01

Mais práticas com relacionamentos

Transcrição

Vamos continuar a aumentar o nosso modelo, vendo boas práticas e ideias de como trabalhar melhor quando temos coleções, métodos e outras coisas que irão auxiliar bastante no nosso dia a dia. Para este capítulo, vamos criar a classe `TestaCurso2`, que é baseada na classe `TestaCurso`. Ela cria o mesmo curso e adiciona as mesmas aulas, que nós já vimos anteriormente:

```
public class TestaCurso2 {

    public static void main(String[] args) {

        Curso javaColecoes = new Curso("Dominando as colecoes 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 colecoes", 24));

    }
}
```

Mas o que mais podemos fazer com o curso? Quando tínhamos somente as aulas, nós as ordenamos. Então o que podemos fazer aqui é ordenar as aulas do curso. Mas primeiro, vamos pegá-las e imprimi-las:

```
public class TestaCurso2 {

    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 colecoes", 24));

        List<Aula> aulas = javaColecoes.getAulas();
        System.out.println(aulas);

    }
}
```

O resultado no console são as aulas sendo impressas na ordem em que elas foram inseridas na lista:

```
[[Aula: Trabalhando com ArrayList, 21 minutos], [Aula: Criando uma Aula, 20 minutos], [Aula: Mo
```

Para ordená-las, nós já sabemos utilizar o método `sort` da classe `Collections`. Então vamos utilizá-lo e imprimir novamente as aulas. Faça o teste:

```
import java.util.*;

public class TestaCurso2 {

    public static void main(String[] args) {

        Curso javaColecoes = new Curso("Dominando as colecoes 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 colecoes", 24));

        List<Aula> aulas = javaColecoes.getAulas();
        System.out.println(aulas);

        Collections.sort(aulas);
        System.out.println(aulas);

    }
}

public class Curso {

    private String nome;
    private String instrutor;
    private List<Aula> aulas = new LinkedList<Aula>();

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

    public String getNome() {
        return nome;
    }

    public String getInstrutor() {
        return instrutor;
    }

    public List<Aula> getAulas() {
        return Collections.unmodifiableList(aulas);
    }

    public void adiciona(Aula aula) {
        this.aulas.add(aula);
    }

    public int getTempoTotal() {
        return this.aulas.stream().mapToInt(Aula::getTempo).sum();
    }
}
```

```
public class Aula implements Comparable<Aula> {

    private String titulo;
    private int tempo;

    public Aula(String titulo, int tempo) {
        this.titulo = titulo;
        this.tempo = tempo;
    }

    public String getTitulo() {
        return titulo;
    }

    public int getTempo() {
        return tempo;
    }

    @Override
    public String toString() {
        return "[Aula: " + this.titulo + ", " + this.tempo + " minutos]";
    }

    @Override
    public int compareTo(Aula outraAula) {
        return this.titulo.compareTo(outraAula.getTitulo());
    }
}
```

Recebemos uma *exception*, que nos é familiar. No capítulo passado, recebemos a mesma *exception*, `UnsupportedOperationException`, que aconteceu quando tentamos invocar o método `add` diretamente do `getAulas`, em vez de utilizar o método `adiciona`.

E aqui está ocorrendo a mesma coisa, o método `getAulas` retorna uma ***unmodifiable list***, ou seja, retorna uma lista de aulas que não pode ser modificada. Ou seja, nós não podemos ficar invocando métodos que adicionam na lista, que removam seus itens e nem que mudem de ordem os seus elementos, justamente o que o `sort` está tentando fazer.

Então parece que encontramos um problema de trabalhar com a *unmodifiable list*. Mas na verdade esse não é o problema, é a solução! Eu não quero que ninguém fique mexendo no atributo privado de aulas. Ele tem essa ordenação porque eu quero, se alguém quiser alterar essa ordem, ele tem que me pedir, e não simplesmente chegar no atributo e sair modificando-o.

Mas será que faz sentido termos um método para ordenar as aulas, algo como `ordenaAulas`? Não há necessidade disso, seria estranho precisarmos criar métodos (que já existem na API) para manipular as aulas. Existe uma outra solução: geralmente há um construtor das nossas coleções que recebem o próprio tipo, para construir um igual, como se fosse um clone. Para isso, basta passarmos a lista de aulas para o construtor do `ArrayList`, por exemplo:

```
public class TestaCurso2 {

    public static void main(String[] args) {

        Curso javaColecoes = new Curso("Dominando as colecoes 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 colecoes", 24));

        List<Aula> aulasImutaveis = javaColecoes.getAulas();
        System.out.println(aulasImutaveis);

        List<Aula> aulas = new ArrayList<>(aulasImutaveis);
    }
}
```

Como isso é algo bem comum, a API já tem algo pronto para nós! Assim, não precisamos fazer um `for` e ir adicionando todas as aulas da lista `aulasImutaveis` em uma nova lista. Basta passar a lista `aulasImutaveis` para o construtor e ele já fará isso, criando uma nova lista com os mesmos elementos da lista que estamos passando no construtor!

Agora podemos ordenar as aulas e imprimi-las:

```
public class TestaCurso2 {

    public static void main(String[] args) {

        Curso javaColecoes = new Curso("Dominando as colecoes 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 colecoes", 24));

        List<Aula> aulasImutaveis = javaColecoes.getAulas();
        System.out.println(aulasImutaveis);

        List<Aula> aulas = new ArrayList<>(aulasImutaveis);

        Collections.sort(aulas);
        System.out.println(aulas);
    }
}
```

E o resultado mostrado no console são as aulas em ordem alfabética:

```
[[Aula: Trabalhando com ArrayList, 21 minutos], [Aula: Criando uma Aula, 20 minutos], [Aula: Mo
[[Aula: Criando uma Aula, 20 minutos], [Aula: Modelando com coleções, 24 minutos], [Aula: Traba
```

Você pode pensar então que seria melhor não trabalhar com *unmodifiable lists*, mas com a experiência e prática, você verá que essa alternativa deixa mais fácil a manutenção do seu código.

Tempo total das aulas de um curso

Para continuar com a implementação do nosso modelo, podemos exibir agora o tempo total das aulas de um curso. Poderíamos fazer um `for` nas aulas do curso, no próprio `main`, somar todos os seus tempos e imprimir. Mas outras

pessoas podem querer no futuro exibir também esse tempo total, então faz sentido que o curso tenha um método que retorne esse tempo para nós.

Na classe `Curso`, vamos criar o método `getTempoTotal`. Ele percorrerá a lista de aulas do curso e somará os seus tempos, utilizando uma variável auxiliar:

```
public int getTempoTotal() {
    int tempoTotal = 0;
    for (Aula aula : aulas) {
        tempoTotal += aula.getTempo();
    }
    return tempoTotal;
}
```

Agora podemos imprimir o tempo total na classe `TestaCurso2`:

```
System.out.println(javaColecoes.getTempoTotal());
```

Poderíamos ter feito isso de várias maneiras, como por exemplo ter um atributo de classe `tempoTotal`, e toda vez que uma aula fosse adicionada, somaríamos o seu tempo no `tempoTotal`:

```
public void adiciona(Aula aula) {
    this.aulas.add(aula);
    this.tempoTotal += aula.getTempo();
}
```

Mas aqui nós iremos utilizar um jeito novo, que tem no curso do Java 8 (https://cursos.alura.com.br/course/java8-lambdas). No Java 8, toda coleção tem um método que se chama `stream`, não iremos entrar em detalhes, considerando que o nosso foco são as boas práticas e API de *Collections*. Ao invocarmos esse método, nós pediremos os inteiros porque trabalhamos com o tempo - que é um inteiro. Ele se chamará `mapToInt` e passaremos para ele qual campo inteiro queremos (`Aula::getTempo`). No final, nós somaremos esses valores chamando o método `sum`:

```
public int getTempoTotal() {
    return this.aulas.stream().mapToInt(Aula::getTempo).sum();
}
```

O que queremos mostrar aqui é que podemos implementar o método `getTempoTotal` de várias formas diferentes, mas o `TesteCurso2` continuará funcionando, pois todo o nosso código está encapsulado.

Imprimindo um curso

Por último, queremos poder imprimir um curso. Na classe `TestaCurso2`, se imprimirmos o `javaColecoes`, não teremos um resultado muito agradável. Faça o teste com o exemplo abaixo:

```
import java.util.*;

public class TestaCurso2 {
```

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

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

        System.out.println(javaColecoes);
    }
}

public class Curso {

    private String nome;
    private String instrutor;

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

    public String getNome() {
        return nome;
    }

    public String getInstrutor() {
        return instrutor;
    }
}
```

Queremos fazer o que fizemos com a aula. O que fizemos? Sobrescrevemos o método `toString` da classe `Curso`:

```
public class Curso {

    // restante do código

    @Override
    public String toString() {
        return "[Curso: " + this.getNome() + ", tempo total: " + this.getTempoTotal()
                + ", aulas: + " + this.aulas + "]";
    }
}
```

E temos o seguinte resultado:

```
[Curso: Dominando as coleções do Java, tempo total: 65, aulas: + [[Aula: Trabalhando com ArrayL
```

No método `toString`, talvez não seja das melhores ideias ficar concatenando `String`s e outros objetos, fizemos aqui só para conseguirmos ver como se trabalha com coleções de coleções, no caso, um curso que tem muitas aulas.

Um ponto a ser ressaltado é a classe `Collections`, utilizada muitas vezes aqui. Devemos ficar bastante atentos. Ela disponibiliza vários métodos, como por exemplo para embaralhar uma lista, inverter a ordem da mesma, trocar dois elementos de posição, entre outros métodos que podem ser muito úteis para nós. Vamos ver mais alguns desses métodos no decorrer do curso.

Mas é fundamental vocês sempre estarem atentos, procurarem na API, "abusarem" do comando `CTRL + Espaço` do Eclipse, pois descobrirão que muitas coisas que pensam em fazer, possivelmente já estão prontas para serem utilizadas.

O que aprendemos neste capítulo:

- Uma solução para o *unmodifiable list*.
- `Stream` do Java 8.