01

O poder dos sets

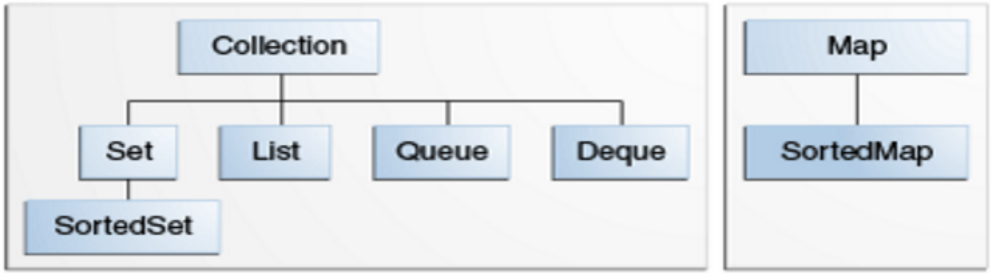
Transcrição

Vamos continuar com o curso de *Collections*, neste capítulo vamos conhecer uma nova coleção e finalmente entender o que seria uma coleção, qual sua diferença para lista e assim por diante.

Uma nova coleção, o Set

Vamos aumentar o nosso modelo para começar a trabalhar com alunos, pois um curso tem alunos. Se queremos guardar os alunos, que pertencem a determinado curso, podemos muito bem criar uma lista de alunos na classe `Curso`, assim como temos uma lista de aulas. Mas queremos dar um passo além, mostrar que dentro da biblioteca de coleções há outras opções possíveis, em vez de `List`, que podem nos ajudar em um caso específico.

Se formos abrir a documentação do Java, vocês podem perceber que várias vezes podem aparecer a seguinte imagem:



Isso é a herança das interfaces, dentro da API de coleções. Até agora só trabalhamos com a interface `List`. Sim, ela é uma interface, tanto que nunca demos `new` em uma `List`, sempre em um `ArrayList` ou `LinkedList`. O interessante é que existem outras coleções, existe uma interface `Collection`, que é a "mãe" das outras coleções, que veremos daqui para frente.

A coleção que veremos neste capítulo é a segunda coleção mais utilizada, o `Set`, que lembra muito um conjunto matemático. Então, em vez de criarmos uma lista, criaremos um *set*. Para testá-lo, usaremos a classe `TestaAlunos` para testar os futuros alunos do nosso modelo (depois criaremos a classe `Aluno`). Como `Set` é uma interface, não podemos usar o `new`, então vamos dar `new` na implementação mais utilizada dela, o `HashSet`, que iremos entender com o tempo como é o seu funcionamento e quais suas grandes vantagens, comparados com o `ArrayList`:

```
public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
    }
}
```

Agora vamos adicionar três alunos e imprimi-los, usando o já conhecido método `add`:

```
public class TestaAlunos {
```

```
    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");

        System.out.println(alunos);

    }
}
```

Mas até aqui funcionou exatamente como uma lista, então qual seria a diferença? Para mostrar uma delas, vamos adicionar mais três alunos. Teste e observe o resultado:

```
import java.util.*;

public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");
        alunos.add("Sergio Lopes");
        alunos.add("Renan Saggio");
        alunos.add("Mauricio Aniche");

        System.out.println(alunos);
    }
}
```

A ordem da impressão saiu meio estranha. Os alunos não foram impressos na ordem em que foram adicionados. E é essa a primeira característica que podemos perceber quando estamos utilizando um conjunto, um *set*, não temos garantia da ordem em que os elementos vão ficar dentro desse conjunto, desse "saco de objetos". Um conjunto (diferente de uma lista, que representa uma sequência de objetos) é uma "sacola", e lá dentro está cheio de objetos, e você não sabe em que ordem eles estão.

Mas aí você pode pensar então que um conjunto é uma opção pior que uma lista. Não necessariamente. Muitas vezes, e você vai perceber que são mais do que imagina, não é necessário que haja uma ordem entre os elementos da coleção. Podemos simplesmente querer saber quais alunos estão matriculados no curso, não nos importa quem foi o primeiro aluno a se matricular, não temos essa necessidade neste caso. Mas se tivermos essa necessidade, usaríamos uma lista.

E não é por acaso que um conjunto não tenha os métodos de acesso que utilizam a ordem do elemento, como o método `get`, por exemplo. Claro, como não temos garantia da ordem dos elementos, não podemos invocar o `get` pedindo o quarto elemento, já que como não existe ordem, não existe esse quarto elemento.

Mas e para acessar esses elementos? Podemos fazer um `foreach`:

```
public static void main(String[] args) {

    Set<String> alunos = new HashSet<>();
```

```
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");
        alunos.add("Sergio Lopes");
        alunos.add("Renan Saggio");
        alunos.add("Mauricio Aniche");

        for (String aluno : alunos) {
            System.out.println(aluno);
        }

        System.out.println(alunos);
    }
```

Com isso, imprimimos cada uma das `String` s que estão nesse conjunto.

Vantagens do uso do HashSet

Por enquanto nós "perdemos" a ordem, se compararmos o conjunto com a lista. Então quais são as vantagens?

A primeira vantagem é que ele não aceita elementos repetidos. Podemos testar isso adicionando duas `String` s iguais e depois imprimi-las. Faça o teste:

```
import java.util.*;

public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");
        alunos.add("Nico Steppat"); // outro Nico Steppat, exatamente igual ao anterior

        System.out.println(alunos);
    }
}
```

Todos os `Set` s do Java garantem para nós que só haverá um objeto dentro do conjunto, nenhum outro igual. Ele ignorará todos os outros elementos iguais, isso pode ser comprovado se imprimirmos o tamanho do conjunto, invocando o método `size` . No caso do exemplo acima, o resultado será **3**.

Mas a grande vantagem de se utilizar o conjunto é a velocidade de performance, quando utilizamos métodos que procuram objetos dentro de uma coleção (por exemplo, o método `contains`).

Toda coleção possui o método `contains` , isso porque esse método é da interface "mãe" das coleções, a `Collection` , logo uma lista também possui esse método.

O `contains` retorna um booleano dizendo se a coleção possui ou não determinado objeto que passamos para o método. Exemplo:

```
boolean pauloEstaMatriculado = alunos.contains("Paulo Silveira");
```

E esse método é extremamente rápido quando executado em um `HashSet`. Muito mais rápido que em uma lista. Por quê?

Baseado no exemplo, o `contains` de uma lista faz uma busca linear, ou seja, busca elemento por elemento, para verificar que "Paulo Silveira" não se encontra no meio dos objetos da coleção. Já o `HashSet` utiliza uma **tabela de espalhamento** para tentar fazer a busca em tempo constante, tornando a busca mais rápida.

Em um conjunto com 10 mil, 100 mil objetos, realizando buscas frequentes, a diferença do tempo de execução entre o `HashSet` e uma lista é notável.

Quando usar cada um?

Essa questão varia de acordo com a necessidade de cada um, o que é interessante é que podemos ser ainda mais genéricos quando declaramos as nossas coleções. `HashSet` implementa `Set`, que por sua vez implementa `Collection`, então podemos declarar um `HashSet` da seguinte forma:

```
Collection<String> alunos = new HashSet<>();
```

Com isso, o nosso código continua compilando, já que a maioria dos métodos que vimos até aqui pertencem à interface `Collection`, assim o nosso código fica mais flexível.

Mas podemos perceber que ainda não podemos utilizar métodos que envolvam a ordem dos elementos. Para isso, podemos utilizar um recurso que utilizamos no capítulo passado, criar uma lista, passando a coleção por parâmetro para o construtor:

```
List<String> alunosEmLista = new ArrayList<>(alunos);
```

Agora conseguimos ordenar essa lista, buscar pelo índice, e assim por diante. É comum utilizarmos várias coleções ao mesmo tempo, que compartilham os elementos entre si, para trabalharmos com eles da melhor forma.

O que aprendemos neste capítulo:

- Uma nova coleção: `Set`.
- A implementação `HashSet`.
- Vantagens e desvantagens do `Set`.
- Mais sobre a interface `Collection`.