08. Outros sets e iterators

Date	@27/10/2022
Categoria	Java
Curso	Java Collections: Dominando Listas Sets e Mapas

Tópicos

- Outros sets e iterators
- HashSet é poderoso, mas se eu precisar de acesso ordenado?
- TreeSet
- Recordar e viver
- Acesso sincronizado
- Refazendo à moda antiga
- TreeSet e Comparator

HashSet é poderoso, mas se eu precisar de acesso ordenado?

No capítulo anterior, aprendemos a lidar com conjuntos e vimos as vantagens de usarmos um `HashSet`. Além de não aceitar elementos duplicados, característica dos conjuntos, podemos realizar buscas extremamente rápidas. Ter um código que processe rapidamente é algo que alegra não apenas o desenvolvedor, mas os usuários que esperam menos os resultados.

Contudo, o HashSet, por ser um conjunto, não possui uma ordem previsível. Como assim? Imagine uma sacola de bingo. Dentro dela adicionamos várias bolas numeradas que não se repetem. Se adicionarmos primeiro a bola 11, não quer dizer que quando acessarmos o primeiro elemento do conjunto será a bola 11. Pode ser 15, 34 ou 40. Veja que essa característica do conjunto de não termos uma ordem previsível pode limitar o uso do poderoso HashSet em situações que desejamos maior performance, mas precisamos ter um acesso ordenado e previsível.

Mas nem tudo está perdido. No Collection Framework há uma estrutura de dados que usa o poder do hash e que podemos acessar os elementos de maneira previsível, isto é, se adicionarmos os elementos A, B e C teremos certeza que A é o primeiro, B é o segundo e por aí vai.

O LinkedHashSet nos dá a performance de um HashSet mas com acesso previsível e ordenado.

TreeSet

Para adicionarmos um objeto em um TreeSet ele precisa implementar a interface Comparable. Mas o que fazer se estamos trabalhando com uma instância de uma classe que não temos acesso ou não podemos modificar para implementar `Comparable`? Nesse caso, o construtor do `TreeSet` recebe como parâmetro um objeto que implementa `Comparator`. Dessa forma, o critério de comparação pode ser criado em separado da classe na qual opera.

Recordar e viver

Aprendemos a percorrer uma `Collection` em Java 8 usando o método `forEach`. Vejamos um exemplo:

```
Set<String> conjunto = new HashSet<>();
conjunto.add("A");
conjunto.add("A"); // não adiciona, já existe
conjunto.add("B");
```

Antes do Java 8 usávamos:

```
for(String letra: conjunto) {
    System.out.println(letra);
}
```

```
conjunto.forEach(letra -> {
    System.out.println(letra);
});
```

No entanto, voltando ao passo mais longínquo, mais propriamente antes do Java 5, essa estrutura não existia. Se não existia, como era possível iterar em um conjunto (Set) se ele não possui acesso indexado como uma lista que possui o método `get`? Percorríamos uma lista através de um `Iterator`!

É um objeto que todas as coleções nos dão acesso e serve para iterar entre os objetos dentro da coleção. A ordem na qual os elementos são devolvidos pelo `Iterator` depende da implementação da `Collection` utilizada.

Vejamos um exemplo:

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

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

Acesso sincronizado

Apesar de não termos entrado em detalhe no curso sobre acesso sincronizado ou `Thread Safety`, qual das opções abaixo possui uma `Collection` que é `thread-safe`?

- A classe Vector possui as mesmas características que um ArrayList, com a diferença de que o primeiro possui acesso sincronizado e o segundo não.
