01

Outros sets e iterators

Transcrição

Nós vimos no curso, duas implementações de `List`, o `ArrayList` e o `LinkedList`. Para representar os alunos do `Curso`, utilizamos o `HashSet`, uma das implementações de `Set`, que como foi dito, não garante a ordem dos elementos conforme estes sejam adicionados no conjunto.

Outras implementações de Set

Mas existe uma implementação de `Set` que guarda a ordem em que os elementos foram adicionados, o `LinkedHashSet`. Podemos testá-lo, utilizando-o na classe `Curso`:

```
private Set<Aluno> alunos = new LinkedHashSet<>();
```

E executando a classe `TestaCursoComAluno`. Repare no resultado da impressão dos alunos:

```
Todos os alunos matriculados:
[Aluno: Rodrigo Turini, matricula: 34672]
[Aluno: Guilherme Silveira, matricula: 5617]
[Aluno: Mauricio Aniche, matricula: 17645]
```

Os alunos foram impressos exatamente na mesma ordem em que foram adicionados no `Set`, diferentemente do `HashSet`.

Então há implementações de conjuntos que guardam a ordem que os elementos foram adicionados. Você pode pensar: "Mas foi falado que a ordem não é preservada". E ela não é! O `Set` não guarda a ordem dos elementos, nem tem como acessá-la, a própria interface diz isso. Por isso, mesmo utilizando `LinkedHashSet`, não conseguimos invocar o método `get`, mas na hora de percorrer o conjunto com um `foreach` os seus elementos virão na ordem em que foram adicionados.

Existe ainda uma outra implementação de `Set`, chamada `TreeSet`:

```
private Set<Aluno> alunos = new TreeSet<>();
```

Mas se executarmos a classe `TestaCursoComAluno`, receberemos uma *exception*. Isso acontece porque o `TreeSet` só funciona para classes que implementam a interface `Comparable`, porque internamente o `TreeSet` guarda os objetos na sua ordem natural, que é a ordem implementada por meio do `Comparable`, mas se olharmos a classe `Aluno`, veremos que ela não implementa essa interface, por isso a exceção é lançada.

Vamos voltar a utilizar o `HashSet`, o importante aqui é saber que as coleções possuem várias implementações, tanto as listas quanto os `Set`s.

Código legado de coleções

Como o `Set` não possui um método `get`, vimos como acessar seus elementos usando o `foreach` do Java 8:

```
javaColecoes.getAlunos().forEach(aluno -> {
    System.out.println(aluno);
});
```

E antes do Java 8? Tinha o outro `for`, que também vimos no curso:

```
for (Aluno aluno : javaColecoes.getAlunos()) {
    System.out.println(aluno);
}
```

Só que esse `for` existe desde o Java 5, então como se acessava os elementos de um `Set` antes do Java 5? Era utilizado um objeto bem antigo, o `Iterator`. Ele é um objeto que todas as coleções nos dão acesso, que serve para **iterar** entre os elementos dentro da coleção, selecionando sempre o próximo objeto da coleção.

E a ordem? Se for uma lista, o `Iterator` selecionará os elementos na ordem em que estiverem nela, mas essa ordem não estará garantida no `Set`.

Então vamos mostrar agora como utilizar o `Iterator`, já que algum dia vocês poderão se deparar com um código legado que o utilize. Ainda no nosso exemplo, vamos pegar o `Set` de alunos:

```
Set<Aluno> alunos = javaColecoes.getAlunos();
```

Como foi falado, toda coleção possui `Iterator`, podemos pegá-lo usando o método de mesmo nome:

```
Set<Aluno> alunos = javaColecoes.getAlunos();
Iterator<Aluno> iterador = alunos.iterator();
```

Com o iterador em mãos, existem dois métodos que costumamos usar. O primeiro é o método `hasNext`, que devolve um booleano dizendo se há ou não um próximo elemento na coleção. Então a primeira pergunta que sempre fazemos para o iterador é: "tem um próximo elemento na coleção?". Até porque se não houver um próximo elemento, não iremos querer pegá-lo. O segundo método é o `next`, que justamente devolve o próximo elemento.

Normalmente colocamos isso dentro de um `while`:

```
Set<Aluno> alunos = javaColecoes.getAlunos();
Iterator<Aluno> iterador = alunos.iterator();

while (iterador.hasNext()) {
    System.out.println(iterador.next());
}
```

Enquanto a coleção tem um próximo elemento, nós vamos imprimindo-o.

Então é muito comum acharmos um código legado que utiliza esse `Iterator`, por isso é importante conhecê-lo.

Se precisarmos percorrer mais uma vez a coleção, precisaríamos ir na coleção e pedir um novo `Iterator`, com o método `iterator`, como fizemos anteriormente.

Um outro objeto antigo que pode ser citado é o `Vector`, que era utilizado antes da interface `Collection` existir (`Collection` existe desde o Java 1.2):

```
Vector<Aluno> vetor = new Vector<>();
```

Essa classe é muito antiga e se parece com o `ArrayList`, inclusive ela implementa `List` atualmente. A diferença é que ela pode ser utilizada por várias *threads* simultaneamente, chamado de *thread safe*. Não vamos entrar em muitos detalhes, mas o que pode ser dito é que todas as coleções faladas aqui até agora não são seguras quando utilizadas em várias threads, simultaneamente. Isso quer dizer que invocar vários métodos `add`, `get`, etc, pode acarretar algo que não esperamos, como que um "atropele" o outro, elementos sumirem, *exceptions*... A `Vector` não, ela sempre funciona, mesmo assim não é recomendada a sua utilização, já que existem outras formas de se trabalhar com coleções de maneira *thread safe*.

O que aprendemos neste capítulo:

- As implementações `LinkedHashSet` e `TreeSet`.
- Iteração de uma coleção utilizando o `Iterator`.
- A antiga classe `Vector`.