01

Mapas

Transcrição

Que tal fazermos uma busca por alunos matriculados num curso? Vamos criar uma nova classe chamada `TestaBuscaAlunosNoCurso`, e aproveitar um pouco da classe `TestaCursoComAluno`, onde já temos o curso `javaColecoes`, aulas e alunos matriculados:

```
public class TestaBuscaAlunosNoCurso {

    public static void main(String[] args) {

        Curso javaColecoes = new Curso("Dominando as coleções do Java",
                "Paulo Silveira");

        javaColecoes.adiciona(new Aula("Trabalhando com ArrayList", 21));
        javaColecoes.adiciona(new Aula("Criando uma Aula", 20));
        javaColecoes.adiciona(new Aula("Modelando com coleções", 24));

        Aluno a1 = new Aluno("Rodrigo Turini", 34672);
        Aluno a2 = new Aluno("Guilherme Silveira", 5617);
        Aluno a3 = new Aluno("Mauricio Aniche", 17645);

        javaColecoes.matricula(a1);
        javaColecoes.matricula(a2);
        javaColecoes.matricula(a3);
    }
}
```

Dentro de `Curso` criamos um método que verifica se o aluno está matriculado, mas agora queremos saber algo diferente, queremos buscar um aluno dentro do curso utilizando o seu número de matrícula. Podemos fazer TDD:

```
System.out.println("Quem é o aluno com matricula 5617?");
Aluno aluno = javaColecoes.buscaMatriculado(5617);
System.out.println("Aluno: " + aluno);
```

Como sempre, damos `CTRL+1` e pedimos para o Eclipse criar a estrutura do método para nós. Como podemos descobrir se o Aluno com matrícula **5617** está no curso? De um jeito simples, basta fazermos um `foreach` em `alunos` e testar se o número é igual, correto? E caso não achemos, jogamos uma *exception* já existente do Java:

```
public Aluno buscaMatriculado(int numero) {
    for (Aluno aluno : alunos) {
        if (aluno.getNumeroMatricula() == numero) {
            return aluno;
        }
    }
    throw new NoSuchElementException("Matricula " + numero
            + " não encontrada");
}
```

Vamos testar e ver se funciona. Funciona conforme o esperado!

Uma nova estrutura

Uma pergunta: com que frequência faremos essa busca no curso? Frequentemente, correto? Além disso, o número de alunos pode ficar muito grande e a nossa busca será muito custosa.

Interessante vermos que temos a estrutura de dados eficiente para fazer associações, ou seja, dado um número (a matrícula), teremos um aluno associado correspondente, como se fosse uma tabela. O nome da estrutura que faz muito bem isso é o **Map**. Vale ressaltar que `Map` não é uma implementação de `Collection`, ele é uma interface por si só.

Como dito antes, o `Map` é muito bom em fazer associações. No nosso caso, queremos fazer uma associação entre um número inteiro (`Integer`) e um aluno (`Aluno`). Como podemos fazê-la, então? Devemos fazer isso dentro da nossa classe `Curso`, pois nossa intenção inicial era exatamente isso: buscar um aluno dentro de um curso. A implementação de `Map` mais usada é o `HashMap` :

```
public class Curso{

    private Map<Integer, Aluno> matriculaParaAluno = new HashMap<>();
    // restante do código
}
```

Ao fazer isso, teremos um mapa completamente vazio. Então, podemos modificar o método `matricula` para que, além de adicionar o aluno dentro do `Set`, ele também relacione o número de matricula com o aluno:

```
public void matricula(Aluno aluno) {
    // adiciona no Set de alunos
    this.alunos.add(aluno);
    // cria a relação no Map
    this.matriculaParaAluno.put(aluno.getNumeroMatricula(), aluno);
}
```

Estamos fazendo algo parecido com uma tabela Excel. Temos duas colunas **aluno** e **matricula** e vamos adicionando (***put***) a chave matricula e o valor aluno. E o que isso vai nos ajudar? Bem, podemos mudar o nosso `buscaMatriculado()` e deixá-lo mais simples:

```
public Aluno buscaMatriculado(int numero) {
    return this.matriculaParaAluno.get(numero);
}
```

Otimizado

Muito melhor, não? Em vez de utilizarmos um `for` que poderia demorar bastante dependendo da quantidade de alunos, basta passarmos uma **chave** (que definimos ao criar o Mapa como sendo um `Integer`) e ele irá nos retornar o aluno relacionado. E ainda ganharemos em performance, pois o algoritmo implementado dentro de `HashMap` é bastante otimizado para velocidade (usa o mesmo princípio da **tabela de espalhamento**).

E se testarmos um aluno inexistente? Passando para o método, por exemplo, uma matrícula 5618? O retorno será `null`. Esse é o padrão implementado, caso consultemos a documentação, veremos que o método `get` retorna duas opções: o valor relacionado ou `null`. Isso nos ajuda por não ter que lançar uma *exception* avisando que determinada matrícula não foi encontrada.

Você sempre pode olhar a documentação e estudar a quantidade enorme de métodos que o `HashMap` já implementou para nós.

Lembrando que a chave que definimos na declaração do `Map` tem que ser única. Podemos testar adicionando um aluno com o mesmo número de matrícula que outro, e tentando buscar esse número de matrícula:

```
Aluno a2 = new Aluno("Guilherme Silveira", 5617);
Aluno a4 = new Aluno("Paulo Silveira", 5617);

javaColecoes.matricula(a2);
javaColecoes.matricula(a4);

System.out.println("Quem é o aluno com matricula 5617?");
Aluno aluno = javaColecoes.buscaMatriculado(5617);
System.out.println("Aluno: " + aluno);
```

Veremos que apenas o último aluno adicionado é apresentado. Isso acontece porque ao adicionarmos um aluno com o mesmo número de matrícula que outro já existente, o mais antigo é esquecido pelo `Map` e o novo se torna o valor relacionado àquela matrícula.

Outras Implementações

O `HashMap`, como foi dito anteriormente, é uma das implementações mais usadas de `Map`. Mas temos outras como o `LinkedHashMap`, bastante parecido com o `LinkedHashSet`, que guarda a ordem de inserção. Ou seja, se fôssemos imprimir o `LinkedHashMap`, a impressão apareceria na ordem em que foi inserida.

Outro exemplo de `Map`, porém muito antigo, é o `HashTable`, uma implementação bem antiga de `Map`, pouco usada mas que é uma *thread safe*. Ou seja, é seguro usá-lo em um programa que tenha programação concorrente. Porém, comumente, a pessoa que necessita de um `Map` *thread safe* estudará mais sobre `threads` (inclusive temos um curso aqui (https://cursos.alura.com.br/course/threads-java-1) no Alura) e utilizará `HashMap`.

Pessoal, o que aprendemos neste curso foram as principais Coleções do Java!

É muito importante lembrar de pesquisar no Javadoc do `java.util`, pois existe muita coisa já implementada e vários códigos reutilizáveis: não reinvente a roda!

O que aprendemos neste capítulo:

- A interface `Map`.
- As implementações `HashMap` e `LinkedHashMap`.
- Vantagens e desvantagens do uso do `Map`.