private String last;
    private int score;

    // ...inne metody...
}
```

Gdyby trzeba było utworzyć tablicę wyników turnieju, dobrym pomysłem mogłoby być zastosowanie sortowania najpierw według liczby punktów, potem według nazwiska i na końcu według imienia. Na listingu 4.5 pokazałem, jak to zrobić.

*Listing 4.5. Sortowanie golfistów*

```
private List<Golfer> golfers = Arrays.asList(
    new Golfer("Jack", "Nicklaus", 68),
    new Golfer("Tiger", "Woods", 70),
    new Golfer("Tom", "Watson", 70),
    new Golfer("Ty", "Webb", 68),
    new Golfer("Bubba", "Watson", 70)
);

public List<Golfer> sortByScoreThenLastThenFirst() {
    return golfers.stream()
        .sorted(comparingInt(Golfer::getScore)
                    .thenComparing(Golfer::getLast)
                    .thenComparing(Golfer::getFirst))
        .collect(toList());
}
```

Na listingu 4.6 pokazany jest wynik działania metody `sortByScoreThenLastThenFirst`.

*Listing 4.6. Posortowana lista golfistów*

```
Golfer{first='Jack', last='Nicklaus', score=68}
Golfer{first='Ty', last='Webb', score=68}
Golfer{first='Bubba', last='Watson', score=70}
Golfer{first='Tom', last='Watson', score=70}
Golfer{first='Tiger', last='Woods', score=70}
```

Golfiści są posortowani według liczby punktów, dzięki czemu Nicklaus i Webb znajdują się przed Woodsem i obydwoma Watsonami[1]. Następnie elementy mające tę samą liczbę punktów są sortowane według pola nazwiska, dzięki czemu Nicklaus trafia przed Webba, a Watson przed Woodsa. Na koniec elementy o takich samych nazwiskach zostają posortowane według imion, dzięki czemu Bubba Watson znajduje się przed Tomem Watsonem.

Domyślne i statyczne metody interfejsu `Comparator`, wraz z nową metodą `sorted` z interfejsu `Stream`, znacznie ułatwiają przeprowadzanie skomplikowanych operacji sortowania.

# 4.2 Konwertowanie strumienia na kolekcję

## Problem

Chcemy przekształcić przetworzony strumień w listę, zbiór lub inną liniową kolekcję.

## Rozwiązanie

Należy użyć metod `toList`, `toSet` lub `toCollection` z klasy pomocniczej `Collectors`.

[1] Ty Webb to oczywiście postać z filmu *Golfiarze*. Judge Smails: „Ty, jak ci dzisiaj idzie?" Ty Webb: „Och, Judge, nie zapisuję punktów." Smails: „To skąd wiesz, jak wypadasz w porównaniu z innymi golfiarzami?" Webb: „Porównuję wzrost." Dodanie sortowania według wzrostu pozostawiam jako ćwiczenie do samodzielnego wykonania.

## Omówienie

Typowym sposobem postępowania w Javie 8 jest przepuszczanie elementów strumienia przez potok operacji pośrednich, po których następuje operacja końcowa. Jedną z takich operacji końcowych jest metoda `collect` służąca do konwertowania strumienia na kolekcję.

Metoda `collect` z interfejsu `Stream` występuje w dwóch przeciążonych wersjach pokazanych na listingu 4.7.

*Listing 4.7. Metoda collect z interfejsu Stream<T>*

```
<R,A> R collect(Collector<? super T,A,R> collector)
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

W centrum zainteresowania tej receptury jest pierwsza wersja, która pobiera kolektor jako argument. Kolektory wykonują „zmienną operację redukcji" polegającą na zakumulowaniu elementów w kontenerze wynikowym. W tym przypadku wynik będzie kolekcją.

`Collector` to interfejs, więc nie ma możliwości tworzenia jego obiektów. Zawiera on jednak statyczną metodę służącą do tworzenia obiektów, choć często można sobie poradzić w lepszy i łatwiejszy sposób.

W API Javy 8 często używana jest statyczna metoda o nazwie `of`, która pełni rolę fabryki.

W tej recepturze egzemplarze typu `Collector` są tworzone za pomocą statycznych metod z klasy `Collectors`, które są przekazywane jako argumenty do metody `Stream.collect` w celu wstawienia elementów do kolekcji.

Na listingu 4.8 pokazany jest prosty przykład tworzący listę[2].

*Listing 4.8. Tworzenie listy*

```
List<String> superHeroes =
    Stream.of("Mr. Furious", "The Blue Raja", "The Shoveler",
              "The Bowler", "Invisible Boy", "The Spleen", "The Sphinx")
        .collect(Collectors.toList());
```

Ta metoda tworzy i zapełnia listę `ArrayList` podanymi elementami strumienia. Zbiory tworzy się tak samo łatwo, czego dowodem jest listing 4.9.

*Listing 4.9. Tworzenie zbioru*

```
Set<String> villains =
    Stream.of("Casanova Frankenstein", "The Disco Boys",
              "The Not-So-Goodie Mob", "The Suits", "The Suzies",
```

[2] Nazwiska w tej recepturze zaczerpnąłem z filmu *Superbohaterowie*, jednego z najbardziej niedocenianych obrazów lat 90. (Mr. Furious: „Lance Hunt *to* Captain Amazing." The Shoveler: „Lance Hunt nosi okulary. Captain Amazing *nie* nosi okularów." Mr. Furious: „Zdejmuje je podczas transformacji." The Shoveler: „To bez sensu! Przecież nic by nie *widział*!").

```
                "The Furriers", "The Furriers") ❶
            .collect(Collectors.toSet());
    }
```

❶ Powtórzona nazwa, która zostanie usunięta podczas konwersji na zbiór

Ta metoda tworzy egzemplarz kolekcji `HashSet` i wstawia do niego elementy, odrzucając wszystkie duplikaty.

W obu przedstawionych przykładach zostały użyte domyślne struktury danych — `ArrayList` dla `List` i `HashSet` dla `Set`. Jeśli konieczne jest określenie konkretnej struktury danych, należy użyć metody `Collectors.toCollection`, która jako argument przyjmuje dostawcę (`Supplier`). Na listingu 4.10 pokazałem przykład jej zastosowania.

*Listing 4.10. Tworzenie listy powiązanej*

```
List<String> actors =
    Stream.of("Hank Azaria", "Janeane Garofalo", "William H. Macy",
              "Paul Reubens", "Ben Stiller", "Kel Mitchell", "Wes Studi")
        .collect(Collectors.toCollection(LinkedList::new));
}
```

Argumentem metody `toCollection` jest dostawca kolekcji, w którego roli w tym przypadku wystąpił konstruktor listy powiązanej `LinkedList`. Metoda `collect` tworzy egzemplarz typu `LinkedList` i wstawia do niego podane nazwiska.

Klasa `Collectors` zawiera też metodę do tworzenia tablic obiektów. Nazywa się ona `toArray` i występuje w dwóch wersjach:

```
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> generator);
```

Pierwsza wersja zwraca tablicę zawierającą elementy strumienia, ale bez określania typu. Z kolei druga pobiera funkcję, która tworzy nową tablicę żądanego typu o długości równej rozmiarowi strumienia. Najłatwiej jej używać z referencją do konstruktora tablicy (listing 4.11).

*Listing 4.11. Tworzenie tablicy*

```
String[] wannabes =
    Stream.of("The Waffler", "Reverse Psychologist", "PMS Avenger")
          .toArray(String[]::new); ❶
}
```

❶ Referencja do konstruktora tablicy jako dostawca

Zostaje zwrócona tablica określonego typu, której długość jest równa liczbie elementów w strumieniu.

Aby dokonać transformacji na słownik, należy użyć metody `Collectors.toMap`, która pobiera dwa obiekty typu `Function` — po jednym dla kluczy i wartości.

Weźmy na przykład zwykły obiekt `Actor` zawierający pola `name` i `role`. Jeśli dany będzie zbiór aktorów z pewnego filmu, to program przedstawiony na listingu 4.12 utworzy z nich słownik.

*Listing 4.12. Tworzenie słownika*

```
Set<Actor> actors = mysteryMen.getActors();

Map<String, String> actorMap = actors.stream()
    .collect(Collectors.toMap(Actor::getName, Actor::getRole)); ❶

actorMap.forEach((key,value) ->
    System.out.printf("%s wcielił(a) się w postać %s%n", key, value));
```

❶ Funkcje tworzące klucze i wartości

Oto wynik działania tego kodu:

```
Janeane Garofalo wcielił(a) się w postać The Bowler
Greg Kinnear wcielił(a) się w postać Captain Amazing
William H. Macy wcielił(a) się w postać The Shoveler
Paul Reubens wcielił(a) się w postać The Spleen
Wes Studi wcielił(a) się w postać The Sphinx
Kel Mitchell wcielił(a) się w postać Invisible Boy
Geoffrey Rush wcielił(a) się w postać Casanova Frankenstein
Ben Stiller wcielił(a) się w postać Mr. Furious
Hank Azaria wcielił(a) się w postać The Blue Raja
```

Podobny kod można napisać dla `ConcurrentMap` przy użyciu metody `toConcurrentMap`.

### Zobacz również

Dostawców opisałem w recepturze 2.2. Referencje do konstruktorów są opisane w recepturze 1.3. Jeszcze jeden przykład użycia metody `toMap` znajduje się w recepturze 4.3.

## 4.3 Dodawanie liniowej kolekcji do słownika

### Problem

Chcemy dodać kolekcję obiektów do słownika, w którym klucz jest jedną z własności obiektu, a wartość jest samym obiektem.

### Rozwiązanie

Należy użyć metody `toMap` z klasy `Collectors` i metody `Function.identity`.

### Omówienie

Opisywany tu przypadek jest bardzo krótki i konkretny, ale zaprezentowane rozwiązanie może być bardzo przydatne.

Powiedzmy, że mamy listę obiektów w postaci książek klasy `Book`. Są to zwykłe obiekty Javy zawierające identyfikator, tytuł oraz cenę. Na listingu 4.13 pokazałem jest skrócony kod klasy `Book`.

*Listing 4.13. Prosta klasa reprezentująca książkę*

```
public class Book {
    private int id;
    private String name;
    private double price;

    // ...inne metody...
}
```

Teraz powiedzmy, że mamy kolekcję obiektów klasy `Book`, taką jak pokazana na listingu 4.14.

*Listing 4.14. Kolekcja książek*

```
List<Book> books = Arrays.asList(
    new Book(1, "Nowoczesne receptury Javy", 49.99),
    new Book(2, "Java 8 w akcji", 49.99),
    new Book(3, "Java SE8 dla niecierpliwych ", 39.99),
    new Book(4, "Programowanie funkcyjne w Javie", 27.64),
    new Book(5, "Java i Groovy", 45.99)
    new Book(6, "Receptury Gradle dla Androida", 23.76)
);
```

W wielu przypadkach zamiast listy lepiej byłoby użyć słownika, w którym klucze byłyby identyfikatorami książek, a wartości samymi książkami. Bardzo łatwo taką strukturę uzyskać za pomocą metody `toMap` z klasy `Collectors`. Na listingu 4.15 pokazane są dwa sposoby realizacji tego celu.

*Listing 4.15. Dodawanie książek do słownika*

```
Map<Integer, Book> bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, b -> b)); ❶

bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, Function.identity())); ❷
```

❶ Tożsamościowa lambda: przyjmuje element i go zwraca

❷ Tak samo działa statyczna metoda `identity` z interfejsu `Function`

Metoda `toMap` z klasy `Collectors` przyjmuje jako argumenty dwa obiekty typu `Function`, z których pierwszy generuje klucz, a drugi generuje wartość z przekazanego mu obiektu. W tym przypadku klucze są odwzorowywane przez metodę `getId` klasy `Book`, a wartości są samymi książkami.

Pierwsza metoda `toMap` na listingu 4.15 używa metody `getId` do odwzorowywania kluczy i wyrażenia lambda, które po prostu zwraca swój parametr. Drugi przykład osiąga ten sam cel przy użyciu statycznej metody `identity` z interfejsu `Function`.

## Zobacz również

Funkcje zostały opisane w recepturze 2.4, w której ponadto znajduje się opis operatorów jedno- i dwuargumentowych.

### Dwie statyczne metody tożsamościowe

Statyczna metoda `identity` w interfejsie `Function` ma następującą sygnaturę:

```
static <T> Function<T,T> identity()
```

Na listingu 4.16 pokazana jest jej implementacja w bibliotece standardowej.

*Listing 4.16. Statyczna metoda identity w interfejsie Function*

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

Interfejs `UnaryOperator` rozszerza interfejs `Function`, ale nie ma możliwości przesłonięcia metody statycznej. W dokumentacji Javadocs można znaleźć informację, że ten interfejs także deklaruje statyczną metodę `identity`:

```
static <T> UnaryOperator<T> identity()
```

Jej implementacja w bibliotece standardowej jest zasadniczo taka sama, jak widać na listingu 4.17.

*Listing 4.17. Statyczna metoda identity w interfejsie UnaryOperator*

```
static <T> UnaryOperator<T> identity() {
    return t -> t;
}
```

Te dwie metody różnią się tylko sposobem wywoływania (za pomocą odpowiedniej nazwy interfejsu) i odpowiadającymi im typami zwrotnymi. W tym przypadku nie ma znaczenia, która zostanie użyta, ale warto wiedzieć, że są dwie takie metody.

Decyzja, czy przekazać lambdę, czy użyć metody statycznej, jest jedynie kwestią stylu. Przeniesienie wartości z kolekcji do słownika, w którym kluczami są własności obiektów, a wartościami same te obiekty, w obu przypadkach jest bardzo łatwe.

# 4.4 Sortowanie słowników

## Problem

Chcemy posortować słownik według kluczy lub wartości.

## Rozwiązanie

Należy użyć statycznych metod interfejsu `Map.Entry`.

## Omówienie

Interfejs `Map` od zawsze zawiera publiczny, statyczny interfejs wewnętrzny o nazwie `Map.Entry`, który reprezentuje parę klucz – wartość. Metoda `Map.entrySet` zwraca zbiór elementów `Map.Entry`. Przed pojawieniem się Javy 8 do najczęściej wykorzystywanych metod tego interfejsu należały `getKey` i `getValue`, pobierające odpowiednio klucz i wartość.

W Javie 8 zostały dodane statyczne metody wymienione w tabeli 4.1.

*Tabela 4.1. Statyczne metody w interfejsie Map.Entry (z dokumentacji Javy 8)*

| Metoda | Opis |
|---|---|
| `comparingByKey()` | Zwraca komparator porównujący `Map.Entry` w naturalnym porządku według kluczy |
| `comparingByKey(Comparator<? super K> cmp)` | Zwraca komparator porównujący `Map.Entry` według kluczy przy użyciu przekazanego komparatora |
| `comparingByValue()` | Zwraca komparator porównujący `Map.Entry` w naturalnym porządku według wartości |
| `comparingByValue(Comparator<? super V> cmp)` | Zwraca komparator porównujący `Map.Entry` według wartości przy użyciu przekazanego komparatora |

W ramach przykładu zastosowania tych metod na listingu 4.18 przedstawiony jest kod generujący słownik (`Map`), który kojarzy długości słów z listami słów o danej długości. W każdym systemie Unix w katalogu *usr/share/dict/words* znajduje się plik zawierający treść drugiego wydania słownika Webstera rozmieszczoną po jednym słowie na wiersz. Za pomocą metody `Files.lines` można wczytać plik i utworzyć strumień łańcuchów zawierających te słowa. W tym przypadku strumień będzie zawierał wszystkie słowa ze słownika Webstera.

*Listing 4.18. Wczytanie pliku słownika do struktury Map*

```
System.out.println("\nLiczba słów o każdej długości:");
try (Stream<String> lines = Files.lines(dictionary)) {
    lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()))
       .forEach((len, num) -> System.out.printf("%d: %d%n", len, num));
} catch (IOException e) {
    e.printStackTrace();
}
```

Szczegółowy opis tego przykładu znajduje się w recepturze 7.1, a poniżej przedstawiam zwięzłe podsumowanie:

- Plik zostaje wczytany wewnątrz bloku `try` z zasobami. Interfejs `Stream` implementuje interfejs `AutoCloseable`, dzięki czemu w chwili, gdy kończy się blok `try`, Java wywołuje metodę `close` interfejsu `Stream`, która następnie wywołuje metodę `close` interfejsu `File`.
- Filtr przepuszcza do dalszego przetwarzania tylko słowa zawierające przynajmniej 20 znaków.
- Metoda `groupingBy` z klasy `Collectors` jako pierwszy argument pobiera obiekty typu `Function` reprezentujący klasyfikator. W tym przypadku klasyfikatorem jest długość każdego łańcucha. Jeśli metodzie tej zostanie przekazany tylko jeden argument, zwróci ona strukturę `Map`, w której klucze będą wartościami klasyfikatora, a wartości będą listami elementów spełniających kryteria klasyfikatora. W aktualnie omawianym przypadku efektem wywołania metody `groupingBy` ↪`(String::length)` byłaby struktura `Map<Integer,List<String>>`, w której klucze byłyby długościami słów, a wartości listami słów o tej długości.
- W tym przypadku dwuargumentowej wersji metody `groupingBy` można przekazać drugi kolektor, zwany **kolektorem strumieniowym** (ang. *downstream collector*), który dokona przetwarzania

końcowego listy słów. Tutaj typem zwrotnym jest `Map<Integer,Long>`, gdzie klucze są długościami słów, a wartości — liczbami słów o takiej długości w słowniku.

Oto wynik działania omawianego programu:

```
Liczba słów o każdej długości:
21: 82
22: 41
23: 17
24: 5
```

Innymi słowy, słownik zawiera 82 słowa o długości 21 liter, 41 słów o długości 22 liter, 17 słów o długości 23 liter oraz 5 słów o długości 24 liter[3].

W wynikach widać, że zawartość słownika została wydrukowana w kolejności rosnącej według długości słów. Aby zmienić kolejność na malejącą, należy użyć metody `Map.Entry.comparingByKey` (listing 4.19).

*Listing 4.19. Sortowanie słownika według kluczy*

```
System.out.println("\nLiczba słów o każdej długości (kolejność malejąca):");
try (Stream<String> lines = Files.lines(dictionary)) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()));

    map.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
        .forEach(e -> System.out.printf("Liczba słów o dlugości %d: %2d%n",
            e.getKey(), e.getValue()));
} catch (IOException e) {
    e.printStackTrace();
}
```

Po obliczeniu `Map<Integer,Long>` operacja pobiera `entrySet` i tworzy strumień. Metoda `sorted` interfejsu `Stream` tworzy posortowany strumień za pomocą otrzymanego komparatora.

W tym przypadku metoda `Map.Entry.comparingByKey` generuje komparator sortujący według kluczy, a używając wersji przyjmującej komparator, można zaznaczyć, że elementy mają być posortowane w odwrotnej kolejności.

Metoda `sorted` interfejsu `Stream` tworzy nowy posortowany strumień i w żaden sposób nie modyfikuje źródła danych. Pierwotny słownik pozostaje więc nienaruszony.

Oto wynik działania omawianego programu:

```
Liczba słów o każdej długości (kolejność malejąca):
Liczba słów o długości 24: 5
Liczba słów o długości 23: 17
Liczba słów o długości 22: 41
Liczba słów o długości 21: 82
```

Pozostałych metod sortowania wymienionych w tabeli 4.1 używa się w podobny sposób.

[3] Nawiasem mówiąc, pięć najdłuższych angielskich słów to: *formaldehydesulphoxylate*, *pathologicopsychological*, *scientificophilosophical*, *tetraiodophenolphthalein* i *thyroparathyroidectomize*. Powodzenia w nauce sposobu ich pisania.

## Zobacz również

Dodatkowy przykład sortowania słownika według kluczy lub wartości znajduje się w dodatku A. Kolektory strumieniowe opisałem w recepturze 4.6. Natomiast operacje plikowe dotyczące słowniku Webstera są jednym z tematów poruszonych w recepturze 7.1.

# 4.5 Dzielenie i grupowanie

## Problem

Chcemy podzielić kolekcję elementów na kategorie.

## Rozwiązanie

Metoda `Collectors.partitioningBy` dzieli elementy na te, które spełniają warunek predykatu, i te, które go nie spełniają. Metoda `Collectors.groupingBy` tworzy słownik kategorii, w którym wartości są elementami należącymi do poszczególnych kategorii.

## Omówienie

Powiedzmy, że mamy kolekcję łańcuchów. Aby je podzielić na łańcuchy o parzystej i nieparzystej liczbie znaków, można posłużyć się metodą `Collectors.partitioningBy` w sposób pokazany na listingu 4.20.

*Listing 4.20. Dzielenie łańcuchów według parzystości liczby znaków*

```
List<String> strings = Arrays.asList("to", "jest", "długa", "lista", "łańcuchów", "do",
        "użycia", "w", "ramach", "przykładu");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0)); ❶

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n", key, value));
//
// false: [długa, lista, łańcuchów, w, przykładu]
// true: [to, jest, do, użycia, ramach]
```

❶ Oddzielanie parzystych od nieparzystych

Oto sygnatury obu metod `partitioningBy`:

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
    Predicate<? super T> predicate)
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

Typy zwrotne z powodu zawartości typów generycznych wyglądają dość nieatrakcyjnie, ale w praktyce rzadko trzeba się nimi posługiwać w sposób bezpośredni. Najczęściej wyniki tych operacji są argumentem metody `collect`, która przy użyciu wygenerowanego kolektora tworzy wyjściowy słownik zdefiniowany przez trzeci generyczny argument.

Pierwsza metoda `partitioningBy` przyjmuje jako argument jeden predykat. Dzieli elementy na te, które spełniają warunek predykatu, i te, które tego warunku nie spełniają. W wyniku zawsze zwraca słownik zawierający dokładnie dwie pozycje: listę wartości spełniających warunek predykatu i listę wartości niespełniających warunku predykatu.

Przeciążona wersja omawianej metody przyjmuje dodatkowy argument typu `Collector`, który nazywany jest **kolektorem strumieniowym** (ang. *downstream collector*). Za jego pomocą można wykonać dowolne operacje na listach zwróconych w wyniku podziału, jak opisałem to w recepturze 4.6.

Metoda `groupingBy` działa podobnie jak instrukcja `group by` języka SQL: zwraca słownik, w którym klucze są grupami, a wartości listami elementów należących do tych grup.

Jeśli dane pochodzą z bazy danych, to zdecydowanie lepiej jest je pogrupować w bazie. Nowe metody dodane do API stanowią tylko udogodnienie do pracy z danymi znajdującymi się w pamięci.

Oto sygnatura metody `groupingBy`:

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
    Function<? super T,? extends K> classifier)
```

Argument `Function` przyjmuje każdy element strumienia i wydobywa własność, według której ma zostać wykonane grupowanie. Tym razem zamiast dzielić łańcuchy na dwie kategorie podzielimy je według długości (listing 4.21).

*Listing 4.21. Grupowanie łańcuchów według długości*

```
List<String> strings = Arrays.asList("to", "jest", "długa", "lista", "łańcuchów", "do",
        "użycia", "w", "ramach", "przykładu");

Map<Integer, List<String>> lengthMap = strings.stream()
    .collect(Collectors.groupingBy(String::length)); ❶

lengthMap.forEach((k,v) -> System.out.printf("%d: %s%n", k, v));
//
// 1: [w]
// 2: [to, do]
// 4: [jest]
// 5: [długa, lista]
// 6: [użycia, ramach]
// 9: [łańcuchów, przykładu]
```

❶ Grupowanie łańcuchów według długości

Klucze w powstałym słowniku są długościami łańcuchów (`1`, `2`, `4`, `5`, `6` i `9`), a wartości są listami łańcuchów o danej długości.

## Zobacz również

W recepturze 4.6 opisałem rozszerzoną wersję tej receptury. Można się z niej dowiedzieć, jak przetworzyć listy zwrócone przez operacje `groupingBy` i `partitioningBy`.

# 4.6 Kolektory strumieniowe

## Problem

Chcemy przetworzyć kolekcje zwrócone przez metody `groupingBy` i `partitioningBy`.

## Rozwiązanie

Należy użyć jednej ze statycznych metod pomocniczych z klasy `java.util.stream.Collectors`.

## Omówienie

W recepturze 4.5 pokazałem, jak podzielić elementy na kategorie. Metody `partitioningBy` i `groupingBy` zwracają słownik, w którym kluczami są kategorie (logiczne wartości `true` i `false` w przypadku `partitioningBy`, obiekty w przypadku `groupingBy`), a wartościami — listy elementów należących do tych kategorii elementów. Przypomnij sobie przykład dotyczący dzielenia łańcuchów na te o parzystej i nieparzystej liczbie znaków, który był pokazany na listingu 4.20 i dla wygody przedstawiam go jeszcze raz na listingu 4.22.

*Listing 4.22. Dzielenie łańcuchów według parzystości liczby znaków*

```
List<String> strings = Arrays.asList("to", "jest", "długa", "lista", "łańcuchów", "do",
        "użycia", "w", "ramach", "przykładu");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0));

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n", key, value));
//
// false: [długa, lista, łańcuchów, w, przykładu]
// true: [to, jest, do, użycia, ramach]
```

Dla programisty od list bardziej interesujące może być to, ile elementów trafiło do każdej kategorii. Innymi słowy, zamiast tworzyć słownik, którego wartości są typu `List<String>`, można tylko sprawdzić, ile elementów znajduje się w każdej liście. Istnieje przeciążona wersja metody `partitioningBy`, której drugi argument jest typu `Collector`:

```
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

Tutaj przydatna będzie statyczna metoda `Collectors.counting`. Na listingu 4.23 można obejrzeć ją w akcji.

*Listing 4.23. Liczenie łańcuchów w grupach*

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0,
              Collectors.counting())); ❶

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n", k, v));
//
// false: 5
// true: 5
```

❶ Kolektor strumieniowy

Taka metoda nazywa się **kolektorem strumieniowym**, ponieważ przetwarza listy wynikowe w strumieniu (a nie po zakończeniu operacji dzielenia).

Metoda `groupingBy` także występuje w wersji przyjmującej kolektor strumieniowy:

```
/**
 * @param <T> typ elementów wejściowych
 * @param <K> typ kluczy
 * @param <A> typ akumulacji pośredniej kolektora strumieniowego
 * @param <D> typ wyniku redukcji strumieniowej
 * @param classifier funkcja klasyfikacji odwzorowująca elementy wejściowe na klucze
 * @param downstream {@code Collector} implementujący redukcję strumieniową
 * @return {@code Collector} implementujący kaskadową operację grupowania według kryterium
 */
static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
    Function<? super T,? extends K> classifier,
    Collector<? super T,A,D> downstream)
```

Na początku znajduje się komentarz Javadoc, z którego można się dowiedzieć, że `T` jest typem elementu w kolekcji, `K` jest typem klucza wynikowego słownika, `A` jest akumulatorem, a `D` jest typem kolektora strumieniowego.

Znak `?` oznacza „nieznany”. Więcej informacji na temat typów generycznych w Javie 8 znajduje się w dodatku A.

Niektóre metody interfejsu `Stream` mają odpowiedniki w klasie `Collectors`. Tabela 4.2 zawiera zestawienie obu grup metod.

*Tabela 4.2. Metody klasy Collectors podobne do metod interfejsu Stream*

| Stream | Collectors |
|---|---|
| `count` | `counting` |
| `map` | `mapping` |
| `min` | `minBy` |
| `max` | `maxBy` |
| `IntStream.sum` | `summingInt` |
| `DoubleStream.sum` | `summingDouble` |
| `LongStream.sum` | `summingLong` |
| `IntStream.summarizing` | `summarizingInt` |
| `DoubleStream.summarizing` | `summarizingDouble` |
| `LongStream.summarizing` | `summarizingLong` |

Powtórzę, że zadaniem kolektora strumieniowego jest przetwarzanie kolekcji obiektów utworzonej przez wcześniejszą operację strumieniową, na przykład dzielącą lub grupującą.

## Zobacz również

W recepturze 7.1 pokazany jest przykład użycia kolektora strumieniowego do znajdowania najdłuższych słów w słowniku. W recepturze 4.5 znajduje się dokładniejszy opis metod `partitionBy` i `groupingBy`. Typy generyczne opisałem w dodatku A.

# 4.7 Znajdowanie najmniejszej i największej wartości

## Problem

Chcemy znaleźć największą lub najmniejszą wartość w strumieniu.

## Rozwiązanie

Do wyboru jest kilka możliwości: metody `maxBy` i `minBy` z interfejsu `BinaryOperator`, metody `max` i `min` z interfejsu `Stream` oraz metody pomocnicze `maxBy` i `minBy` z klasy `Collectors`.

## Omówienie

Interfejs `BinaryOperator` to jeden z funkcyjnych interfejsów w pakiecie `java.util.function`. Stanowi rozszerzenie interfejsu `BiFunction` i ma zastosowanie, gdy oba argumenty funkcji i wartość zwrotna są obiektami tej samej klasy.

Interfejs `BinaryOperator` dodaje dwie statyczne metody:

```
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

Każda z nich zwraca `BinaryOperator` używający podanego komparatora.

Aby poznać różne sposoby znajdowania największej wartości w strumieniu, przyjrzymy się zwykłej klasie `Employee` zawierającej trzy atrybuty: `name`, `salary` i `department` (listing 4.24).

*Listing 4.24. Klasa Employee*

```
public class Employee {
    private String name;
    private Integer salary;
    private String department;
    // ...inne metody...
}

List<Employee> employees = Arrays.asList(❶
        new Employee("Cersei", 250_000, "Lannister"),
        new Employee("Jamie", 150_000, "Lannister"),
        new Employee("Tyrion", 1_000, "Lannister"),
        new Employee("Tywin", 1_000_000, "Lannister"),
        new Employee("Jon Snow", 75_000, "Stark"),
        new Employee("Robb", 120_000, "Stark"),
        new Employee("Eddard", 125_000, "Stark"),
        new Employee("Sansa", 0, "Stark"),
        new Employee("Arya", 1_000, "Stark"));

Employee defaultEmployee = ❷
    new Employee("Mężczyzna (lub kobieta) nie ma nazwiska", 0, "Black and White");
```

❶ Kolekcja pracowników

❷ Domyślny obiekt tworzony, gdy strumień jest pusty

Mając kolekcję pracowników, strumień można poddać działaniu metody `reduce`, która jako argument pobiera `BinaryOperator`. Na listingu 4.25 pokazałem, jak znaleźć pracownika z najwyższym wynagrodzeniem.

*Listing 4.25. Przykład użycia metody BinaryOperator.maxBy*

```
Optional<Employee> optionalEmp = employees.stream()
    .reduce(BinaryOperator.maxBy(Comparator.comparingInt(Employee::getSalary)));

System.out.println("Najwięcej zarabia: " +
    optionalEmp.orElse(defaultEmployee));
```

Metoda `reduce` wymaga w argumencie operatora `BinaryOperator`. Statyczna metoda `maxBy` generuje ten operator na podstawie otrzymanego komparatora, który w tym przypadku porównuje pracowników według wysokości zarobków.

To rozwiązanie jest skuteczne, ale istnieje też metoda pomocnicza o nazwie `max`, którą można zastosować bezpośrednio do strumienia:

```
Optional<T> max(Comparator<? super T> comparator)
```

Na listingu 4.26 pokazany jest przykład bezpośredniego użycia tej metody.

*Listing 4.26. Przykład użycia metody Stream.max*

```
optionalEmp = employees.stream()
        .max(Comparator.comparingInt(Employee::getSalary));
```

Wynik jest taki sam.

Należy pamiętać, że w strumieniach typów podstawowych (`IntStream`, `LongStream` i `DoubleStream`) dostępna jest też metoda o nazwie `max`, która nie przyjmuje żadnych argumentów. Na listingu 4.27 znajduje się przykład jej użycia.

*Listing 4.27. Szukanie najwyższych zarobków*

```
OptionalInt maxSalary = employees.stream()
        .mapToInt(Employee::getSalary)
        .max();
System.out.println("Najwyższe zarobki to " + maxSalary);
```

W tym przypadku metoda `mapToInt` konwertuje strumień pracowników na strumień liczb całkowitych, wywołując metodę `getSalary`, a zwrócony strumień jest typu `IntStream`. Następnie metoda `max` zwraca `OptionalInt`.

W klasie pomocniczej `Collectors` dostępna jest też statyczna metoda o nazwie `maxBy`. W omawianym przypadku można jej użyć bezpośrednio (listing 4.28).

*Listing 4.28. Przykład użycia metody Collectors.maxBy*

```
optionalEmp = employees.stream()
    .collect(Collectors.maxBy(Comparator.comparingInt(Employee::getSalary)));
```

To rozwiązanie jest jednak nieporęczne i można je zastąpić użyciem metody `max` z interfejsu `Stream`, jak pokazałem w poprzednim przykładzie. Metoda `maxBy` z klasy `Collectors` może być z powodzeniem stosowana jako kolektor strumieniowy (tzn. do przetwarzania końcowego po operacji grupowania lub partycjonowania). W widocznym na listingu 4.29 kodzie została użyta metoda `groupingBy` z interfejsu `Stream` w celu utworzenia słownika kojarzącego działy z pracownikami, a następnie został znaleziony pracownik o najwyższej pensji w każdym dziale.

*Listing 4.29. Użycie metody Collectors.maxBy jako kolektora strumieniowego*

```
Map<String, Optional<Employee>> map = employees.stream()
    .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.maxBy(
                    Comparator.comparingInt(Employee::getSalary))));
map.forEach((house, emp) ->
        System.out.println(house + ": " + emp.orElse(defaultEmployee)));
```

Metoda `minBy` w każdej z tych klas działa tak samo.

### Zobacz również

Funkcje opisałem w recepturze 2.4. Opis kolektorów strumieniowych znajduje się w recepturze 4.6.

## 4.8 Tworzenie niezmiennych kolekcji

### Problem

Chcemy utworzyć listę, zbiór lub słownik bez możliwości zmiany, wykorzystując do tego celu API `Stream`.

### Rozwiązanie

Należy użyć nowej statycznej metody `collectingAndThen` z klasy `Collectors`.

### Omówienie

Kładące nacisk na przetwarzanie równoległe i klarowność kodu, techniki programowania funkcyjnego zachęcają do stosowania wszędzie tam, gdzie to możliwe, obiektów niezmiennych. Dodana w Javie 1.2 biblioteka `Collections` zawsze zawierała metody umożliwiające tworzenie niezmiennych kolekcji z istniejących struktur, choć robiło się to w dość niezgrabny sposób.

Klasa pomocnicza `Collections` zawiera metody `unmodifiableList`, `unmodifiableSet` oraz `unmodifiableMap` (i jeszcze kilka innych z przedrostkiem `unmodifiable`), których sygnatury są przedstawione na listingu 4.30.

*Listing 4.30. Metody do tworzenia niezmiennych kolekcji z klasy Collections*

```
static <T> List<T> unmodifiableList(List<? extends T> list)
static <T> Set<T> unmodifiableSet(Set<? extends T> s)
static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
```

Każda z tych metod przyjmuje jako argument listę, zbiór lub słownik i zwraca listę, zbiór lub słownik o takiej samej zawartości jak argument, ale różniące się od oryginału tym, że próba wywołania na nich jakiejkolwiek metody mogącej je zmodyfikować, na przykład `add` albo `remove`, powoduje wyjątek `UnsupportedOperationException`.

Jeśli programista korzystający z Javy w wersji starszej od 8 miał do zagospodarowania indywidualne wartości przekazane przy użyciu zmiennej listy argumentów, listę lub zbiór bez możliwości modyfikacji tworzył w sposób pokazany na listingu 4.31.

*Listing 4.31. Tworzenie niezmiennych list lub zbiorów przed Javą 8*

```
@SafeVarargs ❶
public final <T> List<T> createImmutableListJava7(T... elements) {
    return Collections.unmodifiableList(Arrays.asList(elements));
}

@SafeVarargs ❶
public final <T> Set<T> createImmutableSetJava7(T... elements) {
    return Collections.unmodifiableSet(new HashSet<>(Arrays.asList(elements)));
}
```

❶ Programista obiecuje, że nie uszkodzi wejściowego typu tablicowego (szczegółowe informacje znajdują się w dodatku A)

W obu przypadkach wartości przychodzące są przyjmowane i dodawane do listy. Otrzymaną w ten sposób strukturę można spakować za pomocą metody `unmodifiableList` lub, w przypadku zbioru, przekazać jako argument do konstruktora zbioru, a następnie do metody `unmodifiableSet`.

W Javie 8 nowy interfejs API `Stream` udostępnia metodę `Collectors.collectingAndThen`, która została użyta w przykładzie przedstawionym na listingu 4.32.

*Listing 4.32. Tworzenie niezmiennych list i zbiorów w Javie 8*

```
import static java.util.stream.Collectors.collectingAndThen;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

// ...definicja klasy zawierającej następujące metody...

@SafeVarargs
public final <T> List<T> createImmutableList(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toList(),
                    Collections::unmodifiableList)); ❶
}

@SafeVarargs
public final <T> Set<T> createImmutableSet(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toSet(),
                   Collections::unmodifiableSet)); ❶
}
```

❶ „Wykańczacz” pakuje wygenerowane kolekcje

Metoda `Collectors.collectingAndThen` przyjmuje dwa argumenty: kolektor strumieniowy i funkcję zwaną **wykańczaczem** (ang. *finisher*). Cała idea polega na strumieniowym przekazaniu elementów wejściowych i zgromadzeniu ich w liście lub zbiorze oraz przekazaniu tak otrzymanej struktury do metody opakowującej w kolekcję niezmienną.

Konwersja szeregu elementów wejściowych w niezmienny słownik jest nieco bardziej skomplikowana. Po części wynika to z tego, że nie wiadomo, które elementy wejściowe mają pełnić rolę kluczy, a które — wartości. Na listingu 4.33[4] przedstawiłem przykład utworzenia niezmiennego słownika w bardzo niezgrabny sposób przy użyciu inicjalizatora obiektu.

[4] Źródło: wpis na blogu Carla Martensena pt. *Java 9's Immutable Collections Are Easier To Create But Use With Caution*.

*Listing 4.33. Tworzenie niezmiennego słownika*

```
Map<String, Integer> map = Collections.unmodifiableMap(
  new HashMap<String, Integer>() {{
    put("have", 1);
    put("the", 2);
    put("high", 3);
    put("ground", 4);
}});
```

Ci, którzy znają już Javę 9, wiedzą jednak, że całą tę recepturę można zamienić na bardzo prosty zestaw metod fabrycznych: `List.of`, `Set.of` i `Map.of`.

### Zobacz również

W recepturze 10.3 znajdują się przykłady użycia metod fabrycznych Javy 9, które automatycznie tworzą niezmienne kolekcje.

## 4.9 Implementowanie interfejsu Collector

### Problem

Trzeba ręcznie zaimplementować interfejs `java.util.stream.Collector`, ponieważ żadna z metod fabrycznych w klasie `java.util.stream.Collectors` nie spełnia wszystkich oczekiwań.

### Rozwiązanie

Należy dostarczyć wyrażenia lambda lub utworzyć referencje do metod dla funkcji dostawy, akumulacji, kombinacji i wykańczania używanych przez metody fabryczne `Collector.of` wraz z wszystkimi wymaganymi cechami.

### Omówienie

Klasa pomocnicza `java.util.stream.Collectors` zawiera kilka metod statycznych zwracających obiekt typu `Collector`, np. `toList`, `toSet`, `toMap` i `toCollection`. Przykład użycia każdej z nich można znaleźć w różnych miejscach w tej książce. Egzemplarze klas implementujących interfejs `Collector` są wysyłane jako argumenty do metody `collect` interfejsu `Stream`. Na przykład na listingu 4.34 metoda przyjmuje argumenty łańcuchowe i zwraca listę zawierającą tylko te łańcuchy, których długość jest liczbą parzystą.

*Listing 4.34. Zwrócenie listy za pomocą metody collect*

```
public List<String> evenLengthStrings(String... strings) {
    return Stream.of(strings)
        .filter(s -> s.length() % 2 == 0)
        .collect(Collectors.toList()); ❶
}
```

❶ Zebranie łańcuchów o parzystej długości w liście

Jeśli jednak konieczne jest napisanie własnych kolektorów, procedura staje się nieco bardziej skomplikowana. Kolektory korzystają z pięciu funkcji, które współpracują ze sobą w celu zakumulowania elementów w zmiennym kontenerze i ewentualnie przekształcenia wyniku. Te pięć funkcji to dostawca (`supplier`), akumulator (`accumulator`), kombinator (`combiner`), wykańczacz (`finisher`) oraz cechy (`characteristics`).

Zacznę od funkcji `characteristics`, która reprezentuje niezmienny zbiór `Set` elementów typu wyliczeniowego `Collector.Characteristics`. Trzy możliwe wartości to `CONCURRENT`, `IDENTITY_FINISH` oraz `UNORDERED`. Wartość `CONCURRENT` oznacza, że funkcja akumulacyjna może być wywoływana współbieżnie w wielu wątkach na kontenerze wynikowym. Wartość `UNORDERED` oznacza, że operacja wykonywana na kolekcji nie musi zachowywać takiej kolejności elementów, w jakiej je napotkała. Z kolei wartość `IDENTITY_FINISH` sygnalizuje, że funkcja wykańczająca zwraca swój argument bez żadnych zmian.

Jeśli domyślne ustawienia są właściwe, nie ma potrzeby definiowania żadnych cech.

Przeznaczenie każdej z wymaganych metod jest następujące:

`supplier()`

: Tworzy kontener akumulacyjny przy użyciu `Supplier<A>`.

`accumulator()`

: Dodaje pojedynczy element do kontenera akumulacyjnego przy użyciu `BiConsumer<A,T>`.

`combiner()`

: Łączy dwa kontenery akumulacyjne przy użyciu `BinaryOperator<A>`.

`finisher()`

: Przekształca kontener akumulacyjny w kontener wynikowy przy użyciu `Function<A,R>`.

`characteristics()`

: Wartość `Set<Collector.Characteristics>` wybrana z wyliczenia.

Oczywiście jak zwykle wszystko jest prostsze, gdy zna się funkcyjne interfejsy zdefiniowane w pakiecie `java.util.function`. Dostawca tworzy kontener, w którym są gromadzone tymczasowe wyniki. Operacja `BiConsumer` dodaje jeden element do akumulatora. `BinaryOperator` oznacza, że zarówno typ wejściowy, jak i typ wyjściowy są takie same, a w tym przypadku chodzi o połączenie dwóch akumulatorów w jeden. Na koniec funkcja typu `Function` przekształca akumulator do ostatecznej postaci.

Każda z tych metod jest wywoływana w procesie gromadzenia elementów, który jest uruchamiany na przykład przez metodę `collect` z interfejsu `Stream`. Pod względem koncepcyjnym proces gromadzenia elementów jest równoważny z generycznym kodem pokazanym na listingu 4.35, zaczerpniętym z dokumentacji Javy.

*Listing 4.35. Sposób użycia metod z interfejsu Collector*

```
R container = collector.supplier.get(); ❶
for (T t : data) {
    collector.accumulator().accept(container, t); ❷
}
return collector.finisher().apply(container); ❸
```

❶ Utworzenie kontenera akumulacyjnego

❷ Dodanie elementów do kontenera akumulacyjnego

❸ Konwersja kontenera akumulacyjnego na wynikowy przy użyciu metody wykańczającej

Zastanawiający w tym przykładzie jest brak choćby wzmianki o funkcji `combiner`. Wynika to z faktu, że podczas pracy ze strumieniami sekwencyjnymi funkcja ta jest niepotrzebna — algorytm działa zgodnie z opisem. Kiedy jednak strumień jest przetwarzany techniką równoległą, to praca zostaje podzielona na kilka części, z których każda dostarczy własny kontener akumulacyjny. Wówczas za pomocą funkcji `combiner` łączy się te wszystkie cząstkowe kontenery w jeden gotowy do przekazania do funkcji wykańczającej.

Na listingu 4.36 znajduje się przykład kodu podobny do przykładu z listingu 4.34.

*Listing 4.36. Generowanie niezmiennego zbioru SortedSet przy użyciu funkcji collect*

```
public SortedSet<String> oddLengthStringSet(String... strings) {
        Collector<String, ?, SortedSet<String>> intoSet =
                Collector.of(TreeSet<String>::new, ❶
                        SortedSet::add, ❷
                        (left, right) -> { ❸
                              left.addAll(right);
                              return left;
                        },
                        Collections::unmodifiableSortedSet); ❹
        return Stream.of(strings)
                .filter(s -> s.length() % 2 != 0)
                .collect(intoSet);
}
```

❶ Dostawca tworzący nowy zbiór `TreeSet`

❷ `BiConsumer` dodający każdy łańcuch do zbioru `TreeSet`

❸ `BinaryOperator` łączący dwa egzemplarze zbioru `SortedSet` w jeden

❹ Funkcja wykańczająca, która tworzy niezmienny zbiór

W wyniku powstanie niezmienny zbiór łańcuchów posortowanych w kolejności leksykograficznej.

W tym przykładzie została użyta jedna z dwóch wersji metody `static of` służącej do tworzenia kolektorów. Sygnatury obu tych wersji wyglądają następująco:

```
static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier,
    BiConsumer<A,T> accumulator,
    BinaryOperator<A> combiner,
    Function<A,R> finisher,
    Collector.Characteristics... characteristics)
static <T,R> Collector<T,R,R> of(Supplier<R> supplier,
    BiConsumer<R,T> accumulator,
    BinaryOperator<R> combiner,
    Collector.Characteristics... characteristics)
```

Dzięki dostępności w klasie `Collectors` metod pomocniczych do tworzenia kolektorów bardzo rzadko zachodzi potrzeba tworzenia własnego kolektora w pokazany sposób. Z drugiej strony warto wiedzieć, jak się to robi, a poza tym przy tej okazji można było po raz kolejny przyjrzeć się, jak funkcyjne interfejsy z pakietu `java.util.function` we współpracy tworzą interesujące obiekty.

## Zobacz również

Funkcja `finisher` jest jednym z kolektorów strumieniowych, które zostały opisane w recepturze 4.6. Funkcyjne interfejsy `Supplier`, `Function` i `BinaryOperator` są opisane w kilku recepturach w rozdziale 2. Statyczne metody pomocnicze klasy `Collectors` omówiłem w recepturze 4.2.

# Skorowidz

## H



## I



## J



## K

## L



## Ł



## M

## N



## O



## P

## R



## S

## T



## V



## W



## Z

**Notatki**

O'REILLY®

# Java to nowoczesność, prostota i elegancja rozwiązań!

Wprowadzenie do języka Java elementów programowania funkcyjnego, takich jak wyrażenia lambda, referencje do metod czy strumienie, całkowicie odmieniło technikę pracy. Warto je sobie przyswoić, gdyż podejście funkcyjne sprawia, że pisany kod jest prostszy i czytelniejszy, łatwiejsze też się staje uzyskanie współbieżności. Przeczytaj ten przewodnik i dowiedz się, jak nowe funkcyjne idiomy zmieniły sposób pisania kodu źródłowego.

Znajdziesz tu niemal kompletny opis Javy SE 8 i informację o planowanych nowościach w Javie 9. Poszczególne zagadnienia zostały zilustrowane praktycznymi recepturami wraz z komentarzami. W ten sposób pokazano, jak najnowsze elementy Javy ułatwiają proste rozwiązywanie dość złożonych problemów. Poza zaprezentowaniem kluczowych koncepcji, takich jak wyrażenia lambda czy pojęcie interfejsu funkcyjnego, omówiono bardziej problematyczne zagadnienia: typ *Optional*, strumienie wejścia i wyjścia, pakiet *java.time* czy współbieżność i równoległość. Nie zabrakło oczywiście licznych, łatwych do przeanalizowania i zrozumienia przykładów kodu.

W tej książce między innymi:

- podstawowe pojęcia programowania funkcyjnego
- sortowanie strumieni danych
- lenistwo, odroczone wykonywanie i kompozycje domknięć
- mapowanie i mapowanie płaskie
- programowanie funkcyjne a praca z plikami i katalogami

**Dr Ken Kousen** – jest doświadczonym programistą i szkoleniowcem. Kieruje własną firmą i prowadzi techniczne kursy na temat Javy, Androida, Spring, Hibernate, Groovy i Grails. Regularnie występuje podczas cyklicznej konferencji **No Fluff Just Stuff**, która dotyczy technicznych aspektów programowania w Javie. Uzyskał szereg certyfikatów technicznych i ukończył kilka kierunków studiów. Od czasu do czasu wykłada na politechnice Rensselaer w Hartford.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-4073-2
9 788328 340732

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 54,90 zł