Benchmarking Java Streams

Processamento de Dados com Streams de JAVA

Afonso Silva Octávio Maia

9 de Janeiro de 2018

Conteúdo

Parte I.

Testes

1. Cálculo dos valores de transações registadas

Observações

Métodos a testar

```
public double sumArray() {
    double[] values = new double[this.transactions.size()];
    int i = 0;

    for (TransCaixa transaction : this.transactions) {
        values[i++] = transaction.getValor();
    }

    double sum = 0.0;

    for (i = 0; i < values.length; i++) {
        sum += values[i];
    }

    return sum;
}
```

Listing 1.1: Cálculo da soma dos valores das transações atraveś de um array do tipo double

```
public double sumDoubleStream() {
    DoubleStream values = this.transactions.stream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}

public double sumDoubleStreamP() {
    DoubleStream values = this.transactions.parallelStream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}
```

Listing 1.2: Cálculo da soma dos valores das transações através de uma DoubleStream

```
public double sumStream() {
    Stream<Double> values = this.transactions.stream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}

public double sumStreamP() {
    Stream<Double> values = this.transactions.parallelStream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}
```

Listing 1.3: Cálculo da soma dos valores das transações através de Stream<Double>

Resultados

Input	(1) sumArray	(2) sumDoubleStream	(3) sumDoubleStreamP	(4) sumStream	(5) sumStreamP
1000000 transactions	0,010215	0,010794	0,009364	0,026481	0,015645
2000000 transactions	0,021934	0,021800	0,009912	0,053940	0,026226
4000000 transactions	0,044380	0,047799	0,026682	0,113063	0,051981
8000000 transactions	0,078939	0,089719	0,046436	0,214286	0,126348

Análise e conclusões

Após uma breve observação dos gráficos, podemos afirmar que a estrutura de dados mais adequada é de facto a *DoubleStreamP* que implementa streams paralelas. Em contraste, a pior estrutura em nível de performance é a *Stream*, sendo até 5 vezes mais lenta que a *DoubleStreamP*.

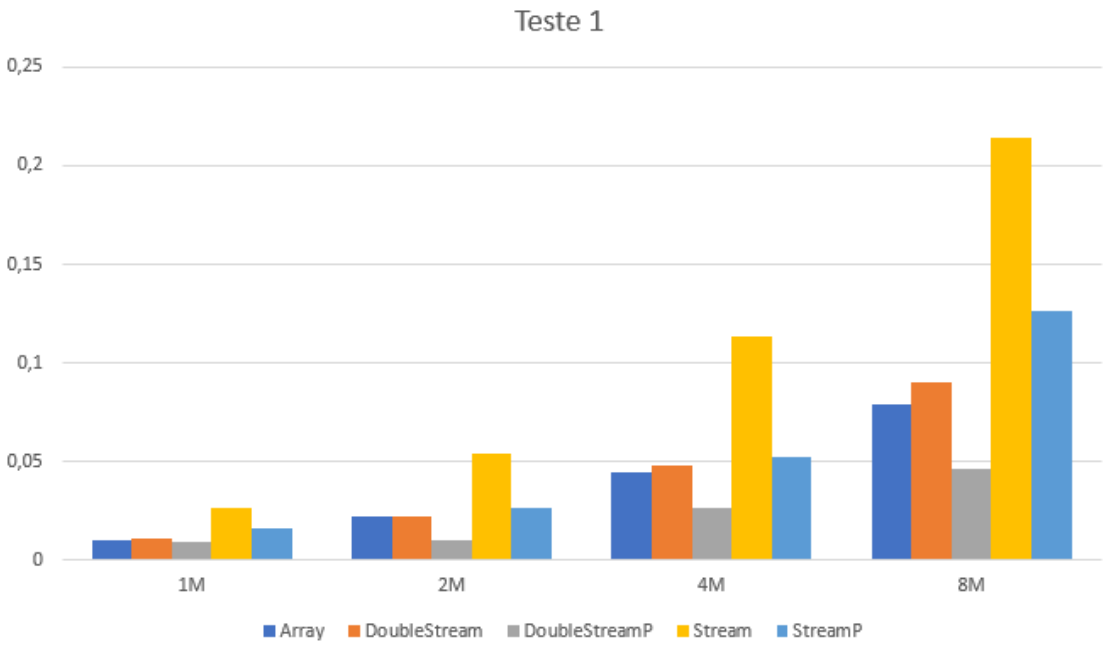

Figura 1.1.: Representação gráfica destes resultados

2. Extração dos primeiros e últimos 20% de transações realizadas

Observações

Métodos a testar

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateList() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> sorted = new ArrayList<>(this.transactions);
    sorted.sort(Comparator.comparing(TransCaixa::getData));

    List<TransCaixa> first = sorted.subList(0, nelems);
    List<TransCaixa> last = sorted.subList(
            sorted.size() - 1 - nelems, sorted.size() - 1);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateSet() {
    int nelems = 20 * this.transactions.size() / 100;

    TreeSet<TransCaixa> sorted = new TreeSet<>(
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1
    );
    sorted.addAll(this.transactions);

    List<TransCaixa> first = new ArrayList<>(sorted)
            .subList(0, nelems);
    List<TransCaixa> last = new ArrayList<>(sorted.descendingSet())
            .subList(0, nelems);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStream() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
```

```
            .sorted(Comparator.comparing(TransCaixa::getData))
            .limit(nelems)
            .collect(Collectors.toList());
    List<TransCaixa> last = this.transactions.stream()
            .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
            .limit(nelems)
            .collect(Collectors.toList());

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStreamP() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
            .sorted(Comparator.comparing(TransCaixa::getData))
            .limit(nelems)
            .collect(Collectors.toList());
    List<TransCaixa> last = this.transactions.stream()
            .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
            .limit(nelems)
            .collect(Collectors.toList());

    return new SimpleEntry<>(first, last);
}
```

Resultados

Input	(1) byDateList	(2) byDateSet	(3) byDateStream	(4) byDateStreamP
1000000 transactions	1,029137	1,939311	1,861916	2,239986
2000000 transactions	2,331822	3,706627	3,858658	4,378531
4000000 transactions	5,745454	9,411794	9,281572	10,556521
8000000 transactions	12,635500	21,085741	20,579760	22,002980

Análise e conclusões

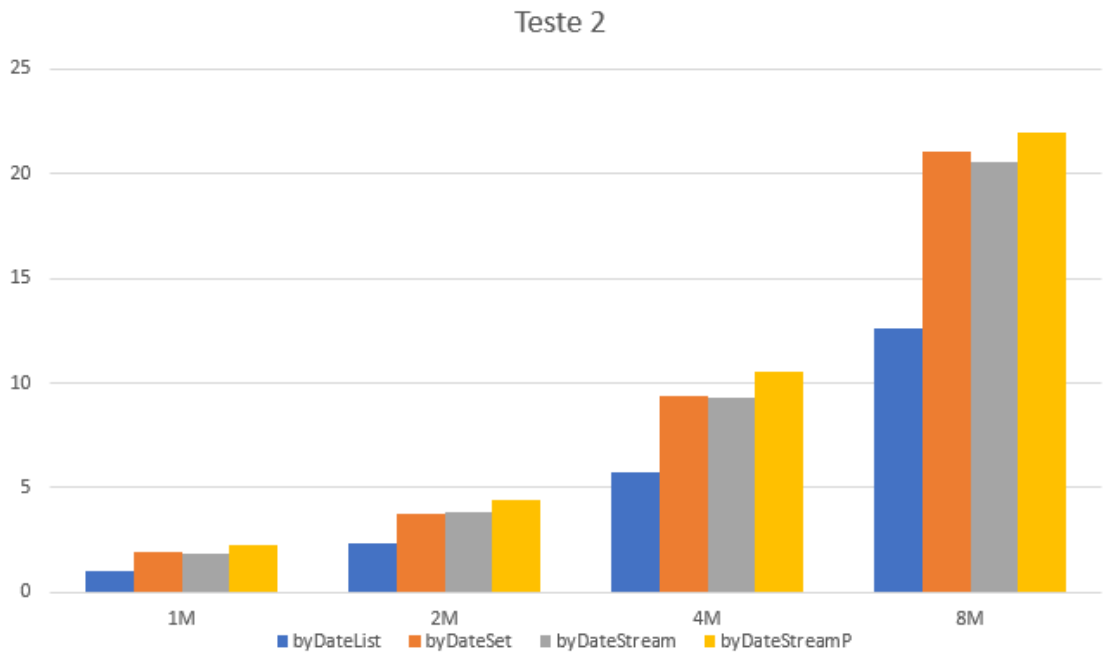

Figura 2.1.: Representação gráfica destes resultados

3. Esforço de eliminação de duplicados

Observações

Métodos a testar

```
public Integer[] uniqueArray() {
    Set<Integer> nodups = new TreeSet<>();
    for (int value : this.values) {
        nodups.add(value);
    }

    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.1: Eliminação dos duplicados através de um array de inteiros

```
public Integer[] uniqueList() {
    List<Integer> aux = new ArrayList<>();
    for (int value : this.values) {
        aux.add(value);
    }

    List<Integer> nodups = new ArrayList<>(new HashSet<>(aux));
    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.2: Eliminação dos duplicados através de uma lista de inteiros

```
public int[] uniqueIntStream() {
    IntStream values = new Random().ints(this.values.length, 0, 9999);
    return values.distinct().toArray();
}
```

Listing 3.3: Eliminação dos duplicados através de uma stream de inteiros

Resultados

Input	(1) uniqueArray	(2) uniqueList	(3) uniqueIntStream
1000000 random numbers	0,130102	0,027098	0,025731
2000000 random numbers	0,253295	0,056908	0,048782
4000000 random numbers	0,483225	0,094416	0,087328
8000000 random numbers	1,033746	0,223069	0,203687

Análise e conclusões

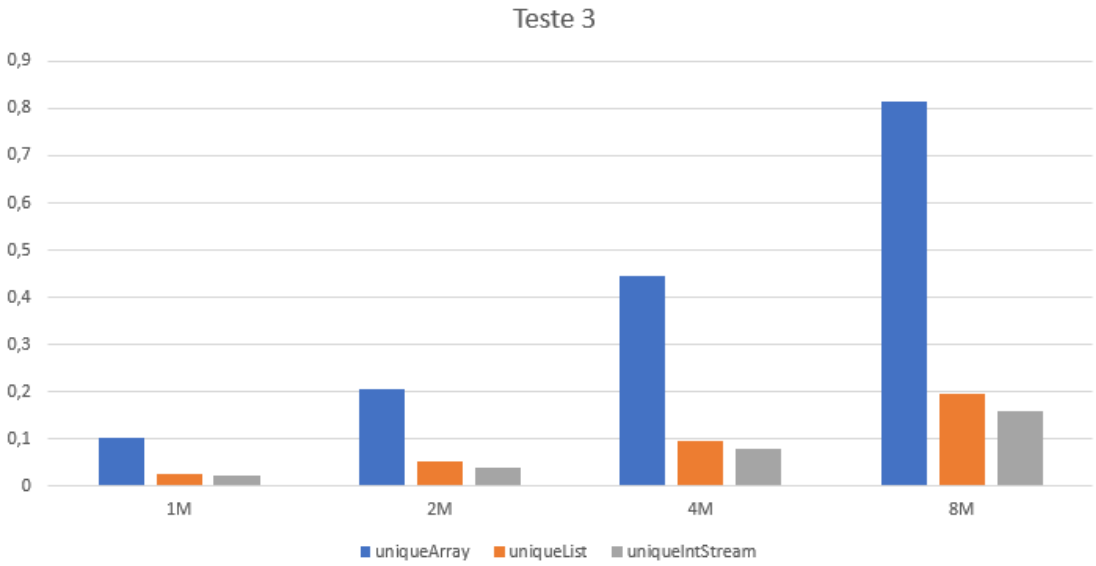

Figura 3.1.: Representação gráfica destes resultados

4. Comparação entre a aplicação de método estático, BiFunction e Lambda

Observações

Métodos a testar

```
public static int div(int x, int y) {
    return x / y;
}

public int[] divSMethodStream() {
    return Arrays.stream(this.values)
        .map(x -> div(x, 2)).toArray();
}

public int[] divSMethodStreamP() {
    return Arrays.stream(this.values).parallel()
        .map(x -> div(x, 2)).toArray();
}
```

Listing 4.1: Divisão de todos os números por 2 através de um método estático

```
public int[] divBiFunStream() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values)
        .map(x -> f.apply(x, 2)).toArray();
}

public int[] divBiFunStreamP() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values).parallel()
        .map(x -> f.apply(x, 2)).toArray();
}
```

Listing 4.2: Divisão de todos os números por 2 através de uma BiFunction

```
public int[] divLambdaStream() {
    return Arrays.stream(this.values).map(x -> x / 2).toArray();
}

public int[] divLambdaStreamP() {
```

```
    return Arrays.stream(this.values).parallel().map(x -> x / 2).toArray();
}
```

Listing 4.3: Divisão de todos os números por 2 através de um Lambda

Resultados

1000000 transactions	1,005284	1,529525	1,639273	1,830654
2000000 transactions	2,284556	3,531155	3,664016	4,204185
4000000 transactions	5,314449	8,259452	8,455954	9,671057
8000000 transactions	11,686906	20,374520	18,379551	21,186783

Análise e conclusões

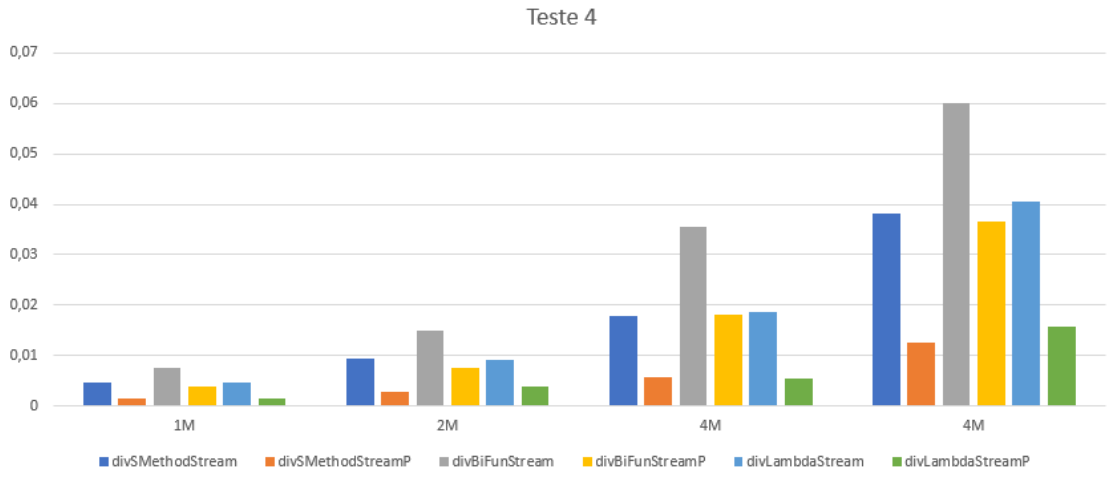

Figura 4.1.: Representação gráfica destes resultados

5. Comparação da ordenação através de um TreeSet e do método `sorted`

Observações

Métodos a testar

```
public TreeSet<TransCaixa> sortTreeSet() {
    Comparator<TransCaixa> byDate =
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1;

    return this.transactions.stream()
        .collect(Collectors.toCollection(
                () -> new TreeSet<>(byDate)));
}
```

Listing 5.1: Ordenação através de um TreeSet

```
public List<TransCaixa> sortList() {
    Comparator<TransCaixa> byDate =
        Comparator.comparing(TransCaixa::getData);

    return this.transactions.stream()
        .sorted(byDate).collect(Collectors.toList());
}
```

Listing 5.2: Ordenação através do método sorted

Resultados

Input	(1) sortTreeSet	(2) sortList
1000000 transactions	1,510102	0,849679
2000000 transactions	3,449318	1,840600
4000000 transactions	8,242589	4,047586
8000000 transactions	19,584347	8,851000

Análise e conclusões

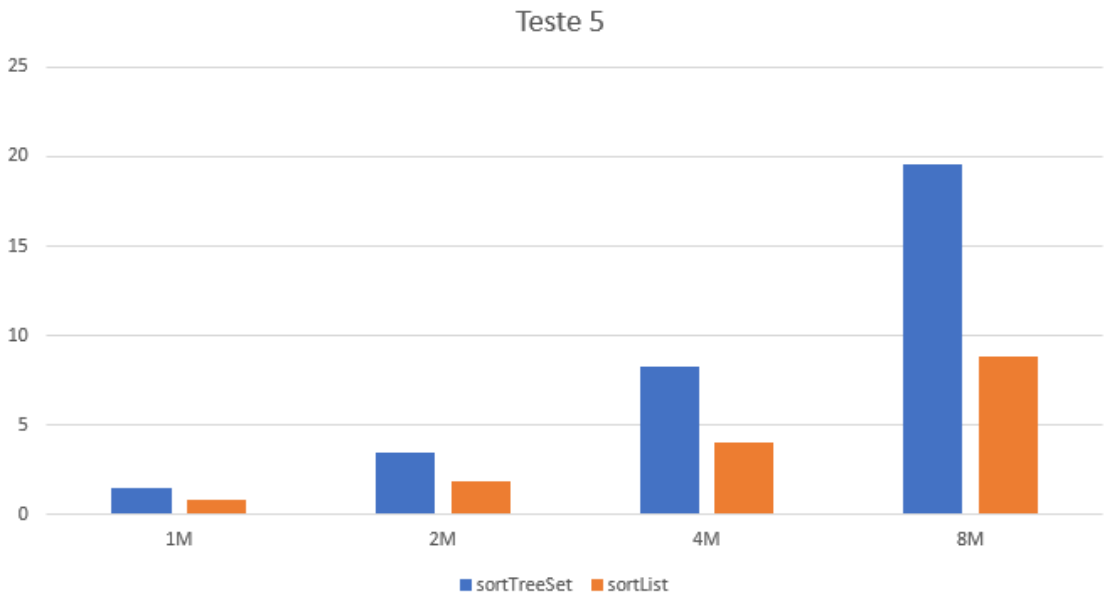

Figura 5.1.: Representação gráfica destes resultados

6. Determinação da maior transação entre uma determinada hora

Observações

Métodos a testar

```
String biggestTransaction7() {
    List<TransCaixa> transactions = new ArrayList<>(this.transactions);

    transactions.sort(new Comparator<TransCaixa>() {
        @Override
        public int compare(TransCaixa t1, TransCaixa t2) {
            return Double.compare(t1.getValor(), t2.getValor());
        }
    });

    for (TransCaixa transaction : transactions) {
        int hour = transaction.getData().getHour();

        if (hour >= 16 && hour <= 20) {
            return transaction.getTrans();
        }
    }

    return null;
}
```

Listing 6.1: Determinação da maior transação entre uma determinada hora apenas com funcionalidades do JAVA7

```
Optional<String> biggestTransaction8() {
    t.getData().getHour() >= 16 && t.getData().getHour() <= 20;

    return this.transactions.stream()
        .filter(timeInRange)
        .max(Comparator.comparing(TransCaixa::getValor))
        .map(TransCaixa::getTrans);
}
```

Listing 6.2: Determinação da maior transação entre uma determinada hora com auxílio de Streams

Resultados

Input	(1) biggestTransaction7	(2) biggestTransaction8
1000000 transactions	0,255225	0,017511
2000000 transactions	0,598334	0,037735
4000000 transactions	1,174918	0,182393
8000000 transactions	2,133888	0,144549

Análise e conclusões

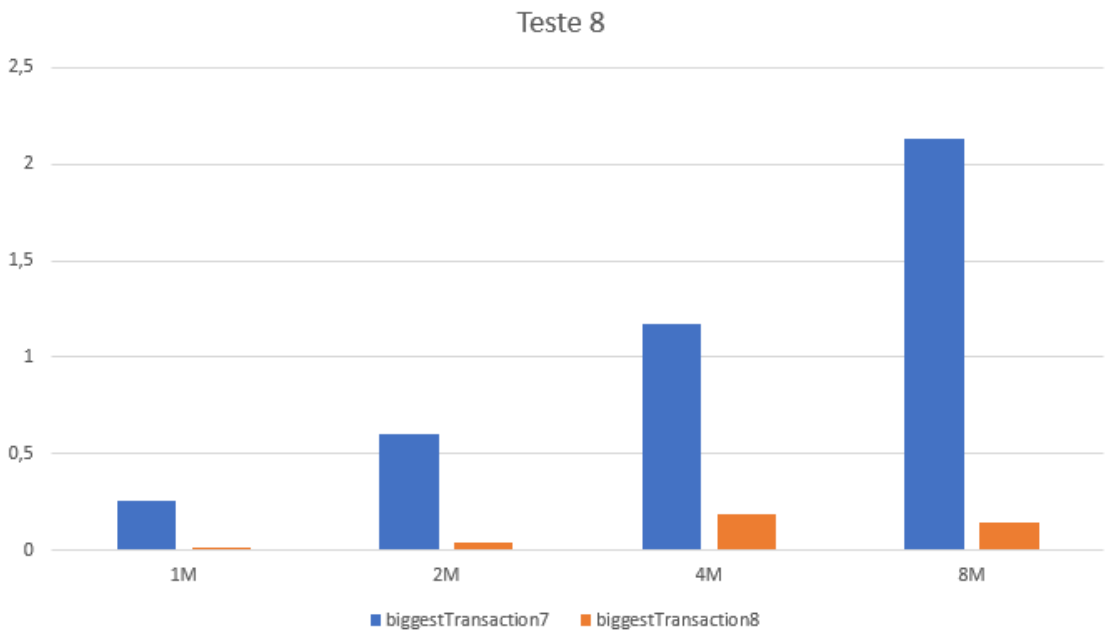

Figura 6.1.: Representação gráfica destes resultados

7. Cálculo do total facturado numa determinada semana do ano

Observações

Métodos a testar

```
public double totalInWeekList() {
    final int week = 12;
    List<List<TransCaixa>> byWeek = new ArrayList<>();

    // Inicializar cada uma das listas
    for (int i = 0; i < 54; i++) {
        byWeek.add(i, new ArrayList<>());
    }

    for (TransCaixa transaction : this.transactions) {
        byWeek.get(transaction.getData()
                .get(ChronoField.ALIGNED_WEEK_OF_YEAR))
                .add(transaction);
    }

    // Calcular total faturado
    double total = 0.0;

    for (TransCaixa transaction : byWeek.get(week)) {
        total += transaction.getValor();
    }

    return total;
}
```

Listing 7.1: Cálculo do total facturado na semana 12 do ano

```
public double totalInWeekStream() {
    final int week = 12;
    Map<Integer, List<TransCaixa>> byWeek = this.transactions.stream()
            .collect(Collectors.groupingBy(
                    t -> t.getData().get(ChronoField.ALIGNED_WEEK_OF_YEAR)));
    return byWeek.entrySet().stream()
            .filter(e -> e.getKey() == week)
            .findFirst()
            .map(e -> e.getValue().stream()
```

```
                .mapToDouble(TransCaixa::getValor).sum())
            .orElse(0.0);
}
```

Listing 7.2: Cálculo do total facturado na semana 12 do ano com recurso a streams

Resultados

Input	(1) totalInWeekList	(2) totalInWeekStream
1000000 transactions	0,047309	0,071204
2000000 transactions	0,088859	0,120726

Análise e conclusões

8. Cálculo do total de IVA associado a cada mês

Observações

Métodos a testar

```
public List<Double> iva() {
    List<Double> ivas = new ArrayList<>(13);

    // Initialize the IVAS with 0
    for (int i = 0; i < 13; i++) {
        ivas.add(0.0);
    }

    for (TransCaixa transaction : this.transactions) {
        Month month = transaction.getData().getMonth();
        Double value = transaction.getValor();
        Double iva = value < 20                ? 0.15 * value :
                     (value > 20 && value < 20 ? 0.20 * value :
                                                 0.23 * value);
        ivas.set(month.getValue(), ivas.get(month.getValue()) + iva);
    }

    return ivas;
}
```

Listing 8.1: Cálculo do total de IVA para cada mês

```
public Map<Month, Double> ivaStream() {
    return this.transactions.stream()
            .collect(Collectors.groupingBy(t -> t.getData().getMonth()))
            .entrySet()
            .stream()
            .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    e -> e.getValue().stream()
                            .mapToDouble(TransCaixa::getValor)
                            .map(x -> x < 20             ? 0.15 * x :
                                      (x > 20 && x < 20 ? 0.20 * x :
                                                          0.23 * x ))
                            .sum()
            ));
```

```
}
```

Listing 8.2: Cálculo do total de IVA para cada mês, com recurso a streams

Resultados

Input	(1) iva	(2) ivaStream
1000000 transactions	0,026395	0,120314
2000000 transactions	0,050004	0,135051

Análise e conclusões