Zaawansowane programowanie w Javie

Studia zaoczne – Wykład 6

dr hab. Andrzej Zbrzezny, profesor UJD

Katedra Matematyki i Informatyki
Uniwersytet Jana Długosza w Częstochowie

27 kwietnia 2024

Zalecana Literatura

Literatura podstawowa

- Cay Horstmann.
  Java. Przewodnik doświadczonego programisty. Wydanie III.
  Wydawnictwo Helion. Gliwice, październik 2023.
- Joshua Bloch.
  Java. Efektywne programowanie. Wydanie III.
  Wydawnictwo Helion. Gliwice, sierpień 2018.
- Cay Horstmann.
  Java. Podstawy. Wydanie XII.
  Wydawnictwo Helion. Gliwice, grudzień 2022.
- https://dev.java/

Tworzenie map

- Załóżmy, że mamy strumień `people` typu `Stream<Person>` i chcemy zebrać elementy w mapę, aby później mieć możliwość wyszukiwania ludzi po ich identyfikatorach.
- Metoda `Collectors.toMap` ma dwa argumenty funkcyjne, które tworzą klucze i wartości mapy. Przykładowo:

```
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::getId, Person::getName));
```

- W typowym przypadku, gdy wartościami powinny być rzeczywiste elementy, jako drugiej funkcji należy użyć `Function.identity()`:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::getId, Function.identity()));
```

Tworzenie map

- Jeżeli istnieje więcej niż jeden element z tym samym kluczem, pojawia się konflikt i wyrzucony zostaje wyjątek `IllegalStateException`.
- Można przesłonić to zachowanie, dodając trzeci argument funkcji rozwiązujący konflikt i ustalający wartość dla klucza na podstawie istniejącej i nowej wartości.
- Funkcja może zwrócić istniejącą wartość, nową wartość lub kombinację wartości.
- Trójargumetowa metoda `Collectors.toMap`:

```
public static <T, K, U> Collector<T, ?, Map<K, U>> toMap(
    Function<? super T, ? extends K> keyMapper,
    Function<? super T, ? extends U> valueMapper,
    BinaryOperator<U> mergeFunction
)
```

Tworzenie map

- Poniżej tworzymy mapę, która obejmuje klucz dla każdego z dostępnych języków, zawierający jego nazwę w języku domyślnym (na przykład „niemiecki”) oraz jego oryginalną nazwę jako wartość (na przykład „Deutsch ”).

```
Stream<Locale> locales =
    Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
    Collectors.toMap(
        Locale::getDisplayCountry,
        Locale::getDisplayLanguage,
        (existingValue, newValue) -> existingValue)
    );
```

- Nie przeszkadza nam to, że język może pojawić się dwukrotnie (na przykład niemiecki dla Niemiec i Szwajcarii) – po prostu wykorzystujemy pierwsze wystąpienie.

Tworzenie map

- Załóżmy teraz, że chcemy znać wszystkie języki z wybranego kraju. Będziemy potrzebowali mapy postaci: `Map<String, Set<String>>`.
- Przykładowo, wartością dla „Switzerland” będzie zbiór: `[French, German, Italian]`.
- Na początku dla każdego języka tworzymy zbiór będący singletonem. Gdy dla danego kraju pojawia się kolejny język, tworzymy połączenie istniejącego i nowego zbioru.

```
Map<String, Set<String>> countryLanguageSets = locales
    .collect(Collectors.toMap(Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // Suma zbiorów a oraz b
            Set<String> result = new HashSet<>(a);
            result.addAll(b);
            return result; }));
```

Tworzenie map

- Jeżeli potrzebujemy uzyskać obiekt typu `TreeMap`, to należy przekazać jako czwarty argument konstruktor do klasy `TreeMap`.
- Należy dostarczyć też funkcję łączącą.
- Oto przykład, w którym zwracany jest obiekt klasy `TreeMap`:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person::getId,
        Function.identity(),
        (existingValue, newValue) ->
            { throw new IllegalStateException(); },
        TreeMap::new));
```

Strumienie

Tworzenie map

- Każda z metod `toMap` ma swój odpowiednik `toConcurrentMap`, zwracający mapę do przetwarzania równoległego.
- Pojedyncza mapa tego typu jest wykorzystywana przy zrównoleglonym przetwarzaniu kolekcji.
- Gdy współdzielona mapa jest wykorzystywana ze strumieniem równoległym, jest bardziej wydajna niż mapy łączone.
- Zauważmy, że elementy nie zachowują kolejności ze strumienia, ale zazwyczaj nie ma to znaczenia.

Strumienie

Przykład (Tworzenie map)

- Program `r08/08_09/CollectingIntoMaps.java`

Zadanie

Przeanalizuj powyższy program i upewnij się, że rozumiesz wszystkie zastosowane w nim konstrukcje.

Strumienie

Grupowanie i partycjonowanie

- Dowiedzieliśmy się, w jaki sposób zebrać wszystkie języki przypisane do danego kraju.
- Było to dość mozolne. Musieliśmy utworzyć zbiór z jednym elementem dla każdej wartości mapy, a następnie określić, w jaki sposób połączyć go z istniejącą lub nową wartością.
- Tworzenie grup wartości o tych samych właściwościach zdarza się bardzo często i metoda `groupingBy` bezpośrednio wspiera ten proces.

Grupowanie i partycjonowanie

- Popatrzmy na grupowanie lokalizacji według krajów. Najpierw utwórzmy taką mapę:

```
Stream<Locale> locales =
    Stream.of(Locale.getAvailableLocales());
Map<String, List<Locale>> countryToLocales =
    locales.collect(
        Collectors.groupingBy(Locale::getCountry));
```

- Funkcja `Locale::getCountry` jest funkcją klasyfikującą grupowania. Można teraz wyszukać wszystkie lokalizacje dla danego kodu kraju, na przykład:

```
// Zwraca lokalizacje
// [wae_CH_#Latn, de_CH, pt_CH, rm_CH_#Latn, gsw_CH,
// fr_CH, rm_CH, it_CH, wae_CH, en_CH, gsw_CH_#Latn]
List<Locale> swissLocales =
    countryToLocales.get("CH");
```

Strumienie

Krótkie powtórzenie na temat lokalizacji

- Każda lokalizacja ma kod języka (`en` dla języka angielskiego) oraz kod kraju (`US` dla Stanów Zjednoczonych).
- Lokalizacja `en_US` oznacza język angielski w wersji dla Stanów Zjednoczonych, a `en_IE` język angielski w wersji dla Irlandii.
- Niektóre kraje mają wiele lokalizacji.
- Przykładowo, `ga_IE` oznacza irlandzką wersję języka celtyckiego (ang. gaelic), a jak widzieliśmy w poprzednim przykładzie, moja maszyna wirtualna zna jedenaście języków używanych w Szwajcarii.

Grupowanie i partycjonowanie

- Gdy funkcja klasyfikująca jest predykatem (czyli zwraca wartość typu **`boolean`**), elementy strumienia są dzielone na dwie listy: tych, dla których funkcja zwraca `true`, i pozostałych.
- W takim przypadku bardziej wydajne jest wykorzystanie `partitioningBy` zamiast `groupingBy`.
- Na przykład poniżej dzielimy wszystkie lokalizacje na te, które korzystają z języka angielskiego oraz pozostałe:

```
locales = Stream.of(Locale.getAvailableLocales());
Map<Boolean, List<Locale>> englishAndOtherLocales =
    locales.collect(Collectors.partitioningBy(
        loc -> loc.getLanguage().equals("en")));
```

Strumienie

Grupowanie i partycjonowanie

- Jeśli wywołujemy metodę `groupingByConcurrent`, otrzymujemy mapę, która po połączeniu z równoległym strumieniem jest wypełniana wielowątkowo.
- Jest to dokładna analogia do metody `toConcurrentMap`.

Przykład (Grupowanie i partycjonowanie)

- Program `r08/08_10/GroupingPartitioning.java`

Zadanie

Przeanalizuj powyższy program i upewnij się, że rozumiesz wszystkie zastosowane w nim konstrukcje.

Kolektory strumieniowe

- Metoda `groupingBy` zwraca mapę, której wartościami są listy. Aby dalej przetwarzać te listy, należy wykorzystać kolektory strumieniowe.
- Przykładowo, jeżeli potrzebujemy zbiorów, a nie list, możemy użyć kolektora `Collectors.toSet`:

```
var locales = Stream.of(Locale.getAvailableLocales());
Map<String, Set<Locale>> countryToLocaleSet = locales
    .collect(groupingBy(Locale::getCountry, toSet()));
```

- W tym przykładzie, tak jak w kolejnych, dla uproszczenia zapisu przyjmujemy założenie, że wykonany jest statyczny import z `java.util.stream.Collectors.*`.
- Kilka kolektorów służy do zamiany pogrupowanych elementów na liczby.

Kolektory strumieniowe

- Metoda `counting` zlicza ilości zebranych elementów. Na przykład:

```
Map<String, Long> countryToLocaleCounts =
    locales.collect(
        groupingBy(Locale::getCountry, counting()));
```

zlicza liczbę lokalizacji dla każdego kraju.

- Metody `summing(Int|Long|Double)` przyjmują jako argument funkcję, wykonują tę funkcję na elementach pobranych ze strumienia i tworzą ich sumę. Przykładowo:

```
Map<String, Integer> stateToCityPopulation =
    cities.collect(groupingBy(City::getState,
        summingInt(City::getPopulation)));
```

oblicza sumę populacji dla każdego stanu na podstawie strumienia zawierającego dane dla miast.

Kolektory strumieniowe

- Metody `maxBy` oraz `minBy`, korzystając z komparatora, zwracają największy i najmniejszy z elementów strumienia. Przykładowo,

```
Map<String, City> stateToLargestCity = cities.collect(
    groupingBy(City::getState, maxBy(
        Comparator.comparing(City::getPopulation))));
```

  zwraca największe miasta stanów.

- Metoda `mapping` uruchamia funkcję na elementach strumienia i potrzebuje dodatkowego kolektora do przetworzenia zwróconych wyników. Przykładowo,

```
Map<String, Optional<String>> stateToLongestCityName =
    cities.collect(groupingBy(City::getState,
        mapping(City::getName, maxBy(
            Comparator.comparing(String::length)))));
```

Kolektory strumieniowe

- W poprzednim przykładzie grupujemy miasta w stanach. W każdym stanie generujemy nazwy miast i wybieramy największą długość nazwy.
- Metoda `mapping` pozwala również w lepszy sposób utworzyć zbiór wszystkich języków przypisanych do kraju:

```
Map<String, Set<String>> countryToLanguages =
    locales.collect(
        groupingBy(Locale::getDisplayCountry,
            mapping(Locale::getDisplayLanguage, toSet())
        )
    );
```

- Uprzednio wykorzystaliśmy `toMap` zamiast `groupingBy`. W tej postaci nie musimy obawiać się o łączenie oddzielnych zbiorów.

Kolektory strumieniowe

- Jeżeli funkcja grupująca lub mapująca zwróciła wartość typu **`int`**, **`long`** lub **`double`**, możemy zbierać elementy w obiektach zawierających ogólne statystyki. Przykładowo:

```
Map<String, IntSummaryStatistics>
    stateToCityPopulationSummary = cities.collect(
        groupingBy(City::getState,
            summarizingInt(City::getPopulation)
        )
    );
```

  Następnie możemy pobrać sumę, liczbę, średnią, najmniejszą i największą wartość funkcji.

- Istnieją również trzy wersje metody `reducing`, które wykonują ogólne redukcje opisane w kolejnym podrozdziale.

Strumienie

Kolektory strumieniowe

- Łączenie kolektorów to potężne narzędzie, ale może również doprowadzić do bardzo złożonych wyrażeń.
- Najlepiej wykorzystać je z `groupingBy` lub `partitioningBy` do przetwarzania pochodzących ze strumienia wartości map.
- W innym przypadku należy po prostu użyć takich metod jak `map`, `reduce`, `count`, `max` czy `min` bezpośrednio na strumieniach.

Przykład (Grupowanie i partycjonowanie)

- Program `r08/08_11/DownstreamCollectors.java`

Zadanie

Przeanalizuj powyższy program i upewnij się, że rozumiesz wszystkie zastosowane w nim konstrukcje.

Operacje redukcji

- Metoda `reduce` to ogólny mechanizm pozwalający na obliczanie wartości ze strumienia. Najprostsza postać przyjmuje funkcję binarną i wykonuje ją, począwszy od dwóch pierwszych elementów. Jest to proste do wytłumaczenia, gdy funkcja jest sumą:

```
Stream<Integer> values = ...;
Optional<Integer> sum =
    values.reduce((x, y) -> x + y);
```

- W tym przypadku metoda `reduce` oblicza $v_0 + v_1 + v_2 + \ldots$, gdzie v_j to elementy ze strumienia.
- Metoda ta zwraca `Optional`, ponieważ nie ma poprawnego wyniku, jeżeli strumień jest pusty.

Operacje redukcji

- W poprzednim przykładzie zamiast `reduce((x, y) -> x + y)` można napisać `reduce(Integer::sum)`.
- Mówiąc ogólnie: jeżeli metoda `reduce` ma operator redukcji op, to redukcja zwraca $v_0\ op\ v_1\ op\ v_2\ op\ \ldots$, gdzie $v_i\ op\ v_{i+1}$ oznacza wywołanie funkcji $op(v_i, v_{i+1})$.
- Operacja powinna być łączna: nie powinno mieć znaczenia, w jakiej kolejności łączymy elementy.
- W języku matematyki $(x\ op\ y)\ op\ z$ musi być równe $x\ op\ (y\ op\ z)$. Pozwala to na wydajne redukowanie strumieni równoległych.
- Istnieje wiele operacji łącznych, które mogą być przydatne w praktyce, takich jak suma, mnożenie, łączenie ciągów znaków, obliczanie wartości maksymalnej i minimalnej, obliczanie sumy oraz przecięcia zbiorów.

Operacje redukcji

- Przykładem operacji, która nie jest łączna, jest odejmowanie. Przykładowo, $(6 - 3) - 2$ nie jest równe $6 - (3 - 2)$.
- Często można wyróżnić wartość e, która spełnia warunek $e\ op\ x = x$ – można wykorzystać ten element do rozpoczęcia obliczeń.
- Przykładowo, 0 spełnia ten warunek w przypadku dodawania.
- Następnie należy wywołać drugą postać `reduce`:

```
// Oblicza 0 + v0 + v1 + v2 + ...
Stream<Integer> values = ...;
Integer sum = values.reduce(0, (x, y) -> x + y)
```

- Wartość taka jest zwracana w przypadku, gdy strumień okaże się pusty i nie jest już konieczne korzystanie z klasy `Optional`.

Strumienie

Operacje redukcji

- Przypuśćmy teraz, że mamy strumień obiektów i musimy utworzyć sumę pewnej własności, takiej jak długości w strumieniach ciągów znaków.
- Nie można wykorzystać prostej postaci `reduce`.
- Wymaga to użycia funkcji `(T, T) -> T` z takimi samymi typami argumentów jak typ zwracanej wartości.
- W tej sytuacji mamy jednak dwa typy: elementy strumienia są typu `String`, a otrzymany wynik ma typ całkowity.
- Istnieje taka postać `reduce`, która radzi sobie z taką sytuacją.
- Najpierw dostarczamy funkcję „zbierającą”: `(total, word) -> total + word.length()`.
- Ta funkcja jest wielokrotnie wywoływana do utworzenia całkowitej sumy.

Operacje redukcji

- Jeżeli jednak obliczenia są zrównoleglane, wykonywanych będzie kilka obliczeń tego rodzaju i konieczne jest połączenie uzyskanych wyników.
- W tym celu dostarczamy drugą funkcję. Całość wywołania wygląda tak:

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```

- W praktyce prawdopodobnie nie będziemy zbyt często korzystać z metody `reduce`.
- Zazwyczaj łatwiej jest mapować strumienie liczb i korzystać z funkcji do obliczenia sumy, wartości maksymalnej lub minimalnej.

Strumienie

Operacje redukcji

- W tym konkretnym przypadku można wywołać `words.mapToInt(String::length).sum()`, która jest zarówno prostsza, jak i bardziej wydajna dzięki temu, że nie wymaga opakowywania wartości.

Przykład (Operacje redukcji)

- Program `r08/08_12/ReductionDemo.java`

Zadanie

Przeanalizuj powyższy program i upewnij się, że rozumiesz wszystkie zastosowane w nim konstrukcje.

Operacje redukcji

- Zdarza się, że metoda `reduce` nie jest wystarczająco ogólna.
- Przykładowo przypuśćmy, że chcemy zebrać wyniki w klasie `BitSet`.
- Jeżeli kolekcja jest zrównoleglana, nie można umieścić elementów bezpośrednio w jednym zbiorze typu `BitSet`, ponieważ obiekt `BitSet` nie gwarantuje bezpiecznej pracy z wieloma wątkami.
- Z tego powodu nie można korzystać z `reduce`.
- Każdy segment musi zaczynać się od pustego zbioru, a `reduce` pozwala umieścić tylko jeden element neutralny.

Operacje redukcji

- Zamiast tego należy wykorzystać `collect`. Funkcja ta pobiera trzy argumenty wskazujące funkcje:
  1. Pierwsza funkcja tworzy nowe instancje docelowego obiektu, na przykład konstruktor dla zestawu funkcji skrótu.
  2. Druga funkcja dodaje element do zbioru tak, jak metoda `add`.
  3. Trzecia funkcja łączy dwa obiekty w jeden, jak metoda `addAll`.
- Poniżej jest przykład wykorzystania metody `collect` dla zbioru bitów:

```
BitSet result = stream.collect(
    BitSet::new, BitSet::set, BitSet::or
);
```

Strumienie typów prostych

- Jak dotąd gromadziliśmy liczby całkowite w obiektach typu `Stream<Integer>`, mimo że ewidentnie nieefektywne jest opakowywanie każdej wartości obiektem.
- To samo dotyczy innych typów prostych: **`double`**, **`float`**, **`long`**, **`short`**, **`char`**, **`byte`** i **`boolean`**.
- Biblioteka strumieni ma specjalne typy `IntStream`, `LongStream` i `DoubleStream` do zapisywania wartości typów prostych bezpośrednio bez opakowywania.
- Jeżeli chcemy zapisać wartość typu **`short`**, **`char`**, **`byte`** czy **`boolean`**, to powinniśmy wykorzystać `IntStream`, zaś dla wartości typu **`float`** wykorzystać `DoubleStream`.

Strumienie typów prostych

- Aby utworzyć `IntStream`, należy wywołać metody `IntStream.of` i `Arrays.stream`:

```
IntStream stream = IntStream.of(0, 1, 1, 2, 3, 5);
// values jest tablicą typu int[]
stream = Arrays.stream(values, from, to);
```

- Tak jak w przypadku strumieni obiektów, można też wykorzystać statyczne metody `generate` i `iterate`.
- Dodatkowo `IntStream` i `LongStream` mają metody statyczne `range` i `rangeClosed` generujące zakresy kolejnych liczb całkowitych:

```
// Bez górnego ograniczenia
IntStream zeroToNinetyNine = IntStream.range(0, 100);
// Z górnym ograniczeniem
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
```

Strumienie

Strumienie typów prostych

- Interfejs `CharSequence` zawiera metody `codePoints` i `chars`, zwracające `IntStream` zawierający kody Unicode znaków lub jednostek kodowych w kodowaniu UTF-16.

```
String sentence = "\uD835\uDD46 is the set of octonions.";
// \uD835\uDD46 is the UTF-16 encoding of the letter,
// unicode U+1D546
IntStream codes = sentence.codePoints();
// The stream with hex values 1D546 20 69 73 20 ...
```

- Mając strumień obiektów można przekształcić go w strumień typów prostych za pomocą metod: `mapToInt`, `mapToLong` lub `mapToDouble`.

Strumienie typów prostych

- Przykładowo, jeżeli mamy strumień ciągów znaków i chcemy przetwarzać liczby całkowite opisujące ich długość, można wykonać to również w `IntStream`:

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

- Aby przekształcić strumień wartości typu prostego w strumień obiektów, należy wykorzystać metodę `boxed`:

```
Stream<Integer> integers =
    IntStream.range(0, 100).boxed();
```

- Klasa Random ma metody: `ints`, `longs` i `doubles`, zwracające strumienie typów prostych liczb losowych.

Strumienie typów prostych

- Generalnie metody w strumieniach typów prostych są odpowiednikami metod w strumieniach obiektów.
- Oto najbardziej zauważalne różnice:
  - Metody `toArray` zwracają tablice wartości typów prostych.
  - Metody zwracające opcjonalne wyniki zwracają `OptionalInt`, `OptionalLong` lub `OptionalDouble`. Te klasy są odpowiednikami klasy `Optional`, ale zamiast metody `get` mają metody: `getAsInt`, `getAsLong` i `getAsDouble`.
  - Istnieją metody: `sum`, `average`, `max` i `min`, zwracające sumę, wartość średnią, maksymalną i minimalną. Te metody nie są definiowane w strumieniach obiektów.
  - Metoda `summaryStatistics` zwraca obiekty typu `IntSummaryStatistics`, `LongSummaryStatistics` czy `DoubleSummaryStatistics`, które mogą równocześnie raportować sumę, średnią, wartość maksymalną i minimalną strumienia.

Strumienie

Przykład (Strumienie typów prostych)

- Program `r08/08_13/PrimitiveTypeStreams.java`

Zadanie

Przeanalizuj powyższy program i upewnij się, że rozumiesz wszystkie zastosowane w nim konstrukcje.

Strumienie równoległe

- Strumienie upraszczają zrównoleglanie wykonywania operacji na dużych zbiorach danych. Proces ten jest w dużej części automatyczny, ale należy przestrzegać kilku zasad.
- Przede wszystkim trzeba mieć strumień równoległy. Można utworzyć taki strumień z każdej kolekcji za pomocą metody `Collection.parallelStream()`:

```
Stream<String> parallelWords =
    words.parallelStream();
```

- Co więcej, metoda `parallel` może przekształcić każdy sekwencyjny strumień w strumień równoległy:

```
Stream<String> parallelWords =
    Stream.of(wordArray).parallel();
```

Strumienie

Strumienie równoległe

- Jeżeli strumień jest w trybie równoległym przy wywoływaniu metody kończącej, wszystkie pośrednie operacje na strumieniu zostaną zrównoleglone.
- Gdy operacje na strumieniu wykonywane są równolegle, celem jest to, by wynik ich działania był taki sam jak przy ich wykonaniu w jednym wątku.
- Ważne jest, by operacje były bezstanowe i można było je wykonać w dowolnej kolejności.
- Na następnym slajdzie znajduje się przykład tego, czego nie można robić.

Strumienie równoległe

- Załóżmy, że chcemy zliczyć wszystkie krótkie słowa w strumieniu ciągów znaków:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(s ->
    { if (s.length() < 12) ++shortWords[s.length()]; });
    // Errorrace condition!
System.out.println(Arrays.toString(shortWords));
```

- Jest to bardzo, bardzo zły kod. Funkcja przekazana do `forEach` działa równolegle w wielu wątkach i każdy z nich aktualizuje tę samą tablicę.
- Jest to klasyczny przykład hazardu. Jeżeli uruchomimy ten program wiele razy, bardzo prawdopodobne jest, że uzyskamy inne ciągi liczb przy kolejnych wywołaniach i żaden z nich nie będzie poprawny.

Strumienie równoległe

- Do programisty należy zapewnienie, by każdą funkcję przekazaną do strumienia równoległego można było wykonywać wielowątkowo.
- Najlepszym sposobem, by tego dokonać, jest unikanie modyfikowania stanów.
- W tym przykładzie można bezpiecznie zrównoleglić obliczenia poprzez grupowanie ciągów równej długości i ich zliczanie:

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
        .filter(s -> s.length() < 12)
        .collect(groupingBy(String::length, counting()));
```

- Domyślnie strumienie powstające z uporządkowanych kolekcji (tablic i list), zakresów, generatorów i iteratorów lub z wywołania `Stream.sorted` są uporządkowane.

Strumienie równoległe

- Wyniki są zbierane w kolejności występowania oryginalnych elementów i całkowicie przewidywalne.
- Jeżeli wykonamy to samo wywołanie dwukrotnie, otrzymamy dokładnie takie same wyniki.
- Porządkowanie nie wyklucza wydajnego zrównoleglania.
- Na przykład przy obliczaniu `stream.map(fun)` strumień może być podzielony na `n` segmentów, z których każdy jest równocześnie przetwarzany.
- Następnie wyniki są ponownie ustawiane w kolejności.
- Niektóre operacje mogą być bardziej efektywnie zrównoleglane, jeśli odrzuci się konieczność ustawiania ich w kolejności.
- Wywołując metodę `Stream.unordered`, wskazujemy, że kolejność nie jest dla nas ważna.

Strumienie

Strumienie równoległe

- Operacją, która może mieć z tego korzyść, jest `Stream.distinct`. Na strumieniu uporządkowanym `distinct` pozostawia pierwszy z równych elementów.
- To utrudnia zrównoleglanie – wątek przetwarzający segment nie może wiedzieć, które elementy odrzucić przed przetworzeniem poprzedzających elementów.
- Jeżeli akceptowalne jest zachowanie dowolnego z równorzędnych elementów, wszystkie segmenty mogą być przetwarzane równolegle (korzystając ze wspólnego zbioru do śledzenia duplikatów).
- Można też przyspieszyć metodę `limit`, rezygnując z porządkowania.

Strumienie równoległe

- Jeżeli potrzebujemy dowolnych `n` elementów ze strumienia i nie ma znaczenia, jakie to będą elementy, należy wywołać:

```
Stream<String> sample =
    words.parallelStream().unordered().limit(n);
```

- Jak powiedzieliśmy uprzednio („Tworzenie map”), łączenie map jest kosztowne.
- Z tego powodu metoda `Collectors.groupingByConcurrent` korzysta z mapy współdzielonej przez równoległe wątki.
- Aby zrównoleglenie dało korzyść, kolejność wartości mapy nie powinna być taka sama jak kolejność w strumieniu:

```
// Wartości nie są zebrane w kolejności ze strumienia
Map<Integer, List<String>> result =
    words.parallelStream().collect(
        Collectors.groupingByConcurrent(String::length));
```

Strumienie równoległe

- Oczywiście nie sprawia różnicy to, że wykorzystujemy kolektor, który działa niezależnie od kolejności:

```
Map<Integer, Long> wordCounts = words.parallelStream()
        .collect(groupingByConcurrent(
             String::length, counting()));
```

- Bardzo ważne jest, aby nie modyfikować kolekcji, z której tworzony jest strumień podczas wykonywania operacji na strumieniu (nawet jeśli modyfikacja jest przystosowana do programów wielowątkowych).
- Pamiętajmy, że strumienie nie przechowują swoich danych – te dane zawsze znajdują się w oddzielnej kolekcji.
- Jeżeli zmodyfikujemy taką kolekcję, wynik działania operacji na strumieniu będzie nieprzewidywalny.

Strumienie równoległe

- Dokumentacja JDK nazywa to wymaganiem nieingerencji (ang. noninterference). Dotyczy to zarówno zwykłych, jak i równoległych strumieni.
- Mówiąc dokładniej: ponieważ pośrednie operacje na strumieniach są leniwe, możliwe jest modyfikowanie kolekcji do chwili wykonania końcowych operacji.
- Na przykład poniższe operacje, choć oczywiście niezalecane, zostaną poprawnie wykonane:

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

Strumienie równoległe

- Kolejny kod jest jednak nieprawidłowy:

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
// Błąd ingerencji
words.forEach(
    s -> if (s.length() < 12) wordList.remove(s));
```

Example (Parallel Streams)

- Program `r08/r08_14/ParallelStreams.java`