01

Streams: trabalhando melhor com coleções

Transcrição

Para que a gente possa elaborar exemplos mais reais e interessantes, vamos criar uma simples classe que representa um curso do Alura. Com ela, vamos refazer alguns passos dos últimos capítulos do curso, de forma mais rápida, a fim de revê-los. Ela terá apenas dois atributos: seu nome e a quantidade de alunos, além de um construtor e seus respectivos getters:

```
class Curso {
    private String nome;
    private int alunos;

    public Curso(String nome, int alunos) {
        this.nome = nome;
        this.alunos = alunos;
    }

    public String getNome() {
        return nome;
    }

    public int getAlunos() {
        return alunos;
    }
}
```

Em uma classe com o `main`, criamos alguns cursos e inserimos em uma lista:

```
List<Curso> cursos = new ArrayList<Curso>();
cursos.add(new Curso("Python", 45));
cursos.add(new Curso("JavaScript", 150));
cursos.add(new Curso("Java 8", 113));
cursos.add(new Curso("C", 55));
```

Se quisermos ordenar esses cursos pela quantidade de alunos, podemos utilizar o `Comparator` da forma antiga, fazendo `lista.sort(new Comparator<Curso>() { .... }` e implementando nosso critério de comparação dentro da classe anônima.

E a forma nova, como fazemos? Podemos já utilizar o `Comparator.comparing`. E indicaremos que queremos que o `getAlunos` seja utilizado como critério de comparação:

```
cursos.sort(Comparator.comparingInt(c -> c.getAlunos()));
```

Esse é o caso que podemos usar um method reference para ficar ainda mais sucinto e legível:

```
cursos.sort(Comparator.comparingInt(Curso::getAlunos));
```

Pronto! Você já pode fazer um `forEach` e imprimir os nomes dos cursos para ver se o resultado é o esperado.

Streams: trabalhando com coleções no java 8

E se quisermos fazer outras tarefas com essa coleção de cursos? Por exemplo, filtrar apenas os cursos com mais de 100 alunos. Poderíamos fazer um loop que, dado o critério desejado seja atendido, adicionamos este curso em uma nova lista, a lista filtrada.

No Java 8, podemos fazer de uma forma muito mais interessante. Há como invocar um `filter`. Para sua surpresa, esse método não se encontra em `List`, nem em `Collection`, nem em nenhuma das interfaces já conhecidas. Ela está dentro de uma nova interface, a `Stream`. Você pode pegar um `Stream` de uma coleção simplesmente invocando `cursos.stream()`:

```
Stream<Curso> streamDeCurso = cursos.stream();
```

O que fazemos com ele? O `Stream` devolvido por esse método tem uma dezena de métodos bastante úteis. O primeiro é o `filter`, que recebe um predicado (um critério), que deve devolver verdadeiro ou falso, dependendo se você deseja filtrá-lo ou não. Utilizaremos um lambda para isso:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
```

Repare que o filtro devolve também um `Stream`! É um exemplo do que chamam de `fluent interface`. Vamos fazer um `forEach` e ver o resultado dos cursos:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
cursos.forEach(c -> System.out.println(c.getNome()));
```

A saída será:

```
Python
C
Java 8
Java Script
```

Estranho. Filtramos apenas os que tem mais de 100 alunos, e ele acabou listando todos! Por quê? Pois **modificações em um stream não modificam a coleção/objeto que o gerou**. Tudo que é feito nesse fluxo de objetos, nesse `Stream`, não impacta, não tem efeitos colaterais na coleção original. A coleção original continua com os mesmos cursos!

Para imprimir os cursos filtrados, podemos usar o `forEach` que existe em `Stream`:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
streamDeCurso.forEach(c -> System.out.println(c.getNome()));
```

Ou melhor ainda, podemos eliminar essa variável temporária, fazendo tudo em uma mesma linha:

```
cursos.stream().filter(c -> c.getAlunos() > 100).forEach(c -> System.out.println(c.getNome()));
```

Por uma questão de legibilidade, vamos espaçar esse código em algumas linhas, mas continuando um único `statement`:

```
cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .forEach(c -> System.out.println(c.getNome()));
```

Interessante não? E por que criaram uma nova interface e não colocar o método de filtrar dentro de `List`? Há vários motivos. Mas repare já em um primeiro: numa coleção tradicional, o que o `filter` faria? Alteraria a coleção em questão, ou manteria intacta, devolvendo uma nova coleção? Coleções no java podem ser mutáveis e imutáveis, o que complicaria ler esses métodos. No `Stream`, sabemos que esses métodos nunca alterarão a coleção original.

Vamos além. Vamos ver as outras funcionalidades. E se quisermos, dados esses cursos filtrados no nosso fluxo (`Stream`) de objetos, um novo fluxo apenas com a quantidade de alunos de cada um deles? Utilizamos o `map`:

```
cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .map(c -> c.getAlunos());
```

Se você reparar, esse `map` não devolve um `Stream<Curso>`, e sim um `Stream<Integer>`! Faz sentido. Podemos concatenar a invocação ao `forEach` para imprimirmos os dados:

```
cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .map(c -> c.getAlunos())
    .forEach(x -> System.out.println(x));
```

Aproveitamos para recapitular o que já vimos: temos a oportunidade de usar o recurso de method references duas vezes. Tanto pra invocação de `getAlunos` quanto a do `println`. Vamos alterar:

```
cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .map(Curso::getAlunos)
    .forEach(System.out::println);
```

O lambda passado para o `filter` não pode ser representado como um method reference, pois não é uma simples invocação de um único método: ele compara com um número. Pode ser que, no começo, você prefira os lambdas, pela sintaxe ser utilizada mais frequentemente. Com o tempo, verá que o method reference não impõe problema de legibilidade algum, muito pelo contrário.

Outro ponto que podemos notar: nem vimos qual é o tipo de interface que `Map` recebe! É uma `Function`, mas repare que usamos o lambda e nem foi necessário conhecer a fundo quais eram os parâmetros que ele recebia. Foi natural. É claro que, com o tempo, é importante que você domine essa nova API, mesmo que acabe utilizando majoritariamente as interfaces como lambdas.

Streams primitivos

Trabalhar com Streams vai ser frequente no seu dia a dia. Há um cuidado a ser tomado: com os tipos primitivos. Quando fizemos o `map(Curso::getAlunos)`, recebemos de volta um `Stream<Integer>`, que acaba fazendo o autoboxing dos `int`s. Isto é, utilizará mais recursos da JVM. Claro que, se sua coleção é pequena, o impacto será irrisório. Mas é possível trabalhar só com `int`s, invocando o método `mapToInt`:

```
IntStream stream = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .mapToInt(Curso::getAlunos);
```

Ele devolve um `IntStream`, que não vai gerar autoboxing e possui novos métodos específicos para trabalhar com inteiros. Um exemplo? A soma:

```
int soma = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .mapToInt(Curso::getAlunos)
   .sum()
```

Em uma única linha de código, pegamos todos os cursos, filtramos os que tem mais de 100 e somamos todos os alunos. Há também versões para `double` e `long` de `Streams` primitivos. Até mesmo o `Comparator.comparing` possui versões como `Comparator.comparingInt`, que recebe uma `IntFunction` e não necessita do boxing. Em outras palavras, todas as interfaces funcionais do novo pacote `java.util.functions` possuem versões desses tipos primitivos.

`Stream` não é uma `List`, não é uma `Collection`. E se quisermos obter uma coleção depois do processamento de um `Stream`? É o que veremos no próximo capítulo.

Não deixe de praticar bastante, descobrir novos métodos e fazer os exercícios propostos!