01

Relacionamentos com coleções

Transcrição

Vamos aprimorar nosso modelo de classes para torná-lo mais real. O objetivo é ter uma riqueza de classes e você poder enxergar o dia a dia do uso das coleções.

Para isso, vamos criar uma classe que representa um `Curso`. Esse `Curso` vai ter um punhado de `Aulas`, que representaremos através de uma lista de `Aula`. Todo curso possuirá também um `nome`, um `instrutor`, o construtor que acharmos necessário e também os getters:

```
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 aulas;
    }
}
```

Opa! Uma `LinkedList`? Qual é a diferença dela para uma `ArrayList`? Você vai ver que vamos utilizá-la da mesma forma que uma `ArrayList` e mais para a frente saberemos as vantagens e desvantagens de cada uma.

Repare que, em vez de declararmos a referência a uma `ArrayList<Aula>` (ou `LinkedList<Aula>`), deixamos mais genérico, utilizando a interface `List`. Por quê? Pelo motivo que já vimos ao estudar orientação a objetos aqui no Alura: não temos motivo para ser super específico na instância que iremos usar. Se forçarmos `ArrayList` na referência, certamente teremos problema o dia que precisarmos trocar essa lista. Se declararmos apenas como `List`, poderemos mudar de implementação, como para uma `LinkedList`, sem problema algum de compilação, por não termos nos comprometido com uma implementação em específico. Fique tranquilo se você ainda não está convencido dessas vantagens. Com tempo de programação e de prática em orientação a objetos isso ficará mais claro.

Vamos testar essa nossa classe `Curso`, adicionando uma aula e mostrando o resultado:

```
public class TestaCurso {
```

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

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

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

O que vai sair aqui? O resultado é `[]`, representando uma coleção vazia. Faz sentido, pois inicializamos nossa lista de aulas com um `new LinkedList` que estará vazio.

Vamos fazer uma brincadeira com as variáveis para ver se você já está acostumado com a forma que o Java trabalha. E se eu adicionar uma aula no `javaColecoes` e imprimir novamente o resultado? Será que a variável `aulas` continuará vazia, já que adicionamos a nova `Aula` dentro da lista do curso?

Para isso, vamos usar a invocação de `javaColecoes.getAulas().add(...)`. Claro, você pode quebrar essa instrução em duas linhas, mas é bom você se acostumar com uma invocação que logo em seguida faz outra invocação. Nesse caso, estamos invocando o `getAulas` e logo em seguida invocando o `add` no que foi retornado pelo `getAulas`:

```
public class TestaCurso {

    public static void main(String[] args) {

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

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

        javaColecoes.getAulas().add(new Aula("Trabalhando com ArrayList", 21));
        System.out.println(aulas);
    }
}
```

O resultado é o esperado por muitos:

```
[]
[Aula: Trabalhando com ArrayList, 21 minutos]
```

Isso é apenas para reforçar que trabalhamos aqui com referências. A variável `aulas` se referencia para uma lista de objetos, que é a mesma que nosso atributo interno do curso em questão se referencia. Isto é, tanto `javaColecoes.getAulas()` quanto a nossa variável temporária `aulas` levam ao mesmo local, à mesma coleção.

Tem gente que vai falar que "se mexeu numa variável, mexeu na outra". Não é bem isso. Na verdade, são duas variáveis distintas mas que se referenciam ao mesmo objeto.

Apenas a classe Curso deve ter acesso às aulas

É comum aparecer trechos de código como `javaColecoes.getAulas().add(...)`. É até fácil de ler: pegamos o curso `javaColecoes`, para depois pegar suas aulas e aí então adicionar uma nova aula.

Mas acabamos violando alguns princípios bons de orientação a objetos. Nesse caso, seria interessante que fosse necessário pedir a classe `Curso` para que fosse adicionada uma `Aula`, possibilitando fazer algo como `javaColecoes.adiciona(...)`. E isso é fácil: basta adicionarmos esse método em `Curso`:

```
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 aulas;
    }

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

E com isso podemos fazer:

```
public class TestaCurso {

    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));

        System.out.println(javaColecoes.getAulas());
    }
}
```

Mas quando alguém for usar a classe `Curso`, ela vai acabar fazendo `javaColecoes.adiciona(...)` ou `javaColecoes.getAulas().add(...)`? Se deixarmos assim, ele poderá fazer de ambas as formas.

Queremos que ele só faça da primeira forma, usando nosso novo método `adiciona`. Como forçar isso? Não há como forçar, mas há como programar *defensivamente*, fazendo com que o método `getAulas` devolva uma *cópia* da coleção de aulas. Melhor ainda: podemos devolver essa cópia de tal forma que ela não possa ser alterada, ou seja, que ela seja não modificável, usando o método `Collections.unmodifiableList`:

```
public class Curso {
    /// restante do código...

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

Veja o código completo abaixo e faça o teste:

```
import java.util.LinkedList;
import java.util.List;
import java.util.Collections;

public class TestaCurso {
    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));

        // tentando adicionar da maneira "antiga". Podemos fazer isso? Teste:
        javaColecoes.getAulas().add(new Aula("Trabalhando com ArrayList", 21));

        System.out.println(javaColecoes.getAulas());
    }
}
```

```
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 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());
    }
}
```

Repare que uma exception será lançada ao tentarmos executar `javaColecoes.getAulas().add`. Qualquer tentativa de modificação vai lançar essa exception, indicando algo como "*opa! você não pode alterar o estado dessa coleção aqui, encontre outra forma de fazer o que você quer*".

LinkedList ou ArrayList?

E o mistério da `LinkedList`? E se tivéssemos usado `ArrayList` na declaração do atributo `aulas` da classe `Curso`? O resultado seria exatamente o mesmo!

Então qual é a diferença? Basicamente performance. O `ArrayList`, como diz o nome, internamente usa um *array* para guardar os elementos. Ele consegue fazer operações de maneira muito eficiente, como invocar o método `get(indice)`.

Se você precisa pegar o décimo quinto elemento, ele te devolverá isso bem rápido. Quando um `ArrayList` é lento? Quando você for, por exemplo, inserir um novo elemento na primeira posição. Pois a implementação vai precisar mover todos os elementos que estão no começo da lista para a próxima posição. Se há muitos elementos, isso vai demorar... Em computação, chamamos isso de **consumo de tempo linear**.

Já o `LinkedList` possui uma grande vantagem aqui. Ele utiliza a estrutura de dados chamada **lista ligada**, e é bastante rápido para adicionar e remover elementos na *cabeça* da lista, isto é, na primeira posição. Mas é lento se você precisar acessar um determinado elemento, pois a implementação precisará percorrer todos os elementos até chegar ao décimo quinto, por exemplo.

Confuso? Não tem problema. Sabe o que é interessante? Você não precisa tomar essa decisão desde já e oficializar para sempre. Como utilizamos a referência a `List`, comprometendo-nos pouco, podemos *sempre* mudar a implementação, isso é, em quem damos `new`, caso percebamos que é melhor uma ou outra lista nesse caso em particular.

Se você gosta desse assunto e gostaria de conhecer profundamente os algoritmos e estruturas de dados por trás das coleções do Java, há o curso de estrutura de dados com esse enfoque, que você pode acessar Aqui (https://www.alura.com.br/curso-online-estrutura-de-dados)

O que aprendemos neste capítulo:

- A implementação `LinkedList`.
- Encapsulamento e princípios de Orientação a Objeto.
- Programação defensiva.