01

Listas de objetos

Transcrição

Começando daqui? Você pode fazer o download (https://github.com/alura-cursos/java-collections/archive/aula2.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Trabalhar com uma lista de `Strings` foi fácil. E se for de uma classe que nós mesmos criamos? Faz diferença?

Vamos criar uma classe `Aula` que possui um `titulo` e o `tempo` em minutos. Um construtor populará esses atributos. Nem vamos colocar setters, já que não temos essa necessidade por enquanto:

```
public class 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;
    }
}
```

Para brincarmos com objetos do tipo `Aula`, criaremos uma `TestaListaDeAula` e adicionaremos aulas dentro de uma `List<Aula>`:

```
public class TestaListaDeAula {

    public static void main(String[] args) {

        Aula a1 = new Aula("Revistando as ArrayLists", 21);
        Aula a2 = new Aula("Listas de objetos", 20);
        Aula a3 = new Aula("Relacionamento de listas e objetos", 15);

        ArrayList<Aula> aulas = new ArrayList<>();
        aulas.add(a1);
        aulas.add(a2);
        aulas.add(a3);

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

Qual será o resultado? O nome das três aulas? Na verdade, não. O método `toString` da classe `ArrayList` percorre todos os elementos da lista, concatenando seus valores também de `toString` . Como a classe `Aula` não possui um `toString` reescrito (`_override_`), ele utilizará o `toString` definido em `Object` , que retorna o nome da classe, concatenado com um `@` e seguido de um identificador único do objeto. Algo como:

```
[Aula@c3bfe4, Aula@d24512, Aula@c13eaa1]
```

Se a sua classe `Aula` estiver dentro de um pacote , e deveria estar, a saída será `br.com.alura.Aula@c3bfe4` , pois o `_full qualified name_` , ou `_nome completo da classe_` , é sempre o nome do pacote concatenado com `.` e o nome curto da classe.

Reescrevendo nosso toString para trabalhar bem com a lista

Vamos então reescrever nosso método `toString` da classe `Aula` , para que ele retorne algo significativo:

```
public class Aula {

    // ... restante do código aqui

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

Confira nosso código completo, que você pode fazer pequenas modificações e testar, inclusive removendo o `toString` por completo:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class TestaListaDeAula {

    public static void main(String[] args) {

        Aula a1 = new Aula("Revistando as ArrayLists", 21);
        Aula a2 = new Aula("Listas de objetos", 20);
        Aula a3 = new Aula("Relacionamento de listas e objetos", 15);

        ArrayList<Aula> aulas = new ArrayList<>();
        aulas.add(a1);
        aulas.add(a2);
        aulas.add(a3);

        System.out.println(aulas);
    }
}

public class 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]";
    }
}
```

E se tentássemos adicionar uma `String`, em vez de uma `Aula`, dentro dessa lista, o que aconteceria? Faça o teste!

Ordenando uma lista de objetos nossos

O que acontece se tentamos utilizar o `Collections.sort` em uma lista de `Aula`?

```
ArrayList<Aula> aulas = new ArrayList<>();
aulas.add(a1);
aulas.add(a2);
aulas.add(a3);
Collections.sort(aulas);
```

A invocação `Collections.sort(aulas)` não compila! Por quê? Pois `Collections.sort` não sabe ordenar uma lista de `Aula`. De qual forma ele faria isso? Pelo nome da aula ou pela duração? Não daria para saber.

Mas como ele saberia então ordenar uma `List<String>`? Será que há um código específico dentro de `sort` que verifica se a lista é de `String`s? Certamente não.

O que acontece é: quem implementou a classe `String` definiu um *critério de comparação* entre duas `String`s, no qual qualquer pessoa pode comparar dois desses objetos. Isso é feito através do método `compareTo`. Faça o seguinte teste:

```
public class TestaComparacaoStrings {

    public static void main(String[] args) {

        String s1 = "paulo";
        String s2 = "silveira";
        int resultado = s1.compareTo(s2);

        System.out.println(resultado);
```

```
    }
}
```

O resultado da comparação é um `int`, pois um `boolean` não bastaria. Esse método devolve um número negativo se `s1` é *menor* que `s2`, um número positivo se `s2` é *menor* que `s1` e 0 se forem *iguais*. Mas o que é maior, menor e igual? No caso da `String`, quem implementou a classe decidiu que o *critério de comparação* seria a ordem lexicográfica (alfabética, por assim dizer). Pode ver isso direto na documentação:

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String- (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String-)

E como nós devemos fazer na classe `Aula`? Um método parecido? Mais que isso: um método com a mesma assinatura! Pois a `String` implementa uma interface chamada `Comparable`, do pacote `java.lang`, que define exatamente esse método! Você pode ver também que o método `Collections.sort` recebe uma `List` de qualquer tipo de objeto que implementa `Comparable`.

Vamos então implementar essa interface na classe `Aula`

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

    // ... restante do código aqui

    @Override
    public int compareTo(Aula outraAula) {
        // o que colocar aqui?
    }
}
```

É aí que devemos decidir o nosso *critério de comparação* de duas aulas. Quando uma aula virá antes da outra? Bem, eu vou optar por ordenar na ordem alfabética. Para isso, vou aproveitar do próprio método `compareTo` da `String`, delegando:

```
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.titulo);
    }
}
```

Agora podemos testar essa classe no `Collections.sort` :

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class TestaListaDeAula {

    public static void main(String[] args) {

        Aula a1 = new Aula("Revistando as ArrayLists", 21);
        Aula a2 = new Aula("Listas de objetos", 20);
        Aula a3 = new Aula("Relacionamento de listas e objetos", 15);

        ArrayList<Aula> aulas = new ArrayList<>();
        aulas.add(a1);
        aulas.add(a2);
        aulas.add(a3);

        // antes de ordenar:
        System.out.println(aulas);

        Collections.sort(aulas);

        // depois de ordenar:
        System.out.println(aulas);
    }
}

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.titulo);
    }
}
```

Ordenando com outro critério de comparação

E se quisermos ordenar essa lista de acordo com o tempo de uma aula? Poderíamos alterar o método `compareTo`, mas assim todas as ordenações de aulas seriam afetadas.

Uma opção é utilizar o segundo argumento que o `Collections.sort` recebe. É um comparador, representado pela interface `Comparator` do pacote `java.util` (cuidado! o nome é semelhante com `Comparable`, mas tem um papel bem diferente).

Você pode implementar essa interface e depois invocar `Collections.sort(aulas, novoComparadorDeAulas)`, onde `novoComparadorDeAulas` é uma instância de um `Comparator<Aula>`.

Parece complicado? Há uma forma mais enxuta de se fazer isso, utilizando os recursos do Java 8. Não é o nosso foco aqui. Existe um curso exclusivo desse assunto na Alura, mas é sempre bom ir se acostumando. Veja como ficaria:

```
Collections.sort(aulas, Comparator.comparing(Aula::getTempo));
```

A frase aqui é semelhante a "*ordene estas `aulas` utilizando como comparação o retorno do método `getTempo` de cada `Aula`*".

Podemos deixá-la mais bonita. Toda lista, a partir do Java 8, possui um método `sort` que recebe `Comparator`. Poderíamos então fazer:

```
aulas.sort(Comparator.comparing(Aula::getTempo));
```

O que aprendemos neste capítulo:

- A utilidade em reescrever o método `toString`.
- `Collections.sort` e o método `compareTo`.
- `Comparator` e recursos do Java 8.