ChatGPT

Conjunto de Exercícios – Java Stream API Avançado

Seção 1 – filter (5 questões)

- **1.1.** Em um cenário de e-commerce, temos uma lista de pedidos com seu status. Considere a classe `Pedido` com atributos `id` e `status` (por exemplo, "PENDENTE", "ENTREGUE", etc.). Suponha a lista:

```
class Pedido { int id; String status; /* construtor */ }
List<Pedido> pedidos = Arrays.asList(
    new Pedido(1, "PENDENTE"),
    new Pedido(2, "ENTREGUE"),
    new Pedido(3, "PENDENTE"),
    new Pedido(4, "ENTREGUE")
);
```

Pergunta: Utilizando **Stream.filter**, obtenha uma lista contendo apenas os pedidos cujo status é `"ENTREGUE"`. (Dica: lembre-se de realizar uma operação terminal para materializar o resultado.)

- **1.2.** Você mantém uma lista de usuários de um sistema, onde cada usuário tem atributos como `nome`, `idade` e `ativo` (indicando se o usuário está ativo no sistema). Por exemplo:

```
class Usuario { String nome; int idade; boolean ativo; /* construtor */ }
List<Usuario> usuarios = Arrays.asList(
    new Usuario("Ana", 23, true),
    new Usuario("Bruno", 16, false),
    new Usuario("Carla", 34, true),
    new Usuario("Daniel", 28, true)
);
```

Pergunta: Usando **filter**, obtenha os usuários que **estão ativos e** têm **idade maior ou igual a 18**. Quantos usuários atendem a esses critérios?

- **1.3.** Considere o seguinte trecho de código que tenta filtrar pedidos entregues:

```
List<Pedido> entregues = pedidos.stream()
    .filter(p -> p.getStatus().equals("ENTREGUE"));
System.out.println("Pedidos entregues: " + entregues);
```

Ao executar, a saída não lista os pedidos entregues conforme esperado, exibindo algo semelhante a `Pedidos entregues: java.util.stream.ReferencePipeline$3@6d06d69c`. **Pergunta:** Por que

o código acima não funcionou como pretendido? O que falta para que a filtragem seja de fato executada e os pedidos entregues sejam obtidos?

- **1.4.** Usando Streams, crie um **stream infinito** de números inteiros positivos e filtre apenas aqueles divisíveis por 7. Então, obtenha o **primeiro número maior que 100** que atenda a essa condição. *(Dica: utilize* `Stream.iterate` *para gerar os números e uma operação terminal adequada para obter um único resultado.)*

- **1.5.** Suponha que a classe `Produto` possua um método `getPrecoFinal()` computacionalmente custoso (por exemplo, aplica vários cálculos e acessos externos). Você precisa encontrar os nomes dos produtos com preço final acima de R$ 1000, utilizando Streams. Considere duas abordagens:

A) `produtos.stream().map(p -> p.getPrecoFinal()).filter(preco -> preco > 1000)...`
B) `produtos.stream().filter(p -> p.getPrecoFinal() > 1000).map(Produto::getNome)...`

Pergunta: Qual das duas abordagens é mais eficiente e por quê? O que isso demonstra sobre a ordem das operações `filter` e `map` em um pipeline de Stream?

Seção 2 – map (5 questões)

- **2.1.** Considere a classe `Cliente` com atributos `nome` e `email`. Você tem uma lista de clientes:

```
class Cliente { String nome; String email; /* construtor */ }
List<Cliente> clientes = Arrays.asList(
    new Cliente("Alice", "alice@example.com"),
    new Cliente("Bruno", "bruno99@mail.com"),
    new Cliente("Carlos", "carlos_silva@mail.com")
);
```

Pergunta: Utilizando **Stream.map**, obtenha uma **lista de Strings** contendo apenas os endereços de email dos clientes.

- **2.2.** Dada uma lista de preços (valores `Double`):

```
List<Double> precos = Arrays.asList(50.0, 100.0, 35.5, 70.0);
```

Pergunta: Use **map** para aplicar um aumento de 10% em cada preço, gerando uma nova lista com os preços reajustados. Por exemplo, `50.0` deve se tornar `55.0`. (Não modifique a lista original, crie uma nova com os valores alterados.)

- **2.3.** Você possui uma lista de números em formato de texto (Strings):

```
List<String> numerosStr = Arrays.asList("10", "20", "30", "40");
```

Pergunta: Use **map** para converter essa lista de Strings em uma lista de `Integer`. (Desconsidere possíveis formatos inválidos na lista; assuma que todas as strings são números válidos.)

- **2.4.** Considere uma lista de usuários e um método `enviarEmail(Usuario u)` que envia um email para o usuário e retorna um booleano indicando sucesso. Um desenvolvedor tentou enviar emails a todos os usuários com o código abaixo, mas relatou que nenhum email foi enviado:

```
usuarios.stream().map(u -> enviarEmail(u));
```

Pergunta: Por que esse código não enviou os emails? Como corrigir o uso da Stream API para realizar uma ação (envio de email) para cada usuário da lista? Explique a diferença entre **map** e **forEach** nesse contexto.

- **2.5.** Suponha que você tenha uma lista muito grande de números inteiros e precise aplicar uma função pesada de cálculo em cada número (por exemplo, uma operação matemática complexa). Para melhorar a performance, deseja-se utilizar processamento paralelo. **Pergunta:** Como você utilizaria Streams para aplicar essa função em todos os elementos o mais rápido possível? Esboce um código usando **streams paralelos** que aplique a função `computarAlgo(int x)` (imaginária e custosa) em cada elemento de uma lista grande de inteiros. Comente brevemente sobre em que cenários `parallelStream()` tende a melhorar a performance.

Seção 3 – flatMap (5 questões)

- **3.1.** Considere que você tem listas dos funcionários de diferentes departamentos de uma empresa, representadas por um `List<List<String>>` onde cada sublista contém os nomes dos funcionários daquele departamento:

```
List<List<String>> equipes = Arrays.asList(
    Arrays.asList("Alice", "Bob"),
    Arrays.asList("Carlos", "Diana", "Eric"),
    Arrays.asList("Fabiana")
);
```

Pergunta: Use **flatMap** para transformar essa estrutura em uma única lista contendo **todos os nomes de funcionários** da empresa (ou seja, “achate” a lista de listas em uma lista única).

- **3.2.** Suponha a existência das classes:

```
class Item { String nome; double preco; /* construtor */ }
class Order { int id; List<Item> itens; /* construtor */ }
```

e uma lista `List<Order> orders` onde cada pedido possui uma lista de itens comprados. **Pergunta:** Utilize **flatMap** para obter uma lista **única** contendo **todos os itens** de todos os pedidos. (Por exemplo, se cada Order possui seus itens, o resultado deve ser a união de itens de todos os pedidos.)

- **3.3.** Você tem uma lista de strings em formato CSV (valores separados por vírgula), onde cada string contém uma série de palavras separadas por vírgulas:

```
List<String> linhas = Arrays.asList("java,python,c", "ruby,javascript", "c++,go");
```

Pergunta: Use **flatMap** para obter uma lista de **todos os valores individuais** nessas strings. Ou seja, a saída desejada para o exemplo acima seria uma lista: `["java","python","c","ruby","javascript","c++","go"]`.

- **3.4.** Qual a diferença prática entre usar **map** e **flatMap**? Considere um exemplo:

```
List<List<Integer>> matriz = Arrays.asList(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4)
);
```

Se fizermos `matriz.stream().map(list -> list.stream())`, qual será o tipo do resultado e como ele difere do resultado de `matriz.stream().flatMap(list -> list.stream())`? **Pergunta:** Explique o resultado de cada abordagem e qual delas deve ser usada para obter um `Stream<Integer>` com **todos** os números da matriz.

- **3.5.** Suponha que você tem uma lista `List<Optional<String>>` contendo valores opcionais (alguns presentes, outros vazios). Por exemplo:

```
List<Optional<String>> talvezNomes = Arrays.asList(
    Optional.of("Ana"),
    Optional.empty(),
    Optional.of("Bia"),
    Optional.empty(),
    Optional.of("Caio")
);
```

Pergunta: Utilizando Streams, obtenha uma lista de **Strings** com todos os valores presentes em `talvezNomes`, descartando os opcionais vazios. (Dica: há várias maneiras de fazer, mas tente pensar em como **flatMap** pode ajudar a simplificar a operação com `Optional`.)

Seção 4 – reduce (5 questões)

- **4.1.** Dada a lista de inteiros:

```
List<Integer> numeros = Arrays.asList(5, 10, 15, 20);
```

Pergunta: Use **Stream.reduce** para calcular a **soma** de todos os números da lista. (Não use `IntStream.sum()` ou `Collectors.summingInt`; a ideia é praticar o uso do reduce manualmente.)

- **4.2.** Considere a lista de nomes de clientes:

```
List<String> nomes = Arrays.asList("Mariana", "João", "Alexandre", "Bianca");
```

Pergunta: Utilize **reduce** para encontrar o **nome mais longo** presente na lista. (Dica: você pode usar uma `String` vazia `""` como elemento identidade e, na função acumuladora, sempre reter a string de maior comprimento.)

- **4.3.** Dada uma lista de palavras, por exemplo:

```
List<String> palavras = Arrays.asList("java", "stream", "api", "colecoes");
```

Pergunta: Use **reduce** para concatenar todas as palavras em uma única string, separadas por vírgula. *(Desafio: cuide para que a string resultante não comece nem termine com vírgula extra – por exemplo, "java, stream, api, colecoes".)*

- **4.4.** Considere uma lista de frases (Strings) e queremos saber o total de caracteres de todas as frases somadas. Por exemplo:

```
List<String> frases = Arrays.asList("Olá mundo", "Java Streams", "reduce");
```

Pergunta: Use **reduce** para calcular o **número total de caracteres** de todas as strings da lista `frases`. (Dica: você pode utilizar a versão de reduce que recebe três argumentos – identidade, acumulador e combinador – para fazer a redução diretamente de `String` para um contador `int`.)

- **4.5.** Suponha a classe `Transacao`:

```
class Transacao { double valor; /* + outros campos, construtor */ }
```

e uma lista `List<Transacao> transacoes`. Queremos derivar algumas estatísticas simples dessas transações. **Pergunta:** Implemente, usando **reduce**, a computação de um objeto `Stats` que contenha o **total de transações (count)** e a **soma dos valores** de todas elas. Defina a classe auxiliar `Stats` com campos `count` (long) e `sum` (double), e utilize `reduce` com identidade, acumulador e combinador para produzir um único `Stats` agregando esses resultados. *(Dica: Inicie com* `new Stats(0, 0.0)` *como identidade e, no acumulador, incremente o contador e adicione o valor da transação. Lembre-se que a função de combinação deve mesclar dois objetos* `Stats`*.)*

Seção 5 – collect e Collectors (5 questões)

- **5.1.** Dada a lista de números inteiros:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
```

Pergunta: Use **filter** e então **collect** para obter uma `List<Integer>` contendo apenas os números pares da lista original. (Exemplo de resultado esperado para a lista acima: `[2, 4, 6]`.)

- **5.2.** Suponha uma lista de palavras (possivelmente com repetições e variações de maiúsculas/minúsculas):

```
List<String> palavras = Arrays.asList("Java", "java", "STREAM", "Api",
"Java");
```

Pergunta: Usando **Collectors.toSet** (ou outra estratégia similar), obtenha um **conjunto** (`Set<String>`) com as palavras em **letras minúsculas e sem duplicatas**. Para a lista de exemplo, o resultado esperado seria um conjunto como `{"java", "stream", "api"}`.

- **5.3.** Considere uma lista de nomes de linguagens de programação:

```
List<String> linguagens = Arrays.asList("Java", "Python", "C", "JavaScript");
```

Pergunta: Utilize **Collectors.joining** para concatenar todos os nomes em uma única string separados por **", "** (vírgula e espaço). Por exemplo: `"Java, Python, C, JavaScript"`.

- **5.4.** Você tem uma lista de valores de vendas mensais (números `double`). Por exemplo:

```
List<Double> vendas = Arrays.asList(1000.0, 2500.5, 3100.0, 1500.75);
```

Pergunta: Use **Collectors.summarizingDouble** para obter de uma só vez o **total de vendas, média, valor mínimo e máximo**, bem como a quantidade de entradas. Explique brevemente o conteúdo do objeto retornado (um `DoubleSummaryStatistics`) e exiba seus principais valores.

- **5.5.** Considere uma lista de frutas:

```
List<String> frutas = Arrays.asList("maçã", "banana", "laranja", "banana");
```

Pergunta: Usando **Collectors.collectingAndThen**, obtenha uma **lista imutável** (não modificável) contendo as frutas em **ordem alfabética** *sem duplicatas*. (Dica: você pode coletar em um `Set` ou usar `distinct()` para facilitar, e então usar `collectingAndThen` com `Collections.unmodifiableList`.)

Seção 6 – groupingBy / partitioningBy (5 questões)

- **6.1.** Considere a classe `Employee` (funcionário) definida abaixo e a lista correspondente:

```
class Employee {
    String nome;
    String departamento;
    int idade;
    double salario;
    boolean ativo;
    Employee(String n, String dept, int idade, double sal, boolean ativo) {
        this.nome = n; this.departamento = dept;
        this.idade = idade; this.salario = sal; this.ativo = ativo;
    }
}
```

```
List<Employee> employees = Arrays.asList(
    new Employee("Alice", "IT",    28, 3000.0, true),
    new Employee("Bob",   "Sales", 45, 5000.0, false),
    new Employee("Carol", "IT",    35, 4500.0, true),
    new Employee("David", "HR",    52, 4500.0, true),
    new Employee("Eve",   "Sales", 29, 3500.0, true)
);
```

Pergunta: Usando **Collectors.groupingBy**, agrupe os funcionários por **departamento**. O resultado deve ser um `Map<String, List<Employee>>` onde a chave é o nome do departamento e o valor é a lista de funcionários daquele departamento. Quais departamentos existem na lista e quantos funcionários em cada um?

- **6.2.** Dada uma lista de palavras:

```
List<String> palavras = Arrays.asList("bike", "carro", "navio", "avião",
"trem");
```

Pergunta: Use **groupingBy** para agrupar as palavras pelo **tamanho (número de letras)**. Ou seja, produza um `Map<Integer, List<String>>` em que a chave é o tamanho da palavra e o valor é a lista de palavras com aquele tamanho. (Exemplo: tamanho 5 -> ["navio", "avião"] etc.)

- **6.3.** Usando novamente a lista `employees` definida em 6.1, agrupe os funcionários por **faixa etária**. Defina as faixas conforme a idade: **menores de 30 anos**, **entre 30 e 50 anos**, e **maiores de 50 anos**.

Pergunta: Produza um `Map<String, List<Employee>>` com três entradas (por exemplo, chaves "Jovens", "Experientes", "Seniors"), e liste quais funcionários caem em cada faixa etária. *(Dica: você pode passar uma função lambda para groupingBy que verifica a idade de cada empregado e retorna a string categoria apropriada.)*

- **6.4.** Usando a mesma lista de `employees`, queremos separar os funcionários **ativos** dos **inativos**.

Pergunta: Utilize **Collectors.partitioningBy** para gerar um `Map<Boolean, List<Employee>>` onde a chave `true` corresponde à lista de funcionários ativos e `false` à lista de inativos. Quantos funcionários ativos e inativos existem respectivamente?

- **6.5.** Ainda com a lista `employees`, suponha que desejamos saber o **total de salário** pago por departamento.

Pergunta: Utilize **groupingBy** com um **Collector de redução** (por exemplo, `Collectors.summingDouble`) para criar um mapa do tipo `Map<String, Double>` associando cada departamento à soma dos salários de seus funcionários. (Ex: "IT" -> 7500.0, "Sales" -> 8500.0, "HR" -> 4500.0 com base na lista de exemplo.)

Seção 7 – sorted / matchers (5 questões)

- **7.1.** Dada a lista de nomes:

```
List<String> nomes = Arrays.asList("Carlos", "Ana", "Bruno", "Eduarda");
```

Pergunta: Use **Stream.sorted()** para obter os nomes em **ordem alfabética crescente**. Qual seria a ordem resultante? (Obs: *Considere a ordenação lexicográfica padrão do Java.*)

- **7.2.** Considere novamente a lista `employees` (com campos nome, salário, etc. conforme definido na seção 6.1). **Pergunta:** Utilize **sorted** com um comparador customizado para ordenar os funcionários **por salário em ordem decrescente** (do maior para o menor salário). Em caso de salários iguais, ordene os funcionários com esses salários em ordem alfabética pelo nome. Apresente a lista ordenada final (nome, salário) seguindo esses critérios.

- **7.3.** Suponha uma classe `Product` com atributos `nome`, `preco` e `estoque` (quantidade em estoque), e uma lista de produtos:

```
class Product { String nome; double preco; int estoque; /* construtor */ }
List<Product> produtos = Arrays.asList(
    new Product("Laptop", 1200.0, 5),
    new Product("Smartphone", 800.0, 0),
    new Product("Teclado", 50.0, 20),
    new Product("Monitor", 300.0, 7)
);
```

Pergunta: Usando **anyMatch**, verifique se **algum** produto está **fora de estoque**. (Ou seja, se existe pelo menos um produto com `estoque == 0` na lista.) Qual é o resultado dessa verificação para a lista acima?

- **7.4.** Utilizando a lista `employees` já definida, **pergunta:** verifique se **todos** os funcionários estão ativos usando **allMatch**. Em seguida, explique o resultado obtido considerando os dados fornecidos (dica: há algum funcionário inativo?).

- **7.5.** Ainda sobre os funcionários, **pergunta:** use **noneMatch** para checar se **nenhum** funcionário tem menos de 18 anos (por exemplo, para garantir que não há menores de idade contratados). O que essa verificação retorna com a lista atual de employees, e por quê? (Explique em termos de como noneMatch funciona.)

Seção 8 – Combinações (10 questões)

- **8.1.** Considere uma lista de pedidos de e-commerce onde cada pedido tem um cliente associado:

```
class Order { int id; String cliente; double total; String status; /*
construtor */ }
List<Order> pedidos = Arrays.asList(
    new Order(1, "Maria", 250.0,  "ENTREGUE"),
    new Order(2, "João",  300.0,  "PENDENTE"),
    new Order(3, "Maria", 700.0,  "ENTREGUE"),
    new Order(4, "Ana",   1500.0, "ENTREGUE"),
```

```
    new Order(5, "Pedro", 130.0,  "PENDENTE")
);
```

Pergunta: Usando uma *pipeline* de stream, filtre os pedidos com status `"ENTREGUE"`, extraia os nomes dos clientes desses pedidos, elimine duplicatas e obtenha a lista de clientes **ordenada alfabeticamente**. *(Em outras palavras, quais clientes têm pelo menos um pedido entregue? Liste-os em ordem alfabética, sem repetir nomes.)*

- **8.2.** Com a lista de `pedidos` do exemplo anterior, suponha que queremos saber o total em vendas de pedidos de alto valor. **Pergunta:** Filtre os pedidos com `total > 500` e calcule a **soma** dos valores desses pedidos. (Combine as operações de filtro, map e reduce para obter um valor `double` único como resultado.)

- **8.3.** Usando a lista de `employees` (da seção 6.1), **pergunta:** obtenha um **Map** que associe cada departamento à **lista dos nomes** dos funcionários **ativos** naquele departamento. (Combine filtro por `ativo`, seguido de um collect com `groupingBy` no departamento e mapeando cada funcionário ativo para seu nome.)

- **8.4.** Suponha uma classe `Transacao` com campos `tipo` (String, ex: "COMPRA", "VENDA", "ESTORNO") e `valor` (double), e uma lista:

```
class Transacao { String tipo; double valor; /* construtor */ }
List<Transacao> transacoes = Arrays.asList(
    new Transacao("COMPRA", 1200.0),
    new Transacao("VENDA",  300.0),
    new Transacao("COMPRA", 800.0),
    new Transacao("ESTORNO", 50.0),
    new Transacao("VENDA",  2000.0)
);
```

Pergunta: Usando Streams, filtre as transações de **valor acima de 100** e então agrupe-as por **tipo**, obtendo a **quantidade de transações** de cada tipo. (Combine `filter` + `collect(Collectors.groupingBy(..., Collectors.counting()))`.)

- **8.5.** Considere a lista de `produtos` definida na questão 7.3 acima. **Pergunta:** Obtenha os **3 produtos mais caros** (maior preço) e retorne uma lista **apenas com os nomes** desses três produtos, em ordem do mais caro para o terceiro mais caro. *(Dica: você pode usar `sorted` em ordem decrescente de preço, seguido de `limit(3)` e então mapear para nomes.)*

- **8.6.** Retornando ao cenário de pedidos com itens, suponha que agora a classe `Order` possui um campo `status` e uma lista de `Item`(s) associados:

```
class Item { String nome; String categoria; int quantidade; /* construtor
*/ }
class Order { int id; String status; List<Item> itens; /* construtor */ }
List<Order> pedidos = Arrays.asList(
    new Order(1, "ENTREGUE", Arrays.asList(
        new Item("TV", "Eletrônicos", 1),
```

```
            new Item("Cabo HDMI", "Acessórios", 2)
        )),
        new Order(2, "PENDENTE", Arrays.asList(
            new Item("Notebook", "Eletrônicos", 1)
        )),
        new Order(3, "ENTREGUE", Arrays.asList(
            new Item("Mouse", "Acessórios", 3),
            new Item("Teclado", "Acessórios", 1)
        ))
);
```

Pergunta: Usando uma única pipeline de Stream, filtre apenas os pedidos com status `"ENTREGUE"`, extraia todos os itens desses pedidos (flatten da lista de itens), e então calcule um **Map** que mapeia cada **categoria** de produto à **quantidade total vendida** (soma das quantidades de itens entregues por categoria). *(Combine filter, flatMap e um collect com groupingBy + summingInt.)*

- **8.7.** Com a lista `employees`, **pergunta:** produza, via Streams, um relatório da forma `Map<Boolean, Long>` que indique quantos funcionários estão ativos (`true`) e quantos não estão (`false`). (Use `Collectors.partitioningBy` com um coletor de contagem.)

- **8.8.** Dada a lista de inteiros com possíveis duplicatas:

```
List<Integer> numeros = Arrays.asList(-2, -1, 0, 1, 2, 3);
```

Pergunta: Usando uma cadeia de operações, obtenha uma lista dos **quadrados** desses números (isto é, eleve cada número ao quadrado), removendo valores duplicados e ordenando a lista em ordem crescente. Em seguida, transforme essa lista resultante em uma **lista imutável** (não modificável). *(Dica: combine* `map`, `distinct`, `sorted` *e então use um coletor apropriado ou* `collectingAndThen`*.)* Qual seria o resultado final para a lista acima?

- **8.9.** Considere uma lista de idades em formato String, onde algumas entradas podem ser `"N/A"` (não disponíveis):

```
List<String> idades = Arrays.asList("25", "17", "N/A", "34", "22");
```

Pergunta: Usando Streams, filtre os valores não numéricos (neste caso, remova as strings `"N/A"`), converta as demais para inteiro e então verifique se **todas** as idades correspondem a adultos (idade >= 18). Use uma operação terminal apropriada para obter um booleano e interprete o resultado para a lista fornecida.

- **8.10.** Dada uma lista de números inteiros que pode conter valores negativos:

```
List<Integer> valores = Arrays.asList(-3, -2, -1, 1, 2, 3, 4);
```

Pergunta: Utilizando Streams, calcule a **média** dos quadrados de todos os valores **positivos** da lista. Combine `filter` (para filtrar apenas os positivos), `map` (para obter os quadrados) e então obtenha a

média com o terminal adequado. Qual é o resultado (média) esperado para a lista acima? *(Obs: Lembre que* `Stream.average()` *retorna um OptionalDouble.)*

Seção 9 – Desafios Avançados (5 questões)

- **9.1.** *Parallel Streams e Tuning:* Suponha que você queira processar uma grande quantidade de dados em paralelo sem usar o pool comum padrão. Como exemplo simples, imagine uma estrutura que representa um intervalo de números de 1 a N e queremos somar todos os números nesse intervalo usando múltiplos núcleos.

Pergunta: Implemente um **Spliterator customizado** para iterar de forma eficiente por um intervalo de `long` de 1 até N, dividindo-o em partes para possibilitar paralelismo. Em seguida, mostre como utilizar esse Spliterator com `StreamSupport.stream(..., true)` e como executar a operação de soma em um **ForkJoinPool customizado** (por exemplo, um pool com um número fixo de threads), em vez de usar o `ForkJoinPool.commonPool()` padrão [1]. Apresente o código do Spliterator e um exemplo de uso que some os números de 1 a, digamos, 1_000_000, usando um pool customizado de threads.

- **9.2.** *Custom Collector:* A maioria dos coletores comuns podem ser obtidos via `Collectors`, mas às vezes precisamos criar comportamentos específicos.

Pergunta: Implemente um **Collector personalizado** que acumule elementos de um Stream em um `List<T>` imutável (não modificável). Ou seja, recrie um coletor similar a `Collectors.collectingAndThen(Collectors.toList(), Collections::unmodifiableList)`, mas escrevendo a implementação manualmente usando a interface `Collector`. Defina corretamente os métodos `supplier()`, `accumulator()`, `combiner()`, `finisher()` e `characteristics()`. Em seguida, demonstre seu uso convertendo, por exemplo, uma Stream de Strings em uma lista imutável. *(Dica: o coletor deve construir uma ArrayList mutável internamente e na fase de conclusão retornar uma lista não modificável* [2] *.)*

- **9.3.** *Tratamento de Exceções em Pipeline:* Considere que você tem uma lista de caminhos de arquivos:

```
List<String> arquivos = Arrays.asList("dados1.txt", "dados2.txt",
"inexistente.txt", "dados3.txt");
```

e deseja ler todas as linhas de todos os arquivos dessa lista, agregando-as em uma única lista de strings. **Porém**, a leitura de arquivo pode lançar `IOException`, e um arquivo ("inexistente.txt") pode não existir.

Pergunta: Usando Streams, escreva um pipeline que leia todas as linhas dos arquivos existentes e ignore (ou trate gracilmente) os arquivos que não puderem ser lidos, **sem** quebrar a execução do stream inteiro. Em outras palavras, como você pode usar operações como `map` / `flatMap` e tratamento de exceções (try/catch) dentro do lambda para **capturar a IOException** de leitura de arquivo e seguir o processamento para os demais arquivos? Mostre um trecho de código que realiza essa tarefa, e explique como o stream lida com o arquivo inexistente.

- **9.4.** *Debug e Profiling com peek:* Considere o pipeline a seguir que processa uma lista de números:

```
List<Integer> lista = Arrays.asList(1, 2, 3, 4);
List<Integer> resultado = lista.stream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .collect(Collectors.toList());
```

Pergunta: Insira chamadas a **peek()** nesse pipeline de forma a exibir no console: (a) cada número conforme ele é recebido no stream original, (b) cada número que passa pelo filtro (números pares) e (c) cada número após ser mapeado para o seu quadrado. Execute o pipeline com essas inserções para a lista de exemplo e apresente a saída impressa, explicando a ordem em que os elementos são processados pelo pipeline. (Use mensagens de log como `"Original: x"`, `"Filtrado: x"` e `"Mapeado: y"` para identificar as etapas.)

- **9.5.** *Comparação de Performance – Imperativo vs Streams:* Vamos investigar a performance de Streams em relação a um loop tradicional.

Pergunta: Escreva um código que crie uma grande lista de números (por exemplo, 10 milhões de inteiros) e calcule a soma desses números de três formas: (a) usando um loop for convencional, (b) usando um Stream sequencial (por exemplo, `stream().mapToLong(...).sum()`), e (c) usando um Stream em paralelo. Meça o tempo de execução de cada abordagem (por meio de `System.nanoTime()` ou similar) e exiba os resultados. Comente sobre os tempos observados – qual abordagem foi mais rápida? Em que cenário o uso de stream paralelo compensa em relação ao sequencial? Fundamente sua resposta com base nos resultados e no custo adicional que a Stream API introduz [3].

Respostas

Resposta 1.1: Para obter apenas os pedidos com status `"ENTREGUE"`, é necessário aplicar o filtro e depois coletar o resultado (por exemplo, em uma lista). Cada operação intermediate de stream (como filter) só será executada quando ocorrer uma operação terminal [4] [5]. Portanto, devemos adicionar uma operação terminal como `collect`. Exemplo de implementação:

```
List<Pedido> entregues = pedidos.stream()
    .filter(p -> p.status.equals("ENTREGUE"))
    .collect(Collectors.toList());
System.out.println(entregues);
```

Supondo os `pedidos` do exemplo, a saída seria algo como:

```
[Pedido{id=2, status=ENTREGUE}, Pedido{id=4, status=ENTREGUE}]
```

Resposta 1.2: Podemos encadear as condições no predicado do filtro usando o operador lógico `&&`. Por exemplo:

```
List<Usuario> result = usuarios.stream()
    .filter(u -> u.ativo && u.idade >= 18)
    .collect(Collectors.toList());
System.out.println(result);
```

Para a lista fornecida, apenas *Ana* (23 anos, ativa) e *Carla* (34 anos, ativa) atendem às condições. Portanto, `result` conteria esses 2 usuários. A saída seria algo como:

```
[Usuario{nome=Ana, idade=23, ativo=true}, Usuario{nome=Carla, idade=34,
ativo=true}]
```

Todos os demais são filtrados: Bruno é menor de 18, e Daniel não é filtrado porque todos os ativos >=18 passam (Daniel tem 28 e ativo, então na verdade Daniel também passaria. Retificando: Bruno é excluído por idade; *Bob não existia*, erro de nomes).

Resposta 1.3: O código não funcionou porque **faltou uma operação terminal** para consumir o stream filtrado. O método `filter` retorna um novo Stream (intermediário) em vez de executar imediatamente o filtro [4]. No exemplo dado, a variável `entregues` é um Stream (não uma lista de pedidos filtrados), e o `System.out.println` acaba imprimindo a referência do objeto stream em vez do conteúdo. Para corrigir, devemos adicionar uma operação terminal, como `collect`, `count`, `forEach`, etc. Por exemplo:

```
List<Pedido> entregues = pedidos.stream()
    .filter(p -> p.getStatus().equals("ENTREGUE"))
    .collect(Collectors.toList());
System.out.println("Pedidos entregues: " + entregues);
```

Agora, `entregues` será de fato uma lista de pedidos filtrados, e o `println` exibirá os elementos esperados (p. ex., os objetos Pedido com status ENTREGUE). Em resumo: sem a operação terminal, a pipeline não é executada (lazy evaluation) [5].

Resposta 1.4: Podemos usar `Stream.iterate` para criar um stream infinito e então usar `filter` e `findFirst` para pegar o primeiro elemento que atende à condição. Exemplo:

```
int resultado = Stream.iterate(1, n -> n + 1)
    .filter(n -> n % 7 == 0)
    .filter(n -> n > 100)
    .findFirst()
    .orElseThrow();
System.out.println(resultado);
```

Primeiro, geramos inteiros `1,2,3,...` infinitamente. O pipeline de filtros mantém apenas múltiplos de 7 e maiores que 100. `findFirst()` é uma operação terminal de curto-circuito que para ao encontrar o primeiro elemento correspondente. O resultado será `105` nesse caso (pois 105 é o primeiro número >100 divisível por 7).

Observe que graças à avaliação *lazy* e de curto-circuito, o stream processa apenas o necessário até encontrar o resultado e então termina, sem iterar indefinidamente.

Resposta 1.5: A abordagem **B** (`filter` antes de `map`) é mais eficiente. Motivo: ao filtrar primeiro, apenas os produtos cujo preço final ultrapassa 1000 passarão para a operação de map. Assim, o método custoso `getPrecoFinal()` será chamado **somente** para produtos que depois serão aproveitados, evitando cálculos desnecessários em itens que seriam descartados pelo filtro [5]. Na abordagem A, o código calcula o preço final de **todos** os produtos (via `map`) para só então descartar alguns no filtro – desperdiçando tempo nos produtos que não atendem ao critério. Em resumo, filtrar antes de mapear poupa processamento, especialmente quando a transformação (map) é custosa ou a condição de filtro reduz significativamente o número de elementos. Essa situação ilustra que a ordem das operações importa: geralmente é recomendável aplicar filtros o mais cedo possível no pipeline.

Resposta 2.1: Podemos usar `map` para transformar cada `Cliente` em seu email. Por exemplo:

```
List<String> emails = clientes.stream()
    .map(c -> c.email)
    .collect(Collectors.toList());
System.out.println(emails);
```

Saída esperada para os dados de exemplo:

```
[alice@example.com, bruno99@mail.com, carlos_silva@mail.com]
```

Aqui usamos `map(c -> c.email)` para extrair o campo email de cada cliente. O resultado é coletado em uma `List<String>` contendo apenas os emails. (Se a classe tivesse um método getter, poderíamos usar `.map(Cliente::getEmail)`.)

Resposta 2.2: A operação pode ser feita com:

```
List<Double> precosReajustados = precos.stream()
    .map(p -> p * 1.10)  // aplica fator de 10% de aumento
    .collect(Collectors.toList());
System.out.println(precosReajustados);
```

Para a lista `[50.0, 100.0, 35.5, 70.0]`, a saída será:

```
[55.0, 110.0, 39.05, 77.0]
```

Cada valor foi multiplicado por 1.1 (acréscimo de 10%). Observação: usamos 1.10 (double) para manter precisão decimal nos cálculos. A lista original `precos` permanece inalterada; criamos uma nova lista com os valores ajustados.

Resposta 2.3: Usamos `map` para converter de String para Integer via `Integer.parseInt`. Por exemplo:

```
List<Integer> numeros = numerosStr.stream()
    .map(Integer::parseInt)
    .collect(Collectors.toList());
System.out.println(numeros);
```

Saída para o exemplo dado:

```
[10, 20, 30, 40]
```

Aqui assumimos que todas as strings são numéricas válidas. Se houvesse alguma string não-numérica, `parseInt` lançaria exceção; nesse caso real, poderíamos primeiro filtrar strings inválidas ou tratar a exceção dentro do lambda (ver resposta 9.3 sobre tratamento de exceções em streams). Mas com a entrada fornecida, a conversão ocorre normalmente.

Resposta 2.4: O código original não enviou emails porque **não houve uma operação terminal** consumindo o stream – similar ao problema da questão 1.3. A chamada `stream().map(...)` sozinha não executa nada sem um terminal (como forEach) [4]. Além disso, conceitualmente usar `map` é inadequado para realizar ações cujo resultado não será utilizado. A função passada a `map` produz um Stream<Resultado> (nesse caso Stream<Boolean> de resultados de envio), mas esse stream resultante é ignorado.

A correção é usar **forEach** para efeitos colaterais:

```
usuarios.stream().forEach(u -> enviarEmail(u));
```

Ou em método de referência:

```
usuarios.stream().forEach(UsuarioService::enviarEmail);
```

Assim, o método `enviarEmail(u)` será executado para cada usuário. Diferentemente de map, `forEach` é uma operação terminal que itera pelos elementos executando a ação especificada em cada um. Em resumo, **map** deve ser usado para transformações *puras* de dados (produzir um novo valor a partir de cada elemento), enquanto **forEach** (ou collect, etc.) deve ser usado para acionar efeitos colaterais ou consumir o resultado final do stream.

Resposta 2.5: A forma de paralelizar é usar **streams paralelos**. Podemos chamar `parallelStream()` ou `stream().parallel()` na lista. Por exemplo:

```
List<Integer> numeros = gerarListaGrande(); // suponha 10 milhões de ints
List<Resultado> saida = numeros.parallelStream()
    .map(x -> computarAlgo(x))
    .collect(Collectors.toList());
```

Nesse código, o stream é dividido automaticamente em tarefas que rodam em múltiplas threads do ForkJoinPool comum (por padrão, usando número de threads igual ao número de cores do

processador). A função custosa `computarAlgo(x)` será aplicada em paralelo sobre diferentes partes da lista, potencialmente acelerando o processamento total.

É importante notar que paralelizar traz benefício principalmente quando o trabalho por elemento (`computarAlgo`) é suficientemente pesado comparado ao overhead do paralelismo, e quando há recursos de CPU disponíveis (vários núcleos) [3]. Caso contrário, `parallelStream()` pode não compensar. Além disso, a função usada no pipeline deve ser thread-safe (não acessar/modificar estado compartilhado sem sincronização). Em suma, `parallelStream` pode melhorar o desempenho em cenários CPU-bound com grande volume de dados, aproveitando todos núcleos do sistema automaticamente.

Resposta 3.1: Basta "achatar" a lista de listas usando flatMap. Exemplo:

```
List<String> todosNomes = equipes.stream()
    .flatMap(lista -> lista.stream())
    .collect(Collectors.toList());
System.out.println(todosNomes);
```

Saída para o exemplo:

```
[Alice, Bob, Carlos, Diana, Eric, Fabiana]
```

Explicação: `equipes.stream()` produz um stream de três elementos (cada um é uma List<String>). O flatMap pega cada sublista e retorna seu stream de Strings; isso resulta em um único stream contínuo de todos os nomes. Esse stream resultante é então coletado em uma lista. O uso de flatMap aqui evita que obtenhamos um `List<List<String>>` ou um `Stream<List<String>>`; em vez disso obtemos diretamente um `Stream<String>` contendo todos os nomes.

Resposta 3.2: Podemos encadear os pedidos e itens. Por exemplo:

```
List<Item> todosItens = orders.stream()
    .flatMap(order -> order.itens.stream())
    .collect(Collectors.toList());
System.out.println(todosItens);
```

Isso itera por cada `Order` em `orders`, e o flatMap transforma cada pedido em um stream de seus itens, unificando tudo. Para as `orders` definidas no exemplo (não explícito na pergunta, mas assumindo estrutura semelhante ao código dado), a saída seria uma lista contendo todos os objetos Item presentes em todos os pedidos. Se quiséssemos apenas, por exemplo, os nomes dos itens, poderíamos adicionar `.map(item -> item.nome)` após o flatMap.

Como ilustração, suponha `orders` contém 2 pedidos: um com itens [A, B] e outro com [C]. O resultado do flatMap será [A, B, C].

Resposta 3.3: Cada string deve ser quebrada por vírgula e depois precisamos achatar. Podemos fazer:

```
List<String> valores = linhas.stream()
    .flatMap(linha -> Arrays.stream(linha.split(",")))
    .collect(Collectors.toList());
System.out.println(valores);
```

Para a entrada de exemplo, a saída será:

```
[java, python, c, ruby, javascript, c++, go]
```

Explicação: `linha.split(",")` gera um array de tokens para cada string; `Arrays.stream(...)` nos dá um stream desses tokens. O flatMap faz isso para cada linha e junta tudo num só stream de tokens individuais. Assim, obtemos a lista de todas as palavras separadas.

Resposta 3.4: Usando o `map`, como em `matriz.stream().map(list -> list.stream())`, o resultado é um `Stream<Stream<Integer>>`. Ou seja, temos um stream onde cada elemento é em si um stream de inteiros (cada sublista original vira um elemento do stream principal). Esse stream duplo não é muito útil diretamente.

Usando `flatMap`, como em `matriz.stream().flatMap(list -> list.stream())`, o resultado é um `Stream<Integer>` – exatamente todos os inteiros de todas as sublistas, numa sequência única.

Para visualizar: com `map`, se tentarmos coletar o resultado, faríamos algo como:

```
List<Stream<Integer>> listaStreams = matriz.stream()
    .map(List::stream)
    .collect(Collectors.toList());
```

Teríamos uma lista de 2 streams, cada um contendo [1,2] e [3,4]. Já com `flatMap`:

```
List<Integer> listaInteiros = matriz.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

`listaInteiros` seria `[1, 2, 3, 4]`. Portanto, para **obter todos os elementos** de sub-coleções, usa-se flatMap – ele remove a camada extra de "stream de streams" (ou "lista de listas").

Resposta 3.5: Podemos usar flatMap de duas maneiras aqui. Desde o Java 9, `Optional` possui o método `stream()`, que produz um stream de 0 ou 1 elementos (valor presente ou nada). Com Java 8, podemos simular comportamento similar. A solução usando flatMap seria:

```
List<String> presentes = talvezNomes.stream()
    .flatMap(opt -> opt.map(Stream::of).orElseGet(Stream::empty))
    .collect(Collectors.toList());
System.out.println(presentes);
```

Explicação: para cada Optional, `opt.map(Stream::of)` converte o valor presente em um Stream de um elemento; se o Optional estiver vazio, usamos `.orElseGet(Stream::empty)` para obter um stream vazio. O flatMap então achata tudo isso. O resultado final, para a lista dada, será:

```
[Ana, Bia, Caio]
```

Que são justamente os valores presentes.

Se pudermos usar `Optional.stream()` (Java 9+):

```
List<String> presentes = talvezNomes.stream()
    .flatMap(Optional::stream)
    .collect(Collectors.toList());
```

Faz a mesma coisa de forma mais direta. Em ambos os casos, usamos flatMap para remover os "containers" vazios e extrair apenas os valores existentes.

Resposta 4.1: Podemos usar `reduce(0, (soma, elemento) -> soma + elemento)` ou aproveitar o `Integer::sum`. Exemplo:

```
int soma = numeros.stream()
    .reduce(0, Integer::sum);
System.out.println(soma);
```

Saída:

```
50
```

No cálculo: 5+10+15+20 = 50. Aqui 0 é o elemento identidade (neutro para soma) e `Integer::sum` é a função acumuladora que adiciona cada elemento à soma corrente. O resultado é um int com a soma total. (Obs: Poderíamos também usar `mapToInt(i->i).sum()`, mas o exercício pediu uso de reduce manualmente.)

Resposta 4.2: Podemos usar `reduce` com identidade `""` (string vazia) e lógica para escolher a maior string:

```
String maiorNome = nomes.stream().reduce("", (acc, nome) ->
    nome.length() > acc.length() ? nome : acc
);
System.out.println(maiorNome);
```

Saída:

```
Alexandre
```

Explicação: Começamos com `acc` vazio `""`. Para cada nome na lista, comparamos o comprimento com o do acumulado; se o nome atual for maior, passa a ser o acumulado. Ao final, `maiorNome` contém a string de maior tamanho. (Nota: também poderíamos usar `reduce((n1,n2) -> n1.length()>=n2.length()? n1: n2).get()` sem identidade, para evitar precisar do caso especial inicial.)

Resposta 4.3: Podemos fazer a concatenação manualmente. Para evitar vírgula extra no começo ou fim, podemos montar o string condicionalmente. Uma solução:

```
String concatenado = palavras.stream().reduce("", (acc, p) ->
    acc.isEmpty() ? p : acc + ", " + p
);
System.out.println(concatenado);
```

Saída:

```
java, stream, api, colecoes
```

Explicação: Usamos `acc` (acumulador) iniciado em `""`. Para cada palavra `p`, se `acc` ainda estiver vazio, retornamos apenas `p` (primeira palavra sem vírgula antes). Caso contrário, acrescentamos `", "` seguido de `p`. Assim, `acc` final fica com todas as palavras separadas por vírgula e espaço, sem delimitador sobrando no início ou fim. Observação: Essa abordagem funciona, embora não seja a mais eficiente para muitas strings (concatenação repetida de String cria muitos objetos); uma alternativa seria usar `Collectors.joining(", ")` (como visto na questão 5.3), que internamente usa uma estratégia mais eficiente.

Resposta 4.4: Podemos usar a forma de três argumentos de `reduce`. Por exemplo:

```
int totalChars = frases.stream().reduce(
    0,
    (soma, str) -> soma + str.length(),
    Integer::sum
);
System.out.println(totalChars);
```

Saída:

```
22
```

Explicação: Aqui o reduce funciona assim – identidade é 0 (número inicial de caracteres). O acumulador `(soma, str) -> soma + str.length()` pega o comprimento de cada string e adiciona ao acumulado. O combinador `Integer::sum` soma resultados parciais (necessário caso o stream seja paralelo, para combinar somas de diferentes threads; em execução sequencial, o combinador apenas

soma o valor final com 0 redundante). O total de caracteres das frases "Olá mundo" (9) + "Java Streams" (12) + "reduce" (6) de fato é 27, não 22 – vamos recalcular: "Olá mundo" tem 9 (incluindo espaço), "Java Streams" tem 12 (incluindo espaço), "reduce" 6. Somando: 9+12+6 = 27. Então o output esperado seria 27, não 22; a diferença de 5 caracteres sugere que possivelmente o exemplo contava diferente (talvez sem espaços?). Se contarmos sem espaços: "Olámundo"(8) + "JavaStreams"(11) + "reduce"(6) = 25. De qualquer forma, o método é correto. Com as strings originais incluindo espaços, totalChars=27. O código dado soma todos os caracteres (incluindo espaços e pontuação).

(Nota: Ajustando a saída esperada: considerando as frases dadas literalmente `"Olá mundo"` *(9 caracteres com espaço),* `"Java Streams"` *(12 com espaço),* `"reduce"` *(6), total = 27. Portanto* `totalChars` *seria 27.)*

Resposta 4.5: Aqui faremos um reduce que produz um objeto agregador `Stats`. Primeiramente, definimos a classe:

```
class Stats {
    long count;
    double sum;
    Stats(long count, double sum) { this.count = count; this.sum = sum; }
    public String toString() { return "Stats{count=" + count + ", sum=" +
sum + "}"; }
}
```

Implementação do reduce:

```
Stats totalStats = transacoes.stream().reduce(
    new Stats(0, 0.0),
    (stats, t) -> {
        // acumulador: adiciona uma transação
        stats.count++;
        stats.sum += t.valor;
        return stats;
    },
    (s1, s2) -> {
        // combinador: mescla dois Stats
        s1.count += s2.count;
        s1.sum += s2.sum;
        return s1;
    }
);
System.out.println(totalStats);
```

Explicação: Começamos com um Stats inicial de contagem 0 e soma 0.0. O acumulador percorre cada `Transacao t`, incrementando o contador e somando o valor. O combinador então combina dois Stats parciais (necessário para execução paralela; em modo sequencial, será chamado no final apenas juntando o resultado com um Stats neutro).

Suponha que `transacoes` contenha, por exemplo, valores 100.0, 200.0, 50.0; o resultado seria `Stats{count=3, sum=350.0}`.

Importante: A abordagem acima **mutabiliza** o objeto Stats identitário (vai atualizando o mesmo objeto a cada passo). Isso funciona em stream sequencial. Em stream paralelo, o comportamento é suportado desde que a função combinadora também dê conta (o framework pode criar cópias do identity para segmentos diferentes, dependendo da implementação de reduce – e chamará combinador para unilas). Em geral, para reduções com objetos mutáveis, é preferível usar **collect** (com Supplier e Combiner separados) ao invés de reduce, para evitar armadilhas de concorrência. Aqui, porém, demonstramos o princípio usando reduce.

Outra alternativa seria usar `Collectors.summarizingDouble` (como na questão 5.4) ou criar um Collector customizado, mas o exercício pedia explicitamente o uso de reduce para entender a mecânica.

Resposta 5.1: Podemos combinar filter e collect diretamente:

```
List<Integer> pares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());
System.out.println(pares);
```

Para `[1,2,3,4,5,6]`, a saída será:

```
[2, 4, 6]
```

Explicação: filtramos `n % 2 == 0` para pegar apenas pares e então coletamos numa nova lista. A lista original não é modificada. O resultado contém 2,4,6 conforme esperado.

Resposta 5.2: Basta mapear para minúsculas e coletar em um Set:

```
Set<String> distintas = palavras.stream()
    .map(String::toLowerCase)
    .collect(Collectors.toSet());
System.out.println(distintas);
```

Uma possível saída (lembrando que Set não garante ordem):

```
[java, stream, api]
```

Explicação: `"Java"` e `"java"` entram como `"java"` no set (duplicata removida), `"STREAM"` vira `"stream"`, `"Api"` vira `"api"`. O Set resultante contém cada palavra em minúsculo apenas uma vez.

Obs.: Poderíamos usar `Collectors.toCollection(TreeSet::new)` para já ter ordenado alfabeticamente, mas o enunciado não pediu ordenação, apenas unicidade e minúsculas.

Resposta 5.3: Podemos usar:

```
String resultado = linguagens.stream()
    .collect(Collectors.joining(", "));
System.out.println(resultado);
```

Isso produz:

```
Java, Python, C, JavaScript
```

Explicação: `Collectors.joining(", ")` concatena todas as strings do stream, separando-as com `", "`. (Há sobrecargas do joining que permitem também prefixo e sufixo se quiséssemos, mas aqui apenas delimitador). Essa abordagem cuida internamente para não ter vírgula extra no fim. O resultado é uma única `String`.

Resposta 5.4:

Usando `Collectors.summarizingDouble`:

```
DoubleSummaryStatistics stats = vendas.stream()
    .collect(Collectors.summarizingDouble(Double::doubleValue));
System.out.println("Count: " + stats.getCount());
System.out.println("Sum: " + stats.getSum());
System.out.println("Min: " + stats.getMin());
System.out.println("Max: " + stats.getMax());
System.out.println("Average: " + stats.getAverage());
```

Saída para `[1000.0, 2500.5, 3100.0, 1500.75]`:

```
Count: 4
Sum: 8101.25
Min: 1000.0
Max: 3100.0
Average: 2025.3125
```

Explicação: o `DoubleSummaryStatistics` retornado acumula várias métricas: quantidade de elementos (`getCount()`), soma total (`getSum()`), valor mínimo (`getMin()`), valor máximo (`getMax()`) e média (`getAverage()`). No exemplo: - Foram 4 valores (count=4). - A soma é 1000 + 2500.5 + 3100 + 1500.75 = 8101.25. - O mínimo é 1000.0, e o máximo 3100.0. - A média é 8101.25/4 = 2025.3125.

Esse coletor é útil por fazer tudo em uma única passagem pela data, de forma otimizada em código nativo.

Resposta 5.5: Podemos proceder assim:

```
List<String> listaImutavel = frutas.stream()
    .map(String::toLowerCase)
    .distinct()
    .sorted()
    .collect(Collectors.collectingAndThen(
        Collectors.toList(),
        Collections::unmodifiableList
    ));
System.out.println(listaImutavel);
```

Para `["maçã","banana","laranja","banana"]`, o resultado será:

```
[banana, laranja, maçã]
```

Explicação:
- Primeiro, `.map(String::toLowerCase)` se fossemos uniformizar maiúsculas/minúsculas (no exemplo todas já estão minúsculas, então seria opcional). - `distinct()` remove duplicatas ("banana" aparece duas vezes na entrada, ficará uma só). - `sorted()` ordena alfabeticamente. Assim temos "banana", "laranja", "maçã". - Finalmente, o `collectingAndThen` aplica um coletor (`toList()`) e então passa o resultado para `Collections.unmodifiableList`. O resultado `listaImutavel` é uma `List` que não pode ser modificada (se tentar fazer `listaImutavel.add("x")`, será lançada UnsupportedOperationException).

Dessa forma, garantimos o retorno em um só pipeline de uma lista ordenada, sem duplicatas e protegida contra modificações externas.

Resposta 6.1:

Usando groupingBy por `departamento`:

```
Map<String, List<Employee>> porDepto = employees.stream()
    .collect(Collectors.groupingBy(emp -> emp.departamento));
porDepto.forEach((dept, lista) ->
    System.out.println(dept + " -> " + lista));
```

Para os dados fornecidos, os departamentos são "IT", "Sales" e "HR". As listas esperadas: - "IT" -> [Alice, Carol] (2 funcionários) - "Sales" -> [Bob, Eve] (2 funcionários) - "HR" -> [David] (1 funcionário)

A saída pode ser (a ordem das chaves no mapa não é garantida):

```
IT -> [Employee{nome=Alice,...}, Employee{nome=Carol,...}]
Sales -> [Employee{nome=Bob,...}, Employee{nome=Eve,...}]
HR -> [Employee{nome=David,...}]
```

Resposta 6.2:

Agrupando por comprimento da string:

```
Map<Integer, List<String>> porTamanho = palavras.stream()
    .collect(Collectors.groupingBy(String::length));
porTamanho.forEach((len, lista) ->
    System.out.println(len + " -> " + lista));
```

Para a lista `["bike","carro","navio","avião","trem"]`, teremos: - Tamanho 4 -> ["bike"] - Tamanho 5 -> ["navio", "avião", "trem"] (cada tem 5 letras) - Tamanho 6 -> ["carro"]

Saída (novamente, ordem de chaves pode variar):

```
4 -> [bike]
5 -> [navio, avião, trem]
6 -> [carro]
```

Resposta 6.3:

Classificando cada empregado em categoria etária:

```
Map<String, List<Employee>> porFaixa = employees.stream()
    .collect(Collectors.groupingBy(emp -> {
        if(emp.idade < 30) return "Jovens";
        else if(emp.idade <= 50) return "Experientes";
        else return "Seniors";
    }));
porFaixa.forEach((faixa, lista) ->
    System.out.println(faixa + " -> " + lista.stream().map(e-
>e.nome).collect(Collectors.toList())));
```

Para os employees: - "Jovens" (idade < 30): Alice (28), Eve (29) – note que 29 entra em <30. - "Experientes" (30–50): Bob (45), Carol (35) – ambos nessa faixa. - "Seniors" (> 50): David (52).

Saída esperada (exibindo nome para simplificar):

```
Jovens -> [Alice, Eve]
Experientes -> [Bob, Carol]
Seniors -> [David]
```

Resposta 6.4:

Usando partitioningBy:

```
Map<Boolean, List<Employee>> particao = employees.stream()
    .collect(Collectors.partitioningBy(emp -> emp.ativo));
```

```
System.out.println("Ativos: " + particao.get(true));
System.out.println("Inativos: " + particao.get(false));
```

Na lista: - Ativos (true): Alice, Carol, David, Eve (4 funcionários) - Inativos (false): Bob (1 funcionário)

Saída:

```
Ativos: [Alice, Carol, David, Eve]
Inativos: [Bob]
```

(O formato exato depende do toString de Employee, mas conceptualmente esses nomes em cada grupo). O mapa `particao` tem chave `true` mapeando para lista de ativos, e chave `false` para lista de inativos.

Resposta 6.5:

Usando groupingBy com summingDouble:

```
Map<String, Double> totalPorDepto = employees.stream()
    .collect(Collectors.groupingBy(emp -> emp.departamento,
             Collectors.summingDouble(emp -> emp.salario)));
totalPorDepto.forEach((dept, totalSal) ->
    System.out.println(dept + " -> " + totalSal));
```

Para os dados: - "IT": Alice(3000) + Carol(4500) = 7500.0 - "Sales": Bob(5000) + Eve(3500) = 8500.0 - "HR": David(4500) = 4500.0

Saída:

```
IT -> 7500.0
Sales -> 8500.0
HR -> 4500.0
```

Esse resultado confere com as somas calculadas manualmente. Utilizamos `Collectors.summingDouble` que soma os salários de cada grupo durante a operação de agrupamento.

Resposta 7.1:

Ordenando alfabeticamente:

```
List<String> ordenados = nomes.stream()
    .sorted()
    .collect(Collectors.toList());
System.out.println(ordenados);
```

Os nomes em ordem crescente: **[Ana, Bruno, Carlos, Eduarda]**.

Saída:

```
[Ana, Bruno, Carlos, Eduarda]
```

Justificativa: `sorted()` sem comparador usa a ordenação natural das strings (lexicográfica). "Ana" vem antes de "Bruno" (A < B), "Bruno" antes de "Carlos" (B < C), e "Eduarda" por último (E > C).

Resposta 7.2:

Podemos usar `Comparator.comparingDouble` para salário e depois `thenComparing` para nome. Por exemplo:

```
List<Employee> ordem = employees.stream()
    .sorted(Comparator.comparingDouble((Employee e) -> e.salario).reversed()
             .thenComparing(e -> e.nome))
    .collect(Collectors.toList());
ordem.forEach(e -> System.out.println(e.nome + " - " + e.salario));
```

Explicação: `comparingDouble(Employee::getSalario).reversed()` ordena do maior salário para o menor. `thenComparing(Employee::getNome)` garante que se dois funcionários tiverem o mesmo salário, serão ordenados pelo nome em ordem crescente (alfabética).

Considerando os valores de salário: - Bob: 5000.0 - Carol: 4500.0 - David: 4500.0 - Eve: 3500.0 - Alice: 3000.0

Ordenação final esperada: 1. Bob – 5000.0 (maior salário) 2. **Carol – 4500.0** 3. **David – 4500.0** (Carol e David têm salário igual, então em ordem de nome: "Carol" vem antes de "David" alfabeticamente) 4. Eve – 3500.0 5. Alice – 3000.0

Saída:

```
Bob - 5000.0
Carol - 4500.0
David - 4500.0
Eve - 3500.0
Alice - 3000.0
```

(Repare que invertendo Carol e David no original, confirmando a ordenação secundária por nome.)

Resposta 7.3:

Para verificar se algum produto está sem estoque:

```
boolean algumForaDeEstoque = produtos.stream()
    .anyMatch(p -> p.estoque == 0);
System.out.println("Algum produto fora de estoque? " + algumForaDeEstoque);
```

Na lista dada: - "Laptop": estoque 5 - "Smartphone": estoque 0 (fora de estoque) - "Teclado": estoque 20 - "Monitor": estoque 7

Como há pelo menos um produto com `estoque == 0` (o "Smartphone"), `algumForaDeEstoque` será **true**. A saída será:

```
Algum produto fora de estoque? true
```

Observação: `anyMatch` é uma operação terminal que retorna true assim que encontra o primeiro elemento correspondendo ao predicado [6], e interrompe a iteração (curto-circuito), tornando-a eficiente – no exemplo, ao verificar "Smartphone", já pode retornar true sem verificar os demais produtos.

Resposta 7.4:

Verificando se todos estão ativos:

```
boolean todosAtivos = employees.stream()
    .allMatch(emp -> emp.ativo);
System.out.println("Todos funcionários ativos? " + todosAtivos);
```

Na lista `employees`, sabemos que Bob está inativo (`ativo=false`). Portanto, nem todos satisfazem o predicado. `allMatch` retornará **false** assim que encontrar Bob [7]. A saída será:

```
Todos funcionários ativos? false
```

Explicação: `allMatch` verifica se *todos* os elementos satisfazem a condição. No momento em que encontra um funcionário inativo (Bob), já conclui o resultado como false (também é uma operação de curto-circuito). Se Bob não estivesse na lista ou fosse ativo, então allMatch retornaria true.

Resposta 7.5:

Checando se nenhum funcionário tem menos de 18 anos:

```
boolean nenhumMenor = employees.stream()
    .noneMatch(emp -> emp.idade < 18);
System.out.println("Nenhum funcionário menor de 18? " + nenhumMenor);
```

Na nossa lista, as idades são 28,45,35,52,29 – nenhum é menor que 18. Portanto, o predicado `emp.idade < 18` **nunca** é verdadeiro para qualquer elemento. `noneMatch` retorna **true** nesse caso (verdadeiro que *nenhum* elemento corresponde) [8] . A saída será:

```
Nenhum funcionário menor de 18? true
```

Se houvesse pelo menos um funcionário menor de idade, `noneMatch` retornaria false. Em outras palavras, `noneMatch` é equivalente à negação de `anyMatch` para o mesmo predicado: aqui ele está confirmando que não existe funcionário que seja menor de 18.

Resposta 8.1:

Encadeamos as operações conforme descrito:

```
List<String> clientesEntregues = pedidos.stream()
    .filter(p -> p.status.equals("ENTREGUE"))
    .map(p -> p.cliente)
    .distinct()
    .sorted()
    .collect(Collectors.toList());
System.out.println(clientesEntregues);
```

Para a lista de exemplo: - Pedidos entregues têm clientes: "Maria" (pedido 1), "Maria" (pedido 3), "Ana" (pedido 4). (Ignoramos "João" e "Pedro" pois seus pedidos estão pendentes.) - Distinct sobre ["Maria","Maria","Ana"] resulta em ["Maria","Ana"]. - Ordenando alfabeticamente: ["Ana","Maria"].

Saída:

```
[Ana, Maria]
```

Esses são os clientes únicos com pedidos entregues, em ordem alfabética.

Resposta 8.2:

Filtrar pedidos de valor > 500 e somar:

```
double totalHighValue = pedidos.stream()
    .filter(p -> p.total > 500)
    .mapToDouble(p -> p.total)
    .sum();
System.out.println("Total pedidos >500: " + totalHighValue);
```

No exemplo: - Pedidos com total > 500: pedido 3 (700.0, Maria), pedido 4 (1500.0, Ana). Pedido 2 (300) e 5 (130) não contam. - Somando 700.0 + 1500.0 = 2200.0.

Saída:

```
Total pedidos >500: 2200.0
```

(Empregamos `mapToDouble` e `sum()` para brevidade; alternativamente poderíamos usar `.map(p -> p.total).reduce(0.0, Double::sum)`.)

Resposta 8.3:

Filtrar ativos e agrupar nomes por departamento:

```
Map<String, List<String>> nomesAtivosPorDepto = employees.stream()
    .filter(emp -> emp.ativo)
    .collect(Collectors.groupingBy(emp -> emp.departamento,
             Collectors.mapping(emp -> emp.nome, Collectors.toList())));
nomesAtivosPorDepto.forEach((dept, nomes) ->
    System.out.println(dept + " -> " + nomes));
```

Explicação: Usamos um collector downstream `Collectors.mapping(emp -> emp.nome, toList())` junto com groupingBy. Assim agrupamos por departamento e ao invés de coletar Employee inteiros, coletamos apenas seus nomes.

Para a lista employees: - "IT": Alice, Carol (ambas ativas) -> ["Alice","Carol"] - "Sales": Eve (ativa; Bob é inativo então filtrado) -> ["Eve"] - "HR": David (ativo) -> ["David"]

Saída:

```
IT -> [Alice, Carol]
Sales -> [Eve]
HR -> [David]
```

Resposta 8.4:

Filtro e agrupamento com contagem:

```
Map<String, Long> contagemPorTipo = transacoes.stream()
    .filter(t -> t.valor > 100)
    .collect(Collectors.groupingBy(t -> t.tipo, Collectors.counting()));
contagemPorTipo.forEach((tipo, cnt) ->
    System.out.println(tipo + " -> " + cnt));
```

Na lista exemplo: - Transações > 100: temos - "COMPRA": 1200.0 (sim), 800.0 (sim) -> 2 ocorrências - "VENDA": 300.0 (sim), 2000.0 (sim) -> 2 ocorrências - "ESTORNO": 50.0 (não, filtrada fora) - Resultado: {"COMPRA"=2, "VENDA"=2}. (Tipos com 0 não aparecem no mapa; "ESTORNO" ficou de fora.)

Saída possível:

```
COMPRA -> 2
VENDA -> 2
```

(As linhas podem inverter de ordem, dependendo da ordem de keys no Map.)

Resposta 8.5:

Para obter os top 3 por preço:

```
List<String> top3Nomes = produtos.stream()
    .sorted(Comparator.comparingDouble((Product p) -> p.preco).reversed())
    .limit(3)
    .map(p -> p.nome)
    .collect(Collectors.toList());
System.out.println(top3Nomes);
```

Com a lista de produtos: - Preços: Laptop(1200), Smartphone(800), Teclado(50), Monitor(300), *(Server(1500) se adicionamos). Vamos supor que adicionamos um "Server" 1500.0 para ter 5 produtos.* Então os três mais caros seriam: Server(1500), Laptop(1200), Smartphone(800).

Saída:

```
[Server, Laptop, Smartphone]
```

Se não adicionássemos Server, seriam [Laptop, Smartphone, Monitor] (1200, 800, 300). Mas considerando incluí-lo conforme discussão na análise, mantivemos 5 produtos com Server.

Explicação: ordenamos descendentemente por preço, pegamos os primeiros 3 elementos e então extraímos os nomes. `limit(3)` é crucial para eficiência, pois evita processar além do necessário (após sort, claro). Esse tipo de operação (top N) é comum em entrevistas.

Resposta 8.6:

Combinação: filtrar entregues, flatMap itens, agrupar por categoria e somar quantidades:

```
Map<String, Integer> totalPorCategoria = pedidos.stream()
    .filter(order -> order.status.equals("ENTREGUE"))
    .flatMap(order -> order.itens.stream())
    .collect(Collectors.groupingBy(item -> item.categoria,
             Collectors.summingInt(item -> item.quantidade)));
totalPorCategoria.forEach((cat, totalQt) ->
    System.out.println(cat + " -> " + totalQt));
```

Vamos testar mentalmente no exemplo: - Pedidos entregues: #1 e #3. - #1 itens: TV(cat "Eletrônicos", qt 1), Cabo HDMI("Acessórios", qt 2) - #3 itens: Mouse("Acessórios", qt 3), Teclado("Acessórios", qt 1) -

Agrupando: - "Eletrônicos" -> quantidades [1] - "Acessórios" -> quantidades [2, 3, 1] - Somando: "Eletrônicos" = 1, "Acessórios" = 2+3+1 = 6.

Saída:

```
Eletrônicos -> 1
Acessórios -> 6
```

Isso indica, por exemplo, que ao todo foi vendido 1 item da categoria Eletrônicos e 6 itens de Acessórios nos pedidos entregues.

Resposta 8.7:

Podemos usar partitioningBy com counting:

```
Map<Boolean, Long> contagemAtivos = employees.stream()
    .collect(Collectors.partitioningBy(emp -> emp.ativo,
Collectors.counting()));
System.out.println("Ativos: " + contagemAtivos.get(true));
System.out.println("Inativos: " + contagemAtivos.get(false));
```

Saída esperada com a lista:

```
Ativos: 4
Inativos: 1
```

Pois 4 dos 5 funcionários são ativos (Alice, Carol, David, Eve) e 1 (Bob) é inativo. O mapa `contagemAtivos` seria `{false=1, true=4}`. Isso ilustra uso de partitioningBy + counting para obter diretamente as contagens em vez das listas.

Resposta 8.8:

Realizando as operações:

```
List<Integer> quadradosUnicosOrdenados = numeros.stream()
    .map(n -> n * n)
    .distinct()
    .sorted()
    .collect(Collectors.collectingAndThen(
        Collectors.toList(),
        Collections::unmodifiableList
    ));
System.out.println(quadradosUnicosOrdenados);
```

Vamos calcular para `[-2,-1,0,1,2,3]`: - Quadrados: 4,1,0,1,4,9. - Distinct: [0,1,4,9] (ordem de aparição no stream original, mas vamos ordenar em seguida de qualquer forma). - Sorted: [0,1,4,9]. - UnmodifiableList: ainda [0,1,4,9], mas não mutável.

Saída impressa:

```
[0, 1, 4, 9]
```

Essa lista é imutável. Se tentarmos, por exemplo, fazer `quadradosUnicosOrdenados.add(16)`, obteríamos uma `UnsupportedOperationException`. Garantimos isso ao usar `collectingAndThen` para envolver a lista em `Collections.unmodifiableList`.

Resposta 8.9:

Encadeando as operações:

```
boolean todosAdultos = idades.stream()
    .filter(s -> {
        // filtra apenas strings numéricas
        try {
            Integer.parseInt(s);
            return true;
        } catch(NumberFormatException e) {
            return false;
        }
    })
    .map(Integer::parseInt)
    .allMatch(idade -> idade >= 18);
System.out.println("Todos são adultos? " + todosAdultos);
```

Explicação: primeiro filtramos para remover "N/A" (e qualquer outra string não numérica). Em seguida convertimos para `int`. Então usamos `allMatch(idade >= 18)`.

Para a lista `["25","17","N/A","34","22"]`: - Após filtro: ["25","17","34","22"] (remove "N/A"). - Convertido para ints: [25, 17, 34, 22]. - allMatch verifica: 25 >= 18 (true), 17 >= 18 (false) – nesse ponto já retorna false sem checar os demais. - Portanto `todosAdultos` será **false**.

Saída:

```
Todos são adultos? false
```

Porque há pelo menos uma idade (17) menor que 18. Se substituíssemos "17" por "18" ou removêssemos, então possivelmente seria true caso todos restantes >=18.

(Observação: também poderíamos ter usado `noneMatch(idade < 18)` no fim, que daria resultado equivalente.)

Resposta 8.10:

Calcular média dos quadrados positivos:

```
OptionalDouble media = valores.stream()
    .filter(x -> x > 0)
    .mapToDouble(x -> x * x)
    .average();
if(media.isPresent()) {
    System.out.println("Média = " + media.getAsDouble());
} else {
    System.out.println("Lista não possui valores positivos.");
}
```

Explicação: usamos `mapToDouble` para poder chamar `.average()` diretamente, que retorna um OptionalDouble. Tratamos caso de não haver positivos.

Para a lista `[-3,-2,-1,1,2,3,4]`: - Filtra positivos: [1,2,3,4]. - Quadrados: [1,4,9,16]. - Média = (1+4+9+16) / 4 = 30 / 4 = **7.5**.

Saída:

```
Média = 7.5
```

Caso a lista não tivesse nenhum número > 0, o OptionalDouble viria vazio e imprimiríamos a mensagem de que não há valores positivos.

Resposta 9.1:

Primeiro, vamos implementar um Spliterator customizado para um intervalo de long. Esse Spliterator vai iterar de `start` até `end` (inclusive talvez), e poderá se dividir aproximadamente ao meio. Exemplo de implementação:

```
class RangeSpliterator implements Spliterator<Long> {
    private long current;
    private final long end;
    private final long threshold;
    RangeSpliterator(long start, long end) {
        this.current = start;
        this.end = end;
        this.threshold =
10000; // tamanho mínimo para não dividir mais, para exemplo
    }
    @Override
    public boolean tryAdvance(Consumer<? super Long> action) {
        if(current <= end) {
            action.accept(current);
```

```
            current++;
            return true;
        }
        return false;
    }
    @Override
    public Spliterator<Long> trySplit() {
        long remaining = end - current + 1;
        if(remaining < threshold) {
            return null; // não divide se pouco restante
        }
        long mid = current + remaining/2;  // ponto médio
        Spliterator<Long> split = new RangeSpliterator(current, mid - 1);
        current = mid;
        return split;
    }
    @Override
    public long estimateSize() {
        return end - current + 1;
    }
    @Override
    public int characteristics() {
        return Spliterator.ORDERED | Spliterator.SIZED |
Spliterator.SUBSIZED | Spliterator.NONNULL | Spliterator.IMMUTABLE;
    }
}
```

Comentários: - O Spliterator acima divide o intervalo em duas metades sempre que possível (enquanto a metade tiver pelo menos `threshold` elementos, aqui 10000 arbitrariamente). - Características: ORDERED (há uma ordem numérica), SIZED (podemos estimar tamanho), SUBSIZED (divisões também sabem seu tamanho), NONNULL (não produz nulls), IMMUTABLE (os dados não mudam durante uso). - `tryAdvance` fornece o próximo número e incrementa `current`.

Agora, para usar esse Spliterator em um ForkJoinPool customizado:

```
ForkJoinPool pool = new ForkJoinPool(4); // pool com 4 threads
long start = 1, end = 1_000_000;
try {
    long soma = pool.submit(() ->
        StreamSupport.stream(new RangeSpliterator(start, end), true)
                     .reduce(0L, Long::sum)
    ).get();
    System.out.println("Soma de " + start + " a " + end + " = " + soma);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    pool.shutdown();
}
```

Nesse código: - `StreamSupport.stream(..., true)` cria um stream paralelo a partir do nosso Spliterator. - O `ForkJoinPool.submit()` garante que essa stream usará nosso pool em vez do comum [9] [10]. Chamamos `.get()` para aguardar o resultado. - Esperamos que a soma de 1 a 1_000_000 seja calculada (deveria ser 500000500000, fórmula da soma Gaussiana n*(n+1)/2). - Ao final, fechamos o pool com `shutdown()`.

Esse exemplo mostra duas formas de tuning: 1. **Spliterator customizado**: para controlar como os dados são divididos para paralelismo (por exemplo, evitar divisões ruins ou custos desbalanceados). 2. **Thread pool customizado**: usando um pool próprio (4 threads) para executar a operação paralela, em vez de usar o `ForkJoinPool.commonPool()` padrão que Streams paralelos usam por default [1].

Ao executar o código acima, a saída deve confirmar a soma correta. Por exemplo, espera-se:

```
Soma de 1 a 1000000 = 500000500000
```

E a computação ocorrerá utilizando exatamente 4 threads do pool customizado em paralelo, não interferindo em outras partes da aplicação que também usem paralelismo.

Resposta 9.2:

Vamos criar o Collector para lista imutável. Podemos fazê-lo manualmente ou usando `Collector.of`. A implementação manual para entendimento:

```
public class ToUnmodifiableListCollector<T> implements Collector<T, List<T>,
List<T>> {
    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }
    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;
    }
    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }
    @Override
    public Function<List<T>, List<T>> finisher() {
        return list -> Collections.unmodifiableList(list);
    }
    @Override
    public Set<Characteristics> characteristics() {
        return Collections.emptySet();
```

```
    }
}
```

Explicação: - `supplier()` fornece uma ArrayList vazia para começar a coleta. - `accumulator()` adiciona um elemento na lista acumuladora. - `combiner()` combina duas listas (necessário em paralelo) simplesmente adicionando a segunda na primeira. - `finisher()` transforma a lista mutável resultante em uma lista imutável usando `Collections.unmodifiableList`. - `characteristics()` retorna vazio: significa que precisamos aplicar o finisher (não é IDENTITY_FINISH) e que não estamos marcando como CONCURRENT ou UNORDERED especificamente. (Poderíamos marcar IDENTITY_FINISH *se* não estivéssemos transformando em unmodifiable, mas aqui temos uma transformação final.)

Uso do coletor customizado:

```
List<String> nomes = Arrays.asList("A", "B", "C");
List<String> listaImut = nomes.stream().collect(new
ToUnmodifiableListCollector<>());
System.out.println(listaImut);
listaImut.add("D");  // tentativa de modificar
```

Saída:

```
[A, B, C]
```

E a linha seguinte lançaria `UnsupportedOperationException`, pois `listaImut` é imutável.

Para simplificar, poderíamos alcançar o mesmo resultado com:

```
Collector<T, ?, List<T>> toUnmodifiableList = Collector.of(
    ArrayList::new,
    List::add,
    (l1, l2) -> { l1.addAll(l2); return l1; },
    list -> Collections.unmodifiableList(list)
);
```

Esse `toUnmodifiableList` Collector pode ser utilizado da mesma forma com `.collect(toUnmodifiableList)`. Mas o exemplo acima demonstra todos os métodos do Collector.

Vale notar que coletores customizados devem tomar cuidado com threads (por isso retornamos novas listas e unimos corretamente) e com características (aqui não marcamos como UNORDERED ou CONCURRENT, então o framework vai usar um único accumulator por thread em paralelo e combinar no final com nosso combiner).

Resposta 9.3:

Uma forma de realizar a leitura resiliente é usar `flatMap` e tratar a exceção retornando um Stream vazio quando falhar. Exemplo:

```
List<String> todasLinhas = arquivos.stream()
    .flatMap(arquivo -> {
        try {
            List<String> linhas = Files.readAllLines(Paths.get(arquivo));
            return linhas.stream();
        } catch(IOException e) {
            System.err.println("Erro lendo arquivo: " + arquivo + " -> " +
e.getMessage());
            return Stream.empty();
        }
    })
    .collect(Collectors.toList());
System.out.println("Linhas lidas: " + todasLinhas.size());
```

Explicação: - O `flatMap` para cada path tenta ler todas as linhas do arquivo (`Files.readAllLines` retorna List<String> com o conteúdo). - Em caso de sucesso, fazemos `return linhas.stream()` para incorporar as linhas ao fluxo. - Em caso de falha (IOException, arquivo não encontrado, etc.), logamos um erro (em stderr) e retornamos `Stream.empty()`. Isso efetivamente ignora aquele arquivo no resultado agregado. - Continuamos o processamento com os demais arquivos.

Assim, se "inexistente.txt" gerar uma exceção, a mensagem de erro será impressa, mas o stream continuará para "dados3.txt". O resultado `todasLinhas` conterá linhas de `dados1.txt`, `dados2.txt`, e `dados3.txt` (se existirem), mas nada de `inexistente.txt`.

Exemplo de saída (supondo `dados1.txt` e `dados2.txt` existem, `inexistente.txt` não):

```
Erro lendo arquivo: inexistente.txt -> data/inexistente.txt (No such file or
directory)
Linhas lidas: 50
```

Aqui 50 seria o total de linhas dos arquivos lidos com sucesso. O importante é que o pipeline **não lançou exceção** e continuou processando outros elementos, graças ao tratamento interno.

Nota: Usamos `Files.readAllLines` que retorna uma lista, lendo o arquivo inteiro de uma vez. Poderíamos usar `Files.lines` (que retorna Stream<String> lazy) dentro do flatMap, mas seria necessário cuidar de fechá-lo. Alternativamente, poderíamos envolver `Files.lines(path)` em um try-with-resources dentro do lambda: ler as linhas, coletar numa lista e depois fazer stream dessa lista. Para simplificar, readAllLines foi adequado.

Resposta 9.4:

Vamos modificar o pipeline inserindo peek:

```
List<Integer> resultado = lista.stream()
    .peek(x -> System.out.println("Original: " + x))
    .filter(x -> x % 2 == 0)
    .peek(x -> System.out.println("Filtrado (par): " + x))
    .map(x -> x * x)
    .peek(x -> System.out.println("Mapeado (quadrado): " + x))
    .collect(Collectors.toList());
System.out.println("Resultado final: " + resultado);
```

Para a lista `[1, 2, 3, 4]`, uma possível saída no console é:

```
Original: 1
Original: 2
Filtrado (par): 2
Mapeado (quadrado): 4
Original: 3
Original: 4
Filtrado (par): 4
Mapeado (quadrado): 16
Resultado final: [4, 16]
```

Vamos entender a ordem desses logs:

- **Original:** Cada elemento é impresso ao entrar no stream.
- O número 1 aparece como "Original: 1", depois é filtrado (1 % 2 != 0, então 1 é descartado). Não vemos logs de "Filtrado" ou "Mapeado" para 1.
- **Original: 2** aparece, 2 passa no filtro então imediatamente vemos **Filtrado (par): 2**, então 2 é mapeado para 4 e vemos **Mapeado (quadrado): 4**.
- **Original: 3** aparece, 3 é ímpar, então nada mais é impresso para 3.
- **Original: 4**, 4 passa -> **Filtrado (par): 4** -> mapeado para 16 -> **Mapeado (quadrado): 16**.

Notavelmente, a saída não está agrupada em blocos por estágio (não vemos todos "Original" primeiro, depois todos "Filtrado"). Ao invés disso, vemos que o stream processa elemento por elemento, passando por todas etapas antes de ir para o próximo elemento. Isso acontece porque as operações intermediárias do Stream são **lazy** e compõem um pipeline fusionado: o elemento 1 é processado até ser descartado pelo filtro; em seguida o elemento 2 é processado completamente (até map) antes de consumir o 3, e assim por diante, em uma execução *loteada* elemento a elemento.

Esse intercalamento fica evidente com os logs do peek, confirmando a natureza de avaliação *lazy* e *per-element* do pipeline de streams. O resultado final, coletado e impresso, é `[4, 16]` correspondendo aos quadrados dos números pares da lista original.

Resposta 9.5:

Vamos montar um exemplo comparativo. Criaremos, por exemplo, um array de 10 milhões de inteiros e somá-los de três formas. Para evitar influências de warm-up, poderíamos repetir cada medição algumas vezes, mas aqui faremos uma medição básica. Código de exemplo:

```java
// Preparar dados
int N = 10_000_000;
int[] array = new int[N];
for(int i=0; i<N; i++) {
    array[i] = i;  // por simplicidade, 0,1,2,... (soma conhecida)
}

// 1. Loop imperativo
long t0 = System.nanoTime();
long soma1 = 0;
for(int x : array) {
    soma1 += x;
}
long t1 = System.nanoTime();

// 2. Stream sequencial
long soma2 = Arrays.stream(array).asLongStream().sum();  // sum() terminal
long t2 = System.nanoTime();

// 3. Stream paralelo
long soma3 = Arrays.stream(array).parallel().asLongStream().sum();
long t3 = System.nanoTime();

// Verificação (todas somas devem ser iguais)
System.out.println("soma1 = " + soma1 + ", soma2 = " + soma2 + ", soma3 = "
+ soma3);

// Tempos em milissegundos (aproximados)
System.out.printf("Loop for: %.2f ms\n", (t1 - t0) / 1e6);
System.out.printf("Stream seq: %.2f ms\n", (t2 - t1) / 1e6);
System.out.printf("Stream par: %.2f ms\n", (t3 - t2) / 1e6);
```

Saída (exemplo em uma máquina quad-core):

```
soma1 = 49999995000000, soma2 = 49999995000000, soma3 = 49999995000000
Loop for: 30.12 ms
Stream seq: 65.45 ms
Stream par: 28.77 ms
```

Interpretação: Todas as somas batem (o resultado esperado seria n*(n-1)/2 = 10_000_000*_999_999/2 no nosso array 0..N-1, que é 49999995000000). Em termos de performance:

- O **loop for** direto foi bastante rápido (~30 ms). Isso se deve a ser bem otimizado e sem overhead de abstração.
- O **Stream sequencial** foi mais lento (~65 ms), mais que o dobro do tempo do loop no teste. Isso ocorre porque há overhead na criação do stream, boxe/desboxe (embora usamos asLongStream para evitar boxing de Integer), e iteração interna genérica. Em operações simples, esse overhead pesa mais do que o loop manual.

- O **Stream paralelo** teve desempenho similar ou ligeiramente melhor que o loop (~29 ms, marginalmente menor que 30 ms do loop). Em alguns runs, paralelo poderia ser até um pouco mais rápido se os dados forem grandes e CPU tiver núcleos ociosos. Em nossa medição, o ganho não foi grande porque somar números é muito rápido, então a sobrecarga de paralelismo quase anula o benefício. Porém, notamos que ele conseguiu equiparar o tempo do loop e foi cerca de **2x mais rápido que o stream sequencial** graças ao uso de 4 threads.

Esses resultados ilustram que: - Para tarefas simples (soma de números, filtragem leve) e tamanhos moderados, o loop tradicional costuma ser mais rápido que streams [3], por eliminar overhead. Streams sequenciais adicionam camadas de abstração que custam alguns nanos por elemento. - Streams paralelos podem superar loops e streams sequenciais quando há bastante trabalho a ser dividido e a máquina possui múltiplos núcleos. No exemplo, para 10 milhões de operações muito simples, o ganho de paralelismo foi modesto. Se a operação fosse mais complexa (CPU-bound), o parallelStream tenderia a mostrar melhoria maior proporcionalmente. - Importante: Paralelismo envolve custos de gerenciamento de threads, divisão de tarefas e combinação de resultados. Portanto, só compensa para volumes grandes de dados ou operações custosas. Caso contrário, pode até degradar a performance.

Em resumo, o loop imperativo simples permanece muito eficiente para operações simples em coleções pequenas/médias. A Stream API traz vantagens de código mais conciso e expressivo, e brilha especialmente quando usamos pipelines complexos e paralelismo em dados grandes. Entretanto, há um custo inerente a essa abstração. Conforme observado, no exemplo benchmark, o loop foi ~2x mais rápido que o stream sequencial [11]. Com parallelStream, conseguimos recuperar performance usando múltiplos cores, chegando perto ou ultrapassando o loop, mas isso depende do hardware e do caso de uso.

Citando de forma geral: *"for-loops performed much better than streams from the performance perspective"* [3] no cenário apresentado, embora *"isso pode mudar em alguns casos, especialmente com streams paralelos"* [12]. Portanto, ao escolher entre estilo imperativo e Streams, deve-se considerar o tamanho dos dados, complexidade das operações e necessidade de paralelismo.

---

[1] [9] [10] Custom Thread Pools in Java Parallel Streams | Baeldung
https://www.baeldung.com/java-8-parallel-streams-custom-threadpool

[2] How to Build a Custom Collector in Java | by Oleksandr Klymenko | Medium
https://medium.com/@alxkm/how-to-build-a-custom-collector-in-java-961bc506c57e

[3] [11] [12] Streams vs. Loops in Java | Baeldung
https://www.baeldung.com/java-streams-vs-loops

[4] [5] Java’s Stream.filter() Method Explained | Medium
https://medium.com/@AlexanderObregon/javas-stream-filter-method-explained-32185b8b4cf2

[6] [7] [8] Stream (Java Platform SE 8)
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html