Zaawansowane programowanie obiektowe i funkcyjne

Wykład 6: Strumienie danych

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.3
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

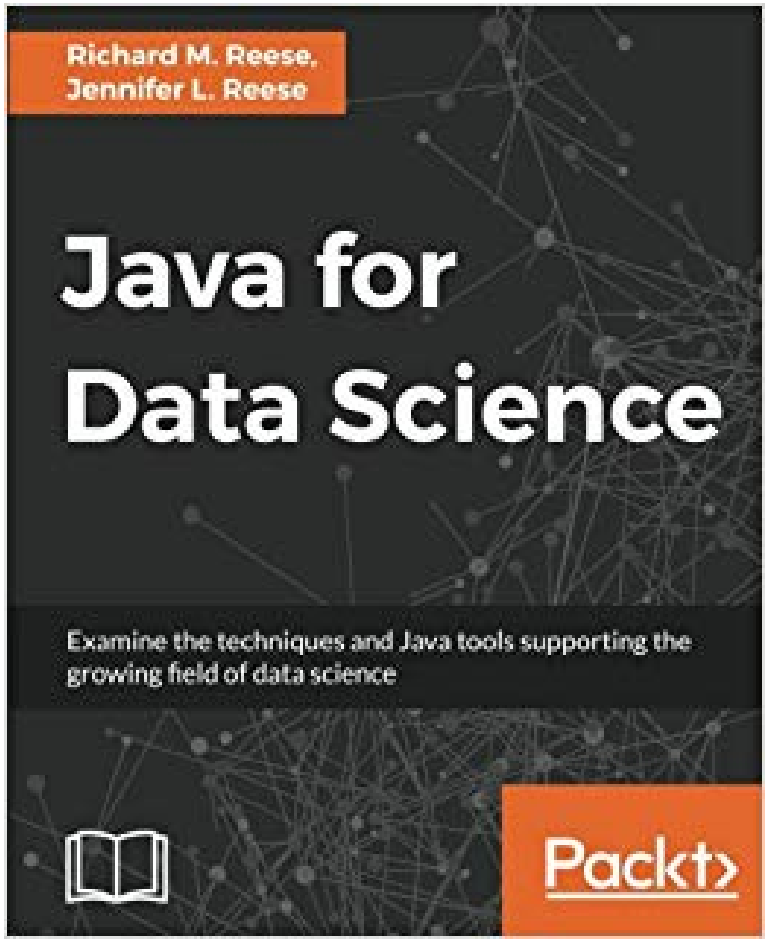
Rzeczpospolita
Polska

Politechnika
Warszawska

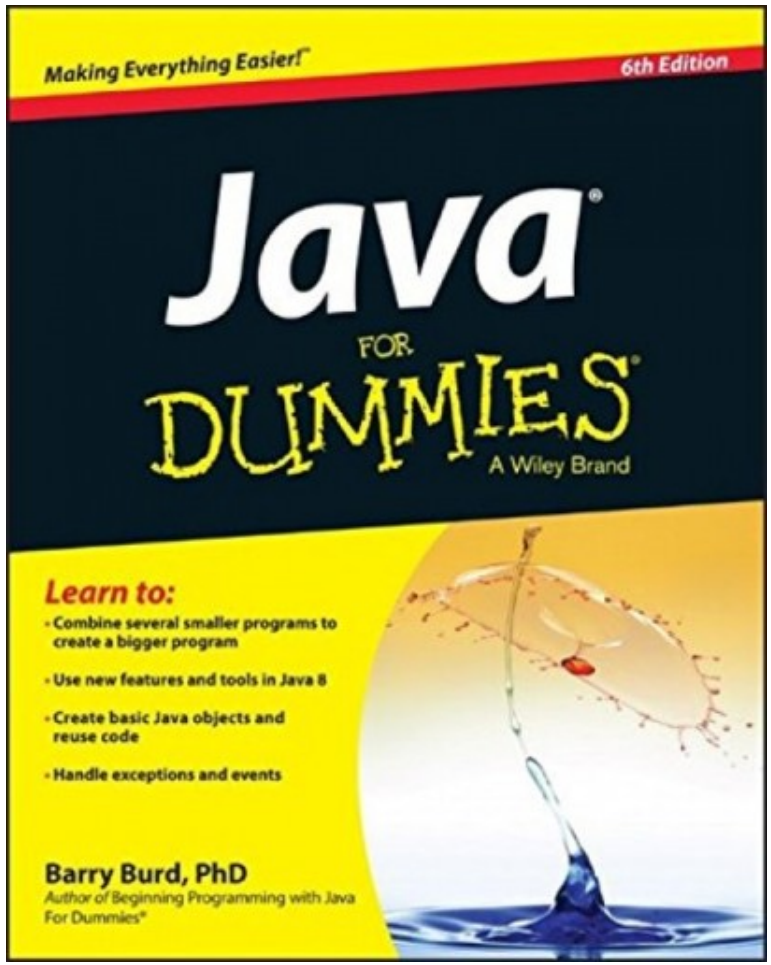
Strumienie danych `Stream`

- Strumienie pozwalają nam skupić się na przetwarzaniu danych bez zastanawiania się w jaki sposób to zrobić.
- Wykorzystują mechanizmy wyrażeń lambda i funkcje wyższego rzędu, ale w sposób czasami niewidoczny dla użytkownika.
- Bardzo łatwo da się je zastosować do obliczeń równoległych.

Zastosowanie w Analizie danych



Rysunek 1: *Java for Data Science [Reese and Reese, 2017]*



Rysunek 2: *Java for Dummies [Burd, 2003]*

Strumień kontra iterator

- Chcemy sprawdzić, czy książka o Javie [Burd, 2003] jest trudną lekturą.

```
1  String content = new String(Files.readAllBytes(
2  Paths.get("Beginning Programming with Java For Dummies.txt")),
       StandardCharsets.UTF_8);
3  List<String> words = Arrays.asList(content.split("[ \t\n.]"));
```

- Sprawdzimy czy zawiera dużo trudnych wyrazów.
- Załóżmy, że słowo jest trudne jeżeli liczy więcej niż 10 liter.

Iterator

```
1  int count = 0;
2  for (String w : words) {
3      if (w.length() > 10) count++;
4  }
```

Strumień

```
1  int count = words.stream()
2  .filter(s -> s.length() > 10)
3  .count();
```

- Okazuje się, że książka zawiera 4077 długich słów (około 2%). Nie jest zatem zbyt trudna.

Obliczenia równoległe

- Można przyśpieszyć obliczenia dzieląc książkę np. na rozdziały i zliczając równolegle liczbę długich słów w każdym z nich.
- Osiągniemy to dość łatwo poprzez zmianę metody `stream` na `parallelStream`, co spowoduje uruchomienie obliczeń równoległych.

```
1  long count = words.parallelStream()
2  .filter(s -> s.length() > 10)
3  .count();
```

Różnice w stosunku do kolekcji

- Strumień pracuje na zbiorze danych.
- Może w tym przypominać kolekcję, istnieją jednak pewne istotne różnice między nimi.
  - Strumień nie gromadzi danych.
  - Operacje nie zmieniają danych wejściowych.
  - Operacje na strumieniach są leniwe.
    - Dane są wczytywane tylko jeżeli jest to potrzebne.

Jak pracować ze strumieniami

1. Utwórz strumień.
2. Określ operacje pośrednie.
3. Określ operację terminalną.

Tworzenie strumienia

- Strumień możemy utworzyć:
  - Z kolekcji.
  - Z plików.
  - Tworząc generator strumienia.

Strumień utworzony z kolekcji

- Tworzenie strumieni statyczną metodą `Stream.of`.

```
1  Stream<String> words = Stream.of(content.split(" "));
```

```
1  Stream<String> sentence = Stream.of("We", "are", "not",
       "in", "Kansans", "anymore");
```

- Tworzenie strumieni z obiektu `Array`.

```
1  Arrays.stream(table, from, to),
```

- Tworzenie pustego strumienia.

```
1  Stream<String> silence = Stream.empty();
```

Tworzenie strumienia z tekstu lub pliku

- Strumień może odczytywać poszczególne linie z pliku `File`

```
1  try (Stream<String> lines = Files.lines(path)) {
2  //actions on lines
3  }
```

- Następnie można podzielić otrzymane teksty definiując separatory wzorcem `Pattern` używając metody `splitAsStream`.

```
1  Stream<String> words
2  = Pattern.compile(" ").splitAsStream(line);
```

Strumienie nieskończone

- Możemy tworzyć, najczęściej w celach testowych, nieskończone strumienie danych.
- Można powtarzać w kółko ten sam element.

```
1  Stream<String> repetitions = Stream.generate(() -> "Echo");
```

- Można generować różne wartości.

```
1  Stream<Double> random = Stream.generate(Math::random);
```

- Można generować kolejne wartości ciągu.

```
1  Stream<BigInteger> factorial
2                  = Stream.iterate(BigInteger.ONE, n ->
                       n.multiply(n.add(BigInteger.ONE)));
```

- Pierwszy element jest zaczynem ciągu kolejne powstają z użycia ostatniego elementu.

Użycie strumienia nieskończonego

- Stwórzmy ciąg wyliczający $n_{i+1} = n_i * (n_i + 1) \wedge n_1 = 1$.

```
1  Stream<BigInteger> factorial = Stream.iterate(BigInteger.ONE, n ->
       n.multiply(n.add(BigInteger.ONE)));
```

- Strumienie są leniwe, więc nie nic się nie wydarzy, dopóki nie rozpoczniemy konsumpcji elementów ciągu.

```
1  factorial.forEach(System.out::println);
```

- Przed zawieszeniem systemu uzyskamy kilka wartości ciągu.

Kolejne wartości strumienia

1. 1
2. 2
3. 6
4. 42
5. 1806
6. 3263442
7. 10650056950806
8. 113423713055421844361000442
9. 12864938683278671740537145998360961546653259485195806

Modyfikowanie strumienia

- Strumień możemy modyfikować przy pomocy kolejnych operacji pośrednich:
  - filtrowania,
  - przekształcania,
  - redukcji,
  - itp.

Filtrowanie i przekształcenie

- Możemy filtrować zawartość strumienia używając polecenia `filter`.

```
1  Stream<String> longWords = words.stream().filter(s ->
       s.length() > 10);
```

- Możemy też przekształcać dane w locie używając polecenia `map`.

```
1  Stream<String> smallWords =
       words.stream().map(String::toLowerCase);
2
3  Stream<String> firstLetters = words.stream().map(s ->
       s.substring(0, 1));
```

Przekształcenie spłaszczające

- Rozważmy metodę, która przekształca tekst w strumień liter.

```
1  public static Stream<String> letters(String s) {
2    List<String> result = new ArrayList<>();
3
4    for (int i = 0; i < s.length(); i++){
5      result.add(s.substring(i, i + 1));
6    }
7    return result.stream();
8  }
```

- Możemy użyć metody `map`, aby przekształcić strumień tekstów używając metody `letters`.

```
1  Stream<Stream<String>> result
2    = words.stream().map(w -> letters(w));
```

- W wyniku przekształcenia otrzymujemy strumień strumieni.
- Chcąc uzyskać strumień homogeniczny musimy użyć metody `flatMap`.

```
1  Stream<String> flatResult
2    = words.stream().flatMap(w -> letters(w))
```

Wybieranie elementów ze strumienia

- Pracując z generowanymi, nieskończonymi, strumieniami, możemy chcieć ograniczyć strumień do skończonej liczby elementów.
- Możemy zredukować strumień do kilku pierwszych wartości używając polecenia `limit`.

```
1  Stream<Double> random
2    = Stream.generate(Math::random).limit(100);
```

- Można także przeskoczyć *n* pierwszych elementów strumienia używając polecenia `skip`.

```
1  Stream<String> words
2    = Stream.of(content.split(" ")).skip(10);
```

Ograniczanie strumienia nieskończonego

- Możemy uzyskać podciąg nieskończonego strumienia określając początek i długość podciągu.

```
1  Stream<BigInteger> factorial = Stream.iterate(BigInteger.ONE, n ->
       n.multiply(n.add(BigInteger.ONE))).skip(5).limit(5);
```

Kolejne wartości strumienia

5. 3263442
6. 10650056950806
7. 113423713055421844361000442
8. 12864938683278671740537145998360961546653259485195806
9. 16550664732451996419846819544443918001751315270637749784185138876653586863957240680891198813173764518544 2

- Co się stanie jeżeli zamienimy kolejność wywołań?

```
1  Stream<BigInteger> factorial
2                  = Stream.iterate(BigInteger.ONE, n ->
                       n.multiply(n.add(BigInteger.ONE))).limit(5).skip(5);
```

- Powstanie pusty strumień.

Łączenie strumieni

- Mając kilka strumieni możemy je złączyć w jeden używając metody concat.

```
1  Stream<String> concatenated = Stream.concat(
2  letters("Pine"), letters("apple"));
3  //The result is P,i,n,e,a,p,p,l,e
```

- Jest to przydatne zwłaszcza przy obliczeniach równoległych i rozproszonych gdy analizujemy dane z różnych źródeł.
- Możemy także łączyć strumienie nieskończone, ale jeżeli pierwszy ze strumieni jest nieskończony to elementy drugiego strumienia nigdy nie wystąpią w stworzonym połączeniu.

Porządkowanie strumienia

- Możemy sortować zawartość strumienia używając metody `sorted` i interfejsu `Comparable`.

```
1  Stream<String> longestFirst =
2    words.stream().sorted(Comparator.comparing(String::length).reversed());
```

- Możemy także ograniczyć wyniki do unikalnych wartości używając metody `distinct`

```
1  Stream<String> uniqueWords
2    = Stream.of("Twinkle", "Twinkle", "Little", "Star").distinct();
3  // The result is "Twinkle", "Little", "Star"
```

Analiza zawartości książki

- Wróćmy do analizy książki. Chcielibyśmy się dowiedzieć jakie są najdłuższe słowa w niej użyte.
- Możemy posłużyć się metodą `sorted`, która uporządkuje dane w strumieniu, aby odczytać pięć najdłuższych słów.

```
1  words.sorted(Comparator.comparing(String::length).reversed())
2    .limit(5).forEach(System.out::println);
```

Pięć najdłuższych wyrazów

1. 1800000000000000000000000000000000000000000000000000
2. 0000000000000000000000000000000000000000000000000000
3. 0000000000000000000000000000000000000000000000000000
4. 0000000000000000000000000000000000000000000000000000
5. 0000000000000000000000000000000000000000000000000000

Analiza zawartości książki II

- Wyniki są mało przydatne, gdyż wielokrotnie pojawia nam się na liście ten sam wpis będący częścią liczby.
- Spróbujmy zmodyfikować nasz strumień tak, aby uzyskać tylko unikalne wyniki.

```
1  words.distinct().sorted(Comparator.comparing(String::length).reversed())
2    .limit(5).forEach(System.out::println);
```

Pięć unikalnych najdłuższych wyrazów

1. 1800000000000000000000000000000000000000000000000000
2. 0000000000000000000000000000000000000000000000000000
3. 00000000000000000000000000000000000000
4. goToTheSupermarketAndBuySome(bananas);
5. beginningprogrammingwithjavafdupdates

- Najdłuższym sensownym tekstem w książce jest `goToTheSupermarketAndBuySome(bananas)` co potwierdza nasze wcześniejsze ustalenia co do trudności tego tekstu;

Debugowanie strumieni

- Stosowanie strumieni zamiast iteratorów jest w praktyce szybsze i bardziej przejrzyste.
- Jednakże uruchamiając funkcje jedna po drugiej utrudniamy sobie możliwość debugowania kodu.
- Z pomocą przychodzi nam tutaj metoda `peek` która pozwala podejrzeć elementy strumienia bez jego modyfikacji.

```
1  Object[] powersof2 = Stream.iterate(1.0, p -> p * 2)
2    .peek(e -> System.out.println("Reading of " + e))
3    .limit(20).toArray();
```

- Metoda jest wywoływana dla każdego nowego elementu strumienia.
- Może wskazywać na metodę, w której umieścimy przerwanie dla debuggera.

Operacje terminalne

- Kończąc pracę ze strumieniem wykonujemy operację terminalną.
- Pozwalają one wykonać działania na wszystkich elementach strumienia.
- Umożliwiają także przekształcenie strumienia do innej postaci.

Redukcje

Definicja

Redukcje są operacjami terminalnymi które redukują strumień do postaci nie będącej strumieniem.

- Przykładem prostej redukcji jest metoda `count`, która zlicza elementy w strumieniu.

```
1  long count = words.stream()
2  .filter(s -> s.length() > 10)
3  .count();
```

Redukcje max i min

- Redukcje `max` i `min` mogą wydawać się analogiczne do redukcji `count`, ale ujawniają nam zawiłości analizy danych.
- Metody te wyliczane dla strumienia liczbowego nie zwracają odpowiednio maksymalnej i minimalnej wartości.
- Zwracają one obiekt typu `Optional<T>`[1] opakowujący wynik.
- Typ ten pozwala zabezpieczyć się przed brakami danych.
- Jeżeli minimalna czy maksymalna wartość istnieje i nie przyjmuje wartości `null` to metoda `isPresent()` obiektu `Optional<T>` zwróci `true`.
- Bezpiecznym sposobem odczytu wartości jest metoda `orElse(T)`, która uzupełnia braki argumentem wywołania.

```
1  Optional<String> longest =
2    longWords.filter(s->s.startsWith("x"))
3    .max((s,t)->s.length()-t.length());
4  System.out.println("longest: " + longest.orElse(""));
```

[1] Poświęcimy mu jeszcze trochę uwagi później

Wyszukiwanie elementów

- Często przetwarzanie strumienia ma na celu znalezienie elementów spełniających pewne warunki.
- Możemy sprawdzić czy istnieje w naszych długich słowach takie, które zaczyna się na "x".

```
1  longWords.anyMatch(s->s.startsWith("x"))
2  //false
```

- Możemy znaleźć pierwsze wystąpienie słowa składającego się z jednastu liter.

```
1  Optional<String> word = longWords
2    .filter(s -> s.length()==11).findFirst();
3  //Programming
```

- Lub dowolne słowo kończące się na "ing".

```
1  Optional<String> word = longWords
2    .filter(s -> s.endsWith("ing")).findAny();
3  //Programming
```

Iterowanie wyników

- Możemy przejść po elementach uzyskanych w wyniku działania strumienia korzystając z metody `iterate`, która zwraca iterator.
- Alternatywnie można wywołać metodę `forEach`.

```
1 stream.forEach(System.out::println);
```

- Metoda przechodzi przez elementy w przypadkowej kolejności `forEach`.
- Jeżeli chcemy zachować istniejącą kolejność, musimy użyć metody `forEachOrdered`.
- Jednakże używając metody `forEachOrdered` nie możemy korzystać z obliczeń równoległych.

Przekształcenie do tablicy

- Metoda `toArray()` przekształca strumień w tablicę `Object[]`.
- Stosując referencję do konstruktora możemy utworzyć tablice określonego typu.

```
1  String[] result = stream.toArray(String[]::new);
```

Kolektor

- Metoda `collect` używa interfejsu `Collector` aby przekształcić strumień w kolekcję.
- Klasa `Collectors` dostarcza metody do konwersji strumienia w popularne kolekcje

```
1  List<String> list =
       stream.collect(Collectors.toList());
2  Set<String> set = stream.collect(Collectors.toSet());
```

Agregacja tekstów

- Jeżeli strumień składa się z tekstów można dokonać ich agregacji w jeden tekst.
- Metoda `joining` łączy wszystkie obiekty typu `String` w jeden obiekt.

```
1  String text = stream.collect(Collectors.joining());
```

- Metoda umożliwia dodawanie separatorów.

```
1  String text = stream.collect(Collectors.joining(", "));
```

Agregacja innych obiektów

- Odczytujemy dane z czujnika pyłów[2].
- Mierzymy zagęszczenie zawieszonych w powietrzu cząsteczek o średnicy nie większej niż $10\mu m$.[3]
- Możemy agregować otrzymane odczyty wymuszając ich konwersję do typu `String` poleceniem `map`

```
1  dataList.filter(d->d>100)
2    .map(Object::toString).collect(Collectors.joining(","));
```

- Może być to przydatne przy eksporcie wyników.

102.87,100.67,101.93,101.2,103.7,104.17,101.2,100.77,100.2,111.53,102.63,108.13,
102.43,108.67,104.23,102.97,110.93,105.27,112.77,100.37,116.7,108.7,103.23,112.37,104.6,
101.07,108.93,106.0,164.27

[2] https://luftdaten.info/en/construction-manual/

[3] https://pl.wikipedia.org/wiki/PM10

Statystyki

- Możemy wyliczyć statystyki dla danych z czujnika pyłów używając metody `summarizing(Int|Long|Double)`.
- Metoda zwraca obiekt klasy `(Int|Long|Double)Summary`.

```
1  DoubleSummaryStatistics summary =
2    dataList.collect(Collectors.summarizingDouble(d->d));
3
4  System.out.println("Measurements - min: "
5  +summary.getMin()+" avg: "
6  +summary.getAverage()+"max: "
7  +summary.getMax());
```

Wynik:

Measurements - min: 7.33 avg: 38.12 max: 164.27

Tworzenie mapy

- Metoda `toMap` klasy `Collectors` pozwala na tworzenie map.
- Mając strumień studentów możemy utworzyć mapowanie numeru indeksu na nazwisko.

```
1  Map<Integer, String> indexToSecondName =
2    students.collect(Collectors.toMap(Student::getIndex,
         Student::getSecondName));
```

- Chcąc stworzyć mapowanie na obiekt `Student` możemy użyć metody `identity()`.

```
1  Map<Integer, Student> indexToStudent =
2    students.collect(Collectors.toMap(Student::getIndex,
         Function.identity()));
```

Duplikaty

- Metoda `toMap` nie radzi sobie z duplikatami kluczy.
- Jeżeli będzie więcej niż jeden student o tym samym indeksie wyrzuci wyjątek `IllegalStateException`.
- Jeżeli chcemy pracować z duplikatami możemy użyć funkcji, która ustali właściwą wartość dla klucza.

```
1  Map<String, String> addressBook =
2    people.stream()
3    .collect(Collectors.toMap(
4                Person::getName,
5                Person::getAddress,
6                (oldAddress, newAddress) ->
                     {System.out.println("duplicate key
                     found!");
7                    return newAddress;
8                }
9            ));
```

Wielowartościowość

- Innym podejściem jest zezwolenie na mapowanie wielowartościowe.

```
1  Map<String, Set<String>> addressBook =
2    people.stream()
3    .collect(Collectors.toMap(
4    Person::getName,
5    person -> Collections.singleton(person.getAddress()),
6    (oldAddress, newAddress) -> { // Merge a i b
7    Set<String> set = new HashSet<>(oldAddress);
8      set.addAll(newAddress);
9      return set;
10   }));
```

Grupowanie

- Możemy grupować dane względem atrybutów.

```
1  Map<String, List<Student>> groupOfStudent =
       names.stream().collect(
2  Collectors.groupingBy(Student::getCourse));
```

- Metoda `Student::getCourse` jest funkcją klasyfikującą.
- Teraz możemy odczytać jacy studenci uczęszczają na dany kurs.

```
1  groupOfStudent.get("Java");
2  //[Student{name=Alicja, course=Java},
       Student{name=Marcin, course=Java}]
```

Typ EnumMap

- Typ wyliczeniowy może być łatwo wykorzystany do tworzenia mapy.
- Istnieje specjalny typ `EnumMap` rzutujący wyliczenie na obiekt.
- Można łatwo pogrupować obiekty względem wartości wyliczenia stanowiącego ich pole.

Dyżury

```
1  ArrayList<Dean> deans = new  ArrayList<Dean>();
2  deans.add(new Dean("Krzysztof Kaczmarski",TUESDAY));
3
4  ...
5
6  System.out.println(deans.stream().collect(
7    groupingBy(l ->l.getOfficeDay(),
8      ()->new EnumMap<>(WorkingDay.class), toSet())));
```

```
MONDAY=[Magdalena Jasionowska-Skop],
TUESDAY=[Krzysztof Kaczmarski,Bogusława Karpińska,
Anna Cena],
WEDNESDAY=[Iwona Wróbel,Przemysław Grzegorzewski,
Łukasz Błaszczyk]
```

Podział

- Specjalną metodę grupowania oferuje metoda `partitioningBy`.
- Zwraca ona dwu elementową mapę.
  - `true`
  - `false`
- Możemy jej użyć do określenia warunków selekcji danych.

```
1  Map<Boolean, List<Student>> groupOfStudent =
       names.stream().collect(
2    Collectors.partitioningBy(
3    s->s.getCourse().equals("Java")));
4
5  System.out.println(groupOfStudent.get(false));
6   //[Student{name=Malgorzata, course=ATL},
        Student{name=Aleksander, course=Android}]
```

Zliczanie

- Elementy w grupach możemy zliczyć używając metody `counting`

```
1  Map<String, Long> countOfStudents;
2  countOfStudents = names.stream().collect(
3    Collectors.groupingBy(Student::getCourse,
         counting()));
4
5  System.out.println(countOfStudents.entrySet());
6  //[Java=2, ATL=1, Android=1]
```

Operacje na grupach

- Podobnie jak w zliczanie możemy przeprowadzić inne operacje na grupach.
- Możemy sumować dane.

```
1  Map<String, Integer> sumOfPoints =
2    names.stream().collect(
3    Collectors.groupingBy(Student::getCourse,
4    Collectors.summingInt(Student::getPoints)));
5
6  System.out.println(sumOfPoints.entrySet());
7  //[Java=107, ATL=24, Android=25]
```

- Możemy znajdować ekstrema.

```
1  Map<String, Optional<Student>> maxOfPoints =
2    names.stream().collect(
3    Collectors.groupingBy(Student::getCourse,
4    Collectors.maxBy(Comparator.comparing(Student::getPoints))));
5
6  //[Java=Optional[Student{name=Marcin, points=66}],
     ATL=Optional[Student{name=Alicja, points=24}],
     Android=Optional[Student{name=Aleksander, points=25}]]
```

Redukcja

- Możemy także zredukować cały strumień do jednej wartości.

```
1  //[Java=173, ATL=22, Android=73]
2  OptionalInt sum =
3    names.stream().mapToInt(Student::getPoints)
4    .reduce((x, y) -> x + y);
5
6  System.out.println(sum.getAsInt());
7  //268
```

Strumienie równoległe

- Strumienie równoległe pozwalają nam znacznie przyśpieszyć przetwarzanie danych poprzez równoległą realizację podzadań.
- W Javie dzieje się to w sposób automatyczny i sprowadza do deklaracji, że chcemy przetwarzać dane równolegle.
- Jednakże zastosowanie strumieni równoległych jest w pewnym stopniu ograniczone w porównaniu do strumieni sekwencyjnych.

Tworzenie strumieni równoległych

- Metoda `parallelStream()` tworzy strumień równoległy z kolekcji

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

- Możemy także przekształcić dowolny strumień sekwencyjny w strumień równoległy.

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

Operacje równoległe

- Operacje równoległe muszą być bezstanowe.
  - Nie mogą opierać się na stanie przetwarzanych danych.
- Operacje nie mogą być zależne od kolejności danych.

Zasada braku interwencji

Nie modyfikuj kolekcji, która jest używana przez strumień (równoległy lub sekwencyjny)

Nieuporządkowane dane

- Zazwyczaj, w przypadku obliczeń równoległych nie możemy gwarantować zachowania porządku danych wejściowych.
- Jednakże rezygnując z niego możemy przyśpieszyć czas obliczeń.
- Metoda `unordered` w sposób jawny określa, że nie musimy zachowywać porządku danych.

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

- Powyższe polecenie zwróci *n* dowolnych elementów z kolekcji.

Równoległa analiza zawartości książki

- Chcemy uzyskać 15 unikalnych długich słów z książki, równolegle przetwarzając strumień.

```
1  words.filter(s -> s.length() > 10).unordered()
2    .distinct().limit(15).forEach(System.out::println);
```

- Porównajmy wyniki z przetwarzaniem sekwencyjnym.

sekwencyjne

1. Programming
2. beginningprogrammingwithjava
3. Introduction
4. Conventions
5. Assumptions
6. Controlling
7. Translating
8. Instructions
9. Development
10. Environment
11. . . .

równoległe

1. declarations
2. Controlling
3. Introduction
4. Assumptions
5. Punctuation
6. Conventions
7. Identifiers
8. Translating
9. Instructions
10. Programming
11. . . .

Mapy równoległe

- Każda metoda typu `toMap` posiada odpowiednik typu `toConcurrentMap` służący do przetwarzania równoległego.
- Metoda `groupingByConcurrent` tworzy mapę uzupełnianą równolegle bez zachowania orginalnej kolejności danych.

```
1  Map<Integer, List<String>> result =
       words.parallelStream().collect(
2  Collectors.groupingByConcurrent(String::length));
```

Klasa `Optional`

- Obiekt `Optional<T>` opakowuje obiekt `T` lub `null`.
- Właściwie użyty jest bezpieczną alternatywą dla bezpośredniego używania referencji do `T`, która może wskazywać `null`.
- W takim przypadku `Optional<T>` zwraca wartość alternatywną

Wartość alternatywna

- Wartość alternatywna może być deklarowana na kilka sposobów, korzystając z różnych metod.
  - Wartość statyczna

```
1  String result = optionalString.orElse("");
```

  - Wartość wyliczana

```
1  String result = optionalString.orElseGet(() ->
       System.getProperty("user.dir"));
```

  - Generowanie wyjątku

```
1  String result =
       optionalString.orElseThrow(IllegalStateException::new);
```

Praca z Optional

- Jeżeli chcemy pracować z wartościami spakowanymi w typie Optional to najlepiej wykorzystać konsumenta bezpiecznie zagnieżdżonego w metodzie ifPresent.

```
1  optionalValue.ifPresent(v -> workWith(v));
```

- Chcąc uzyskać kolekcję niepustych wartości postępujemy podobnie.

```
1  optionalValue.ifPresent(v -> results.add(v));
2  optionalValue.ifPresent(results::add);
```

- Niestety nie otrzymamy w ten sposób informacji czy w danym wypadku dodaliśmy wartość do kolekcji, gdyż metoda ifPresent nie zwraca żadnej wartości (void).
- Aby się tego dowiedzieć musimy użyć następującej konstrukcji:

```
1  Optional<Boolean> added =
       optionalValue.map(results::add);
```

- Referencja added może przyjąć jedną z wartości true, false lub null

Tworzenie instancji `Optional`

- Do stworzenia instancji można użyć następujących metod statycznych klasy `Optional`:
  - `empty()`
    - pusty obiekt `Optional`.
  - `of(value)`
    - obiekt opakowujący wartość `value`, która nie jest `null`.
  - `ofNullable(obj)`
    - obiekt opakowujący wartość `obj`, która może być `null`.

Przykład:

```
1  public static Optional<Double> inverse(Double x) {
2    return x == 0 ?
3      Optional.empty() : Optional.of(1/x);
4  }
```

Superpozycja

- Dane:
  - Metoda `Optional<T> f()` obiektu `s`
  - Klasa `T` z metodą `Optional<U> g()`
- Nie możemy wywołać `s.f().g()` gdyż typ `T` jest opakowany.

```
1  Optional<U> result = s.f().flatMap(T::g);
```

Przykłady superpozycji

- Inwersja

```
1  public static Optional<Double> inverse(Double x) {
2    return x == 0 ? Optional.empty() : Optional.of(1 /
         x);
3  }
```

- Pierwiastek kwadratowy

```
1  public static Optional<Double> squareRoot(Double x) {
2    return x < 0 ? Optional.empty() :
         Optional.of(Math.sqrt(x));
3  }
```

- Superpozycja

```
1  Optional<Double> result =
       inverse(x).flatMap(SafeMath::squareRoot);
```

Bibliografia

[Burd, 2003] Burd, B. A. (2003).
Beginning Programming With Java for Dummies.
JOHN WILEY & SONS INC.

[Reese and Reese, 2017] Reese, R. M. and Reese, J. L. (2017).
Java for Data Science.
Packt.