Wyższa Szkoła Bankowa
Gdańsk Gdynia

Analiza danych w języku Java cz. II

Operacje na strumieniach

Maciej Radzikowski <maciej@radzikowski.com.pl>

Dlaczego Java?

Paradygmaty programowania

Wzorce programowania, określające sposoby rozwiązywania problemów i sterowania przebiegiem programu.

Poszczególne paradygmaty są odpowiednie do danych typów problemów bądź zyskują popularność dzięki braku wad innych paradygmatów.

Paradygmaty są wspierane bądź narzucane przez języki programowania. Niektóre języki mogą wspierać ich kilka.

Paradygmaty programowania w Javie

Podejście imperatywne

Sekwencja instrukcji zmieniających stan programu.

Przypisanie zmiennych, pętle, instrukcje warunkowe.

Skupienie na sposobie, w jaki zadanie ma być wykonane.

Podejście obiektowe

Program jako zbiór komunikujących się ze sobą obiektów, posiadających swój stan (pola) i zachowanie (metody).

Programowanie funkcyjne

Paradygmat zakładający przebieg programu jako szeregu funkcji matematycznych na zbiorach danych.

Skupienie na posiadanych danych, oczekiwanych danych wyjściowych i potrzebnych transformacjach.

Wspierane w Javie od wersji 8 dzięki nowemu Stream API.

Imperative vs. Functional Separation of Concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l -> l.startsWith("ERROR")
        .limit(40)
        .collect(toList());
```

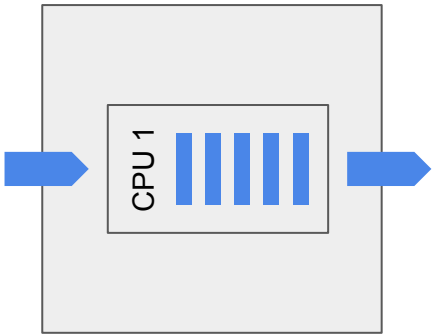
https://twitter.com/mariofusco/status/571999216039542784

Zalety podejścia funkcyjnego w Javie

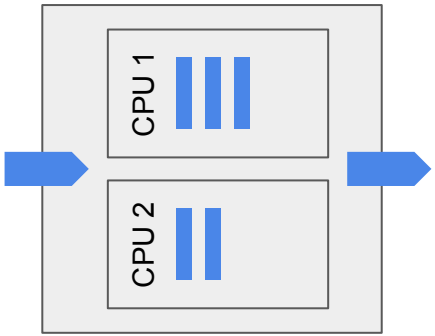
- skupienie na danych
- większa czytelność i zwięzłość kodu
- mniejsze ryzyko błędów programisty związanych ze zmianą stanu
- łatwiejsze przetwarzanie wielowątkowe
- podobieństwo operacji w innych językach i platformach (filter, map, reduce)

Skalowanie przetwarzania danych

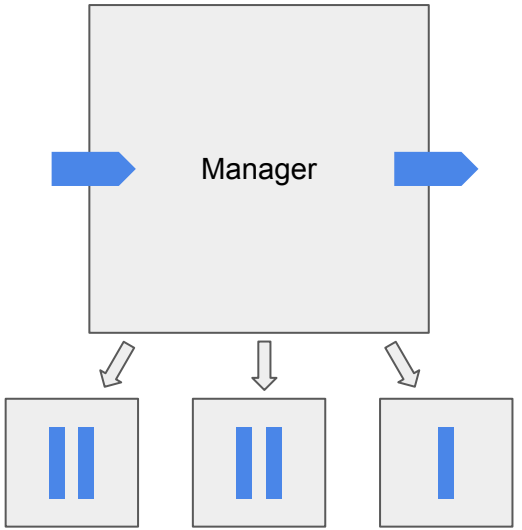
Jednowątkowe (single-core)



Wielowątkowe (multi-core)



Klaster



Java Stream API

Java 8 wprowadza strumienie.

Strumienie mogą być stworzone z:

- kolekcji JCF
- czytanych plików
- generowanych danych

Stream pipeline

Source

```
Stream.of(2, 15, 8, 35, 21, 7)
```

Intermediate operations

```
.filter(x -> x > 10)
.limit(2)
```

Terminal operation

```
.collect(toList())
```

Strumienie i listy

Listy (np. ArrayList) są najczęściej występującym typem danych w aplikacjach, zatem to je najczęściej musimy zamienić na strumień.

Ze strumienia możemy stworzyć nową listę używając gotowego Collectora.

```
List<String> names = Arrays.asList("Bob", "Alice", "Charlie");

Stream<String> namesStream = names.stream();

List<String> newNames =
    namesStream.collect(Collectors.toList());
```

Filtrowanie danych

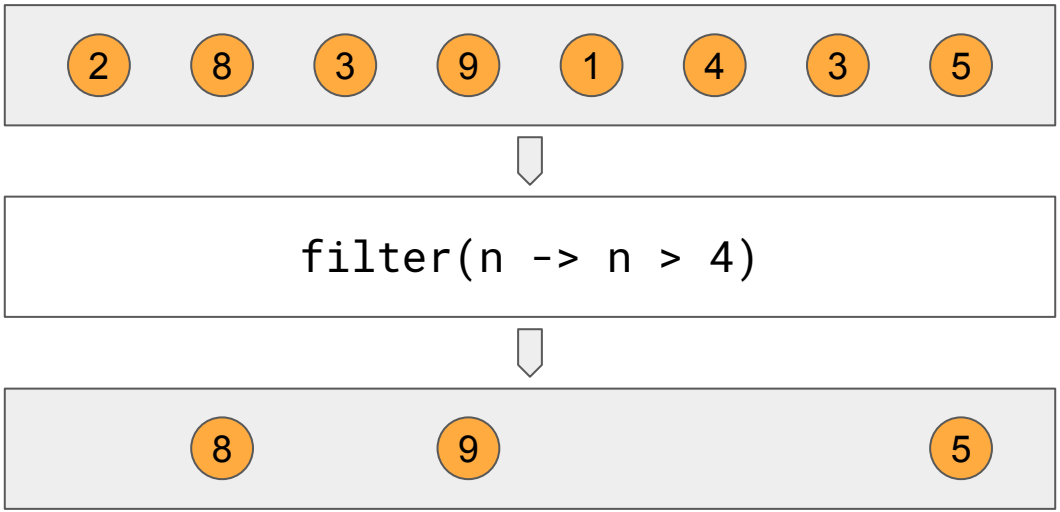
Metoda filter() przyjmuje funkcję, która dla każdego elementu strumienia musi zwrócić wartość true lub false.

Tylko elementy, dla których zwrócono true, będą przekazane dalej w strumieniu.

```
// .filter(item -> boolean)

Stream<String> filteredStream = namesStream
    .filter(name -> name.length() > 3);
```

Filtrowanie danych



Wyrażenia lambda

Wyrażenia lambda pozwalają w skrótowy sposób zaimplementować pojedyncze funkcje, przekazywane np. jako argumenty operacji na strumieniach.

```
name -> name.length() > 3
```

jest odpowiednikiem:

```
public boolean test(String name) {
    return name.length() > 3;
}
```

Numbers Task 1

Mapowanie (konwersja) elementów

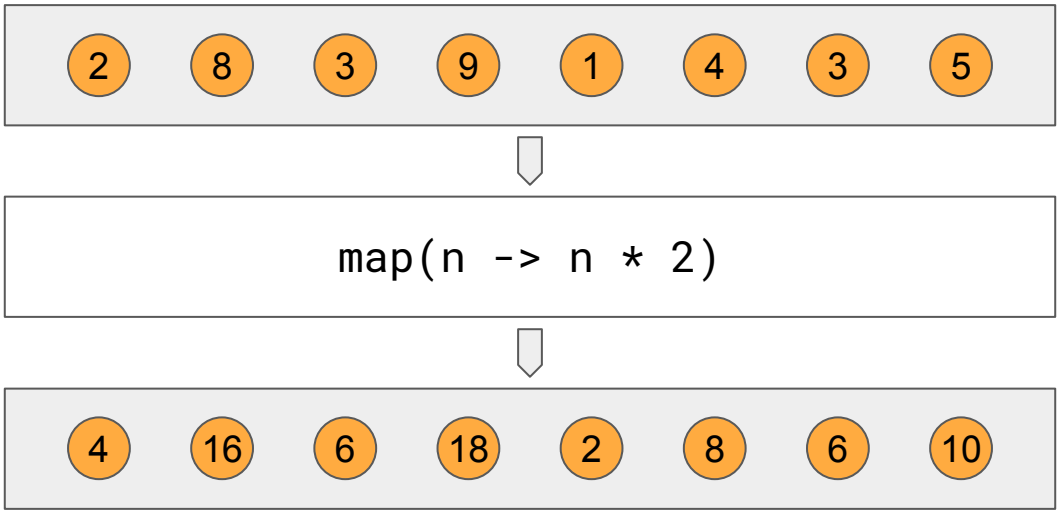
Metoda map() przyjmuje funkcję, która dla każdego elementu strumienia musi zwrócić nową wartość. Ta wartość będzie przekazana dalej w strumieniu.

Nowa wartość nie musi mieć tego samego typu co oryginalna.

```
// .map(item -> otherItem)

Stream<Integer> mappedStream = namesStream
    .map(name -> name.length());
```

Mapowanie (konwersja) elementów



Numbers Task 2

Ograniczanie ilości elementów w strumieniu

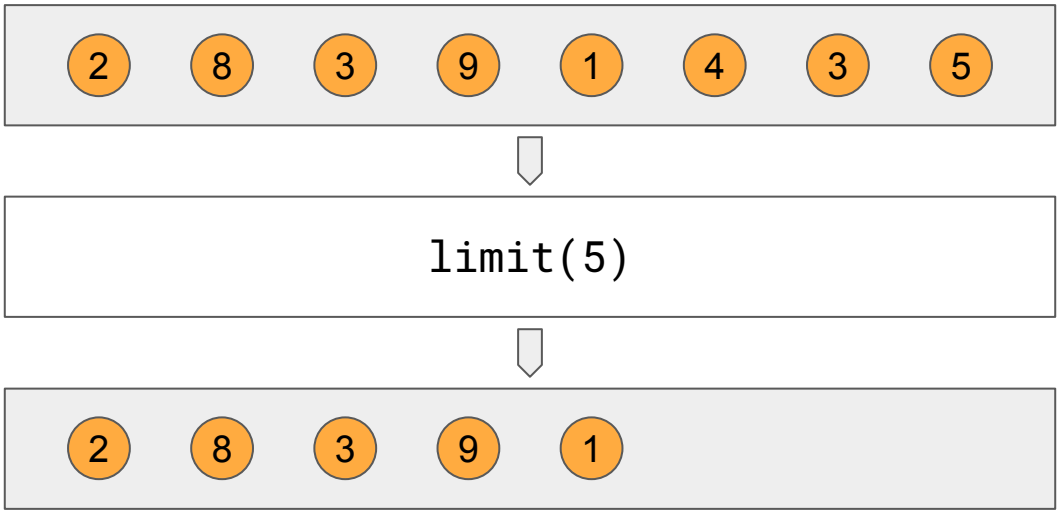
Metoda limit() pozwala podać maksymalną ilość elementów przekazanych dalej w strumieniu.

Jeżeli podana wartość jest większa niż ilość elementów w strumieniu, wszystkie elementy zostaną przekazane dalej.

```
// .limit(long)

Stream<String> shorterStream = namesStream
    .limit(2);
```

Ograniczanie ilości elementów w strumieniu



Numbers Task 3

Łączenie operacji

Operacje pośrednie, takie jak filter(), map() i limit(),
można łączyć i wielokrotnie używać konstruując strumień.

```
Stream<String> longNames = namesStream
    .filter(name -> name.length() > 6)
    .limit(3)
    .map(name -> name.toUpperCase());
```

Movies Tasks

Zliczanie elementów strumienia

Dla uzyskania wyłącznie ilości elementów w strumieniu, a nie samych elementów, można użyć metody kończącej count().

```
long namesCount = namesStream.count();
```

Names Task 1

Funkcje agregujące

Istnieją strumienie liczb (integer / long / double), które posiadają wbudowane dodatkowe metody agregujące: min(), max(), sum(), average().

Strumień liczb można utworzyć ze zwykłego strumienia np. metodą mapToInt().

```
// .mapToInt(item -> int)

OptionalInt namesCount = namesStream
    .mapToInt(name -> name.length())
    .min();
```

Names Task 2

Zbieranie statystyk

Aby uniknąć wielokrotnego tworzenia strumieni i ponownej iteracji przez elementy w celu zebrania różnych statystyk, metoda summaryStatistics() kalkuluje i zwraca wszystkie podstawowe wartości agregujące: count, min, max, average, sum.

```
IntSummaryStatistics namesCount = namesStream
    .mapToInt(name -> name.length())
    .summaryStatistics();
```

Names Task 3

Budowanie tekstu

Collectors.joining() łączy elementy strumienia
i zwraca je jako pojedynczy ciąg tekstowy (String).

Elementy strumienia muszą być typu String.

```
String names = namesStream
    .collect(Collectors.joining());
```

Names Task 4

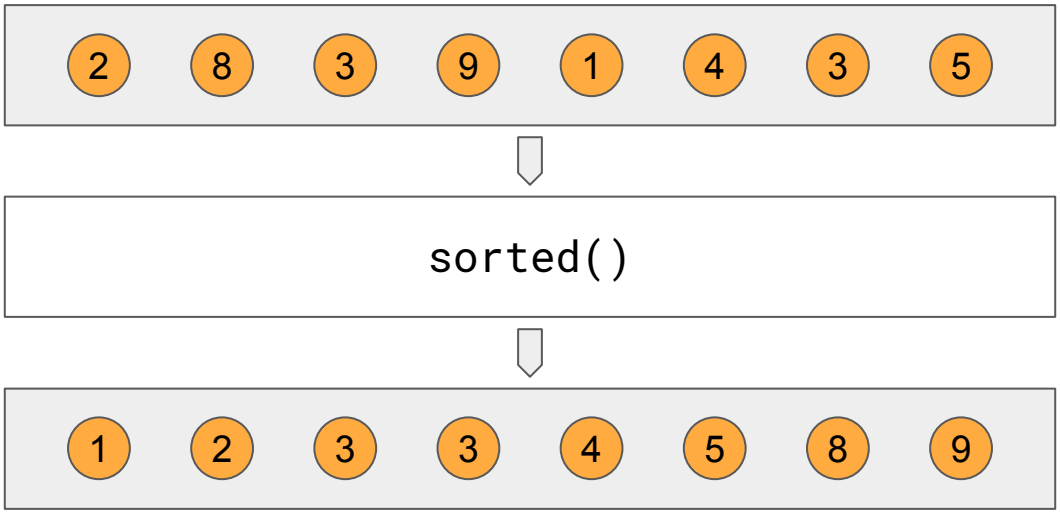
Sortowanie elementów

Metoda sorted() sortuje elementy strumienia.

Opcjonalny parametr określa sposób sortowania.
Możemy go nie podać jeśli elementy posiadają zdefiniowane *naturalne sortowanie* - np. liczby rosnąco a tekst alfabetycznie.

```
Stream<String> = namesStream
    .sorted();
```

Sortowanie elementów



Specyfikacja sortowania

Definicją sortowania może być:

- operacja na elementach zwracająca wartości ujemne, zero i dodatnie dla porównania z elementami mniejszymi, równymi i większymi
- pomocnicza funkcja z klasy Comparator

```
.sorted((a, b) -> a.position() - b.position())
```

```
.sorted(Comparator.comparing(a -> a.position()))
```

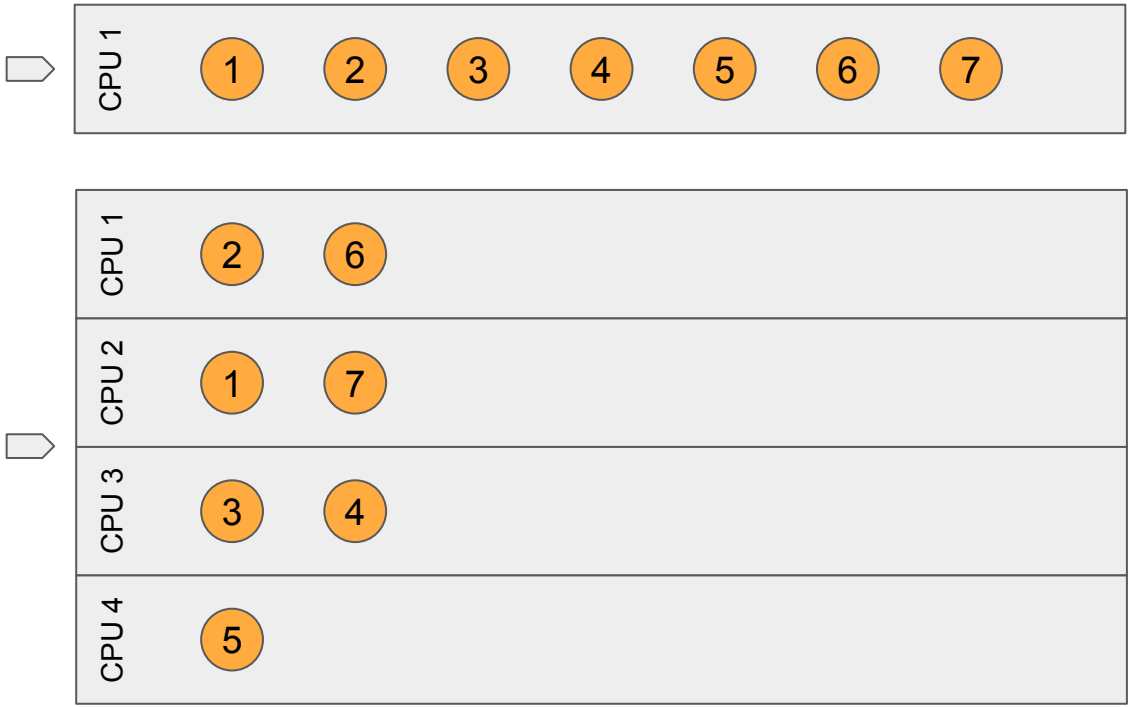
Names Task 5

Przetwarzanie równoległe

Główną motywacją wprowadzenia strumieni do Javy było ułatwienie przetwarzania równoległego, z wykorzystaniem wielu rdzeni procesora.

Strumień oznaczamy jako możliwy do przetwarzania równoległego metodą parallel(). Od implementacji kolejnych operacji (map(), filter(), collect() itd.) zależy, czy i w jaki sposób zostaną one zrównoleglone.

Przetwarzanie sekwencyjne kontra równoległe



Pułapki przetwarzania równoległego

- koszt zarządzania wątkami i zrównoleglania pracy
- niedeterministyczna kolejność przetwarzania
- blokada wątków ze wspólnej puli, używanych przez inne komponenty

Przetwarzanie równoległe powinno być wykorzystywane w reakcji na problemy wydajnościowe, nie domyślnie przed stwierdzeniem problemów.

W zależności od okoliczności przetwarzanie równoległe może zająć więcej czasu niż sekwencyjne.

Parallel Tasks

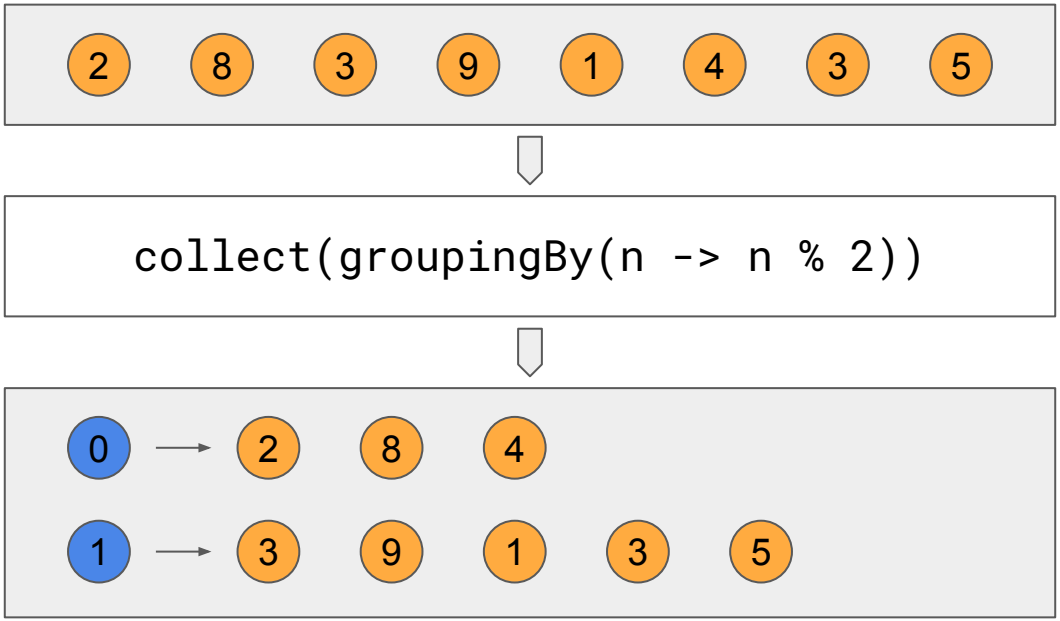
Grupowanie elementów

Grupowanie pozwala zamienić strumień w mapę (asocjację) kluczy do listy wartości.

Służy do tego metoda Collectors.groupingBy(), przyjmująca funkcję zwracającą (obliczającą) klucz pod którym element ma zostać umieszczony.

```
Map<Character, List<String>> namesByFirstLetter = namesStream
    .collect(Collectors.groupingBy(n -> n.charAt(0)));
```

Grupowanie elementów



Tworzenie mapy

Podczas gdy Collectors.groupingBy() tworzy mapę kluczy do list elementów, Collectors.toMap() tworzy mapę klucz - wartość.

Wyliczone klucze muszą być unikalne bądź należy podać sposób rozwiązywania konfliktów jako opcjonalny parametr.

```
Map<String, Integer> namesToLength = namesStream
    .collect(Collectors.toMap(
        name -> name,
        name -> name.length()
    ));
```

BikeRentals Tasks 1 & 2

Redukcja do pojedynczego elementu

Operacja reduce() służy kalkulacji pojedynczego wyniku przez przeprocesowanie kolejno wszystkich elementów strumienia.

Kalkulacji dokonuje się mając do dyspozycji “akumulator” przekazywany między wywołaniami i bieżący element strumienia.

```
// .map(initialAcc, (acc, item) -> newAcc)

String allNames = namesStream
    .reduce(“”, (acc, name) -> acc + “ “ + name);
```

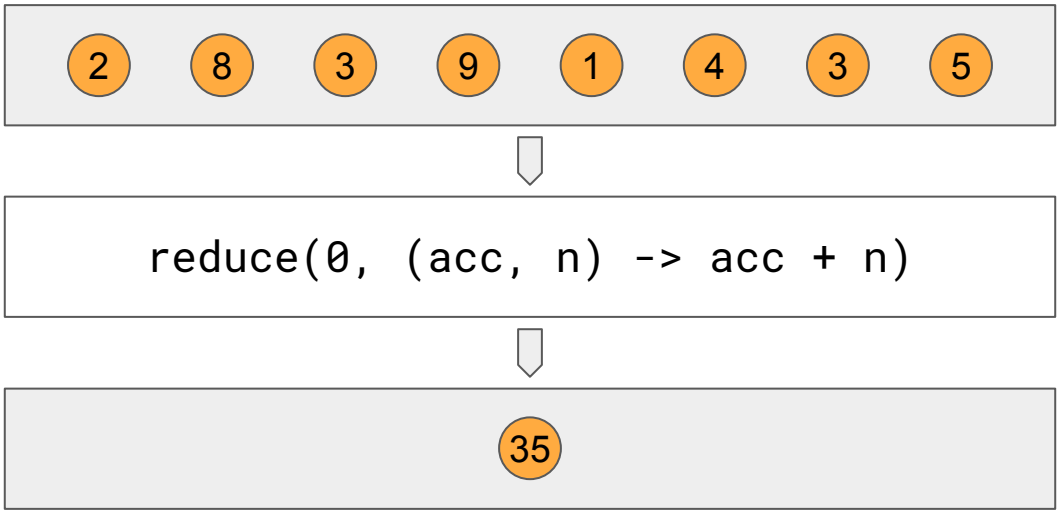
Przykłady użycia redukcji

Za pomocą redukcji możemy uzyskać wyniki wymagające bardziej skomplikowanego procesowania niż za pomocą gotowych Collectorów lub wbudowanych metod statystycznych.

Najprostsze operacje do wykonania za pomocą redukcji:

- suma
- iloczyn
- minimum
- maksimum

Redukcja elementów



Numbers Task 4

BikeRentals Task 3

Zadanie z gwiazdką