Zaawansowane programowanie w Javie

Studia zaoczne – Wykład 5

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/

Inne transformacje

- Dotychczas omawiane transformacje strumieniowe były bezstanowe (ang. *stateless*). Oznacza to, że gdy element jest pobierany z filtrowanego lub odwzorowanego strumienia, wynik nie zależy od poprzednich elementów.
- Istnieje również kilka transformacji stanowych (ang. *stateful*).
- Przykładowo, metoda `distinct` zwraca strumień, który dostarcza elementy z oryginalnego strumienia, w tej samej kolejności, z wyjątkiem tego, że duplikaty są pomijane.
- Strumień musi oczywiście pamiętać elementy, które już widział.

```
// Tylko jedno słowo "merrily" jest zachowane
Stream<String> uniqueWords = Stream.of(
    "merrily", "merrily", "merrily", "gently").distinct();
```

Inne transformacje

- Metoda `sorted` musi widzieć cały strumień i posortować go wcześniej zanim może zacząć dostarczać jakikolwiek elementy. Może się przecież zdarzyć, że najmniejszy element jest ostatnim.
- Oczywiście nie można sortować nieskończonego strumienia.
- Istnieje kilka wersji metody `sorted` służącej do sortowania strumienia.
- Jedna działa ze strumieniami zawierającymi elementy implementujące interfejs `Comparable`, a inna przyjmuje `Comparator`.
- Poniżej sortujemy ciągi znaków od najdłuższego do najkrótszego:

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

Strumienie

Inne transformacje

- Tak jak w przypadku wszystkich przekształceń strumieni, metoda `sorted` zwraca nowy strumień, którego elementami są elementy oryginalnego strumienia w odpowiedniej kolejności.
- Oczywiście można posortować kolekcję bez korzystania ze strumieni. Metoda `sorted` jest przydatna, gdy proces sortowania jest częścią potoku strumienia.

Wycinanie podstrumieni i łączenie strumieni

- Metoda `peek` zwraca inny strumień zawierający takie same elementy jak strumień oryginalny, ale przy pobieraniu każdego elementu jest wywoływana funkcja. Jest to przydatne przy debugowaniu:

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

- Komunikat jest wyświetlany w chwili, gdy element jest rzeczywiście pobierany.
- W ten sposób można sprawdzić, że nieskończony strumień zwracany przez `iterate` rzeczywiście jest przetwarzany w sposób leniwy.

Wycinanie podstrumieni i łączenie strumieni

- Przy debugowaniu można wykorzystać peek do wywołania metody, w której został ustawiony punkt przerwania (ang. breakpoint).
- W większości IDE można również ustawiać punkty przerwania w wyrażeniach.
- Jeżeli chcemy wiedzieć, co się dzieje w konkretnym punkcie w potoku strumienia, możemy dodać

```
.peek(x -> {
    return; })
```

i ustawić punkt przerwania w drugim wierszu.

Strumienie

Przykład (Inne transformacje)

- Program `r08/r08_05/OtherTransformations.java`
  - Kompilacja:
    `javac r08/r08_05/OtherTransformations.java`
  - Wykonanie:
    `java r08.r08_05.OtherTransformations`

Zadanie

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

Strumienie

Proste redukcje

- Gdy już wiemy, w jaki sposób tworzyć i przekształcać strumienie, doszliśmy do najważniejszego punktu – pobrania odpowiedzi dotyczącej danych ze strumienia.
- Metody, którymi się teraz zajmiemy, są nazywane redukcjami (ang. reductions).
- Redukcje są operacjami kończącymi.
- Redukują one strumień do wartości niebędącej strumieniem, którą można wykorzystać w swoim programie.
- Widzieliśmy już prostą redukcję: metodę `count` zwracającą liczbę elementów strumienia.

Proste redukcje

- Inne proste redukcje to `max` i `min`, zwracające największą i najmniejszą wartość.
- Zauważmy, że metody te zwracają wartość `Optional<T>`, która opakowuje wartość zwracaną lub wskazuje, że nie ma żadnej wartości (ponieważ strumień był pusty).
- Dawniej często w takiej sytuacji zwracano wartość **`null`**.
- Może to jednak doprowadzić do wyjątków „null pointer exception” związanych ze wskaźnikiem **`null`**, wtedy gdy pojawi się on w niedokładnie przetestowanym programie.

Strumienie

Proste redukcje

- Typ `Optional` jest lepszym sposobem informowania o braku zwracanej wartości.
- Maksymalną wartość ze strumienia można pobrać w taki sposób:

```
Stream<String> words = Stream.of(
    contents.split("\\PL+")).skip(1);
Optional<String> largest =
    words.max(String::compareToIgnoreCase);
if (largest.isPresent()) {
    System.out.println(
        "largest: " + largest.get()
    );
} else {
    System.out.println("There is no largest word");
}
```

Proste redukcje

- Metoda `findFirst` zwraca pierwszą wartość z niepustej kolekcji. Często przydaje się to w połączeniu z `filter`.
- Przykładowo, można znaleźć pierwszy wyraz zaczynający się od litery `Q`, jeżeli taki istnieje:

```
Optional<String> startsWithQ =
    words.filter(s -> s.startsWith("Q")).findFirst();
```

- Jeżeli wystarczy, że wybrany zostanie dowolny, niekoniecznie pierwszy, pasujący wyraz, można użyć metody `findAny`.
- Zwiększa to efektywność przy równoległym przetwarzaniu strumienia, ponieważ strumień może zwrócić dowolny element i nie jest ograniczony do pierwszego.

```
Optional<String> startsWithQ = words.parallel().
    filter(s -> s.startsWith("Q")).findAny();
```

Proste redukcje

- Jeżeli chcemy po prostu wiedzieć, czy istnieje pasujący element, wystarczy użyć metody `anyMatch`.
- Ta metoda pobiera argument z predykatem, dzięki czemu nie musimy korzystać z `filter` .

```
boolean aWordStartsWithQ =
    words.parallel().anyMatch(s -> s.startsWith("Q"));
```

- Istnieją także metody `allMatch` i `noneMatch` zwracające `true`, jeżeli wszystkie elementy spełniają warunki z predykatu lub żaden element nie spełnia warunków.
- Te metody również mogą być zrównoleglone.

Strumienie

Przykład (Proste redukcje)

- Program `r08/r08_06/SimpleReductions.java`
  - Kompilacja:
    `javac r08/r08_06/SimpleReductions.java`
  - Wykonanie:
    `java r08.r08_06.SimpleReductions`

Zadanie

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

Strumienie

Typ Optional

- Obiekt `Optional<T>` opakowuje obiekt typu `T` lub brak obiektu.
- W pierwszym przypadku mówimy, że wartość jest obecna.
- Typ `Optional<T>` jest bezpieczniejszą alternatywą dla referencji do typu `T`, która może wskazywać obiekt lub przyjmować wartość `null`.
- Jednak jest on bezpieczniejszy tylko pod warunkiem, że poprawnie z niego korzystamy.

Jak korzystać z typu `Optional`

- Kluczem do efektywnego wykorzystywania typu `Optional<T>` jest korzystanie z metody, która tworzy alternatywną wartość, jeżeli zwracana wartość nie istnieje, lub pobiera wartość, jeżeli jest ona obecna.
- Popatrzmy na pierwszy wariant: często istnieje wartość domyślna, którą chcemy wykorzystać, jeżeli nie pojawi się żadna wartość; może to być na przykład pusty ciąg znaków:

```
// Opakowany ciąg znaków lub "", jeżeli brak
String result = optionalString.orElse("");
```

Jak korzystać z typu `Optional`

- Można wywołać kod obliczający wartość domyślną:

```
// Funkcja wywoływana tylko w razie potrzeby
String result = optionalString.orElseGet(
    () -> System.getProperty("user.dir"));
```

- Można też wyrzucić wyjątek, jeżeli nie ma wartości:

```
// Przekaż metodę, która zwraca obiekt wyjątku
String result = optionalString.orElseThrow(
    NoSuchElementException::new);
```

Jak korzystać z typu `Optional`

- Zobaczyliśmy, w jaki sposób utworzyć alternatywną wartość, jeżeli nie ma właściwej wartości.
- Inną strategią przy pracy z wartościami opcjonalnymi jest wykorzystywanie wartości tylko wtedy, gdy jest ona obecna.
- Metoda `ifPresent` przyjmuje funkcję. Jeżeli wartość opcjonalna istnieje, jest ona przekazywana do tej funkcji. W innym przypadku nic się nie dzieje.

```
optionalValue.ifPresent(v -> Process v);
```

Jak korzystać z typu `Optional`

- Jeżeli na przykład chcemy dodać wartość do zbioru – o ile taka wartość się pojawi – wywołajmy

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

lub po prostu

```
optionalValue.ifPresent(results::add);
```

- Gdy wywołujemy tę wersję metody `ifPresent`, funkcja nie zwraca wartości.
- Jeżeli chcemy przetwarzać wynik działania funkcji, użyjmy zamiast tego `map`:

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

Jak korzystać z typu `Optional`

- Teraz `added` ma jedną z trzech wartości: `true` lub `false` opakowane obiektem `Optional`, jeżeli istniała wartość `optionalValue`, albo pusty typ `Optional` w przeciwnym wypadku.
- Ta metoda `map` jest analogiczna do metody `map` interfejsu `Stream`, który już widzieliśmy omawiając metody `filter`, `map` oraz `flatMap`.
- Po prostu wyobraźmy sobie opcjonalną wartość jako strumień o rozmiarze zero lub jeden.
- Wynik ma tutaj rozmiar zero lub jeden i w tym drugim przypadku wywoływana jest funkcja.

Jak korzystać z typu `Optional`

```
jshell> Set<Integer> results = new TreeSet<>()
results ==> []

jshell> Optional<Integer> optVal = Optional.of(3)
optVal ==> Optional[3]

jshell> Optional<Boolean>added = optVal.map(results::add)
added ==> Optional[true]

jshell> Optional<Boolean>added = optVal.map(results::add)
added ==> Optional[false]

jshell> Optional<Integer> empty = Optional.empty()
empty ==> Optional.empty

jshell> Optional<Boolean>added = empty.map(results::add)
added ==> Optional.empty
```

Jak nie korzystać z typu `Optional`

- Jeśli nie używamy wartości `Optional` poprawnie, nie wykorzystamy jej przewagi nad starym podejściem „coś lub null”.
- Przykładowo, metoda opakowująca `get` pobiera opakowany element, jeżeli taki element istnieje, a w przeciwnym przypadku wyrzuca wyjątek `NoSuchElementException`.
- Dlatego

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod();
```

nie jest bezpieczniejsze niż

```
T value = ...;
value.someMethod();
```

Jak nie korzystać z typu `Optional`

- Metoda `isPresent` daje informację, czy obiekt `Optional<T>` zawiera jakąś wartość. Jednakże

```
if (optionalValue.isPresent())
    optionalValue.get().someMethod();
```

nie jest prostsze niż

```
if (value != null)
    value.someMethod();
```

- Jak dotąd omówiliśmy sposoby wykorzystywania obiektów `Optional` utworzonych przez kogoś innego. Teraz poznamy sposoby tworzenia obiektów typu `Optional`.

Tworzenie wartości typu `Optional`

- Jeżeli chcemy napisać metodę tworzącą obiekt `Optional`, istnieje kilka służących do tego statycznych metod, w tym `Optional.of(result)` oraz `Optional.empty()`.
- Na przykład

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

- Metoda `ofNullable` jest wymyślona jako pomost pomiędzy możliwym użyciem wartości **`null`** a wartościami opcjonalnymi.
- `Optional.ofNullable(obj)` zwraca `Optional.of(obj)` jeżeli `obj` nie jest **`null`**, a `Optional.empty()` w innym przypadku.

Łączenie flatMap z funkcjami wartości typu `Optional`

- Przypuśćmy, że mamy metodę `f` zwracającą `Optional<T>` i docelowy typ `T` mający metodę `g` zwracającą `Optional<U>`.
- Jeżeli byłyby one zwykłymi metodami, moglibyśmy je połączyć, wywołując `s.f().g()`.
- Takie połączenie jednak tutaj nie zadziała, ponieważ `s.f()` ma typ `Optional<T>`, a nie`T`.
- Zamiast tego wywołajmy

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

- Jeżeli `s.f()` istnieje, wykonywana jest `g`. W innym przypadku zwracany jest pusty `Optional<U>`

Łączenie flatMap z funkcjami wartości typu `Optional`

- Jak widać, można to powtarzać, jeżeli mamy więcej metod lub wyrażeń lambda zwracających wartości `Optional`.
- Możemy wtedy utworzyć ciąg kroków, po prostu łącząc wywołania `flatMap` które dadzą wynik, pod warunkiem że wszystkie kroki się powiodą.
- Jako przykład przeanalizujmy zaprezentowaną wcześniej bezpieczną metodę `inverse` i rozważmy bezpieczną metodę do obliczania pierwiastka kwadratowego:

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

Łączenie flatMap z funkcjami wartości typu `Optional`

- Możemy w takiej sytuacji wyznaczyć pierwiastek kwadratowy z wartości zwracanej przez `inverse`:

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

lub – jeżeli wolimy:

```
Optional<Double> result = Optional.of(-4.0)
    .flatMap(Test::inverse)
    .flatMap(Test::squareRoot);
```

- Jeżeli metoda `inverse` lub `squareRoot` zwraca `Optional.empty()`, wynik jest pusty.

Strumienie

Typ Optional

- Metoda `flatMap` z klasy `Optional` jest analogiczna do metody `flatMap` z interfejsu `Stream`.
- Metoda ta była wykorzystana do połączenia dwóch metod zwracających strumienie poprzez spłaszczenie zwracanego strumienia strumieni.
- Metoda `Optional.flatMap` działa w taki sam sposób, jeżeli traktujemy wartości opcjonalne jako strumienie o rozmiarze zero lub jeden.

Przykład (Wartości opcjonalne)

- Program `r08/r08_07/OptionalDemo.java`
- Program `r08/r08_07rec/OptionalDemo.java`

Strumienie

Kolekcje wyników

- Jeżeli po zakończeniu pracy ze strumieniem zechcemy obejrzeć jej wyniki, możemy wywołać metodę `iterator` zwracającą iterator, za pomocą którego można przeglądać elementy.
- Alternatywnie, możemy wywołać metodę `forEach`, aby na każdym elemencie wykonać funkcję:

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

- W strumieniu równoległym metoda `forEach` przetwarza elementy w dowolnej kolejności. Jeżeli chcemy przetwarzać elementy strumienia po kolei, to możemy wywołać zamiast niej metodę `forEachOrdered`.
- Oczywiście możemy w ten sposób stracić niektóre lub wszystkie korzyści wynikające z przetwarzania równoległego.

Kolekcje wyników

- Częściej jednak będziemy chcieli zebrać wyniki w strukturze danych. Przykładowo, możemy wywołać metodę `toArray` i otrzymać tablicę zawierającą elementy strumienia.
- Ponieważ nie jest możliwe utworzenie uogólnionej tablicy w czasie działania kodu, wyrażenie `stream.toArray()` zwraca tablicę `Object[]`.
- Jeżeli potrzebujemy tablicy o właściwym typie, przekażmy do niego konstruktor odpowiedniej tablicy:

```
// Wynik wywołania words.toArray()
// jest typu Object[]
String[] result = words.toArray(String[]::new);
```

Strumienie

Kolekcje wyników

- Przypuśćmy, że chcemy zebrać wyniki w zbiorze typu `HashSet`.
- Jeżeli kolekcja jest zrównoleglona, nie można umieścić jej elementów bezpośrednio w pojedynczym obiekcie typu `HashSet`, ponieważ obiekt `HashSet` nie jest bezpieczny wątkowo.
- Dlatego nie można użyć metody `reduce`. Każdy segment musi zaczynać się od własnego pustego obiektu `HashSet`, a `reduce` pozwala tylko na podanie jednej wartości. Zamiast tego należy użyć metody `collect`, która wymaga trzech argumentów:
  1. Funkcji do tworzenia nowych instancji obiektu docelowego, przykładowo konstruktora dla obiektu `HashSet`,
  2. Funkcji, który dodaje element do wyniku, przykładowo metody `add`,
  3. Funkcji, który połączy dwa obiekty w jeden, przykładowo metody `addAll`.
- Zwróćmy uwagę, że obiekt docelowy nie musi być kolekcją. Może to być np. obiekt typu `StringBuilder`.

Kolekcje wyników

- Oto jak działa metoda `collect` dla obiektu typu `HashSet`:

```
HashSet<String> result = stream.collect(
    HashSet::new, HashSet::add, HashSet::addAll);
```

- W praktyce nie musimy tego robić, ponieważ istnieje wygodna metoda `collect` pobierająca instancję interfejsu `Collector`.
- Natomiast klasa `Collectors` dostarcza wielu metod fabrycznych dla popularnych struktur.
- Aby umieścić elementy strumienia w liście lub zbiorze, można wywołać:

```
List<String> result = stream.collect(Collectors.toList());
```

lub

```
Set<String> result = stream.collect(Collectors.toSet());
```

Kolekcje wyników

- Jeżeli chcemy kontrolować, jakiego rodzaju zbiór otrzymujemy, możemy użyć wywołania:

```
TreeSet<String> result = stream.collect(
    Collectors.toCollection(TreeSet::new));
```

lub

```
HashSet<String> result = stream.collect(
    Collectors.toCollection(HashSet::new));
```

Strumienie

Kolekcje wyników

- Aby utworzyć listę `ArrayList` zawierającą elementy strumienia można użyć wywołania:

```
List<String> asList = stringStream.collect(
    ArrayList::new, ArrayList::add, ArrayList::addAll);
```

albo prościej:

```
List<String> asList = stringStream.collect(
    Collectors.toCollection(ArrayList::new));
```

- Aby utworzyć obiekt klasy `StringBuilder` zawierający połączone łańcuchy znaków ze strumienia można użyć wywołania:

```
StringBuilder concat = stringStream.collect(
    StringBuilder::new, StringBuilder::append,
    StringBuilder::append);
```

Kolekcje wyników

- Aby zebrać wszystkie łańcuchy znaków będące elementami strumienia, stosujemy metodę `joining`:

```
String result = stream.collect(Collectors.joining());
```

- Jeżeli chcemy, aby elementy były rozdzielone znacznikiem, przekazujemy go do metody `joining`:

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

- Jeżeli strumień zawiera obiekty inne niż łańcuchy znaków, należy je najpierw przekonwertować na łańcuchy znaków:

```
String result = stream
    .map(Object::toString)
    .collect(Collectors.joining(", "));
```

Kolekcje wyników

- Jeżeli chcemy zredukować zawartość strumienia do sumy, średniej, wartości maksymalnej lub minimalnej, używamy jednej z metod `summarizing(Ing|Long|Double)`.
- Te metody pobierają funkcję mapującą obiekty strumienia na liczby i zwracają wynik typu `(Int|Long|Double)SummaryStatistics`, równocześnie obliczając sumę, średnią, maksimum i minimum.

```
IntSummaryStatistics summary = words.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

Strumienie

Przykład (Kolekcje wyników)

- Program `r08/08_08/CollectingResults.java`

Zadanie

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