Transcrição

Conhecer a API de Collections é algo mais que essencial para o desenvolvedor Java. Elas estão em todos os lugares, você utilizando ou não.

Dentre elas, a `ArrayList` é a que aparece com maior frequência. Antes de chegarmos em toda a hierarquia das tais "Collections", vamos praticar bastante as operações essenciais das listas, mais especificamente essa implementação. O que são exatamente as Collections? Vai ficar mais claro no decorrer do curso. Pense nelas como classes que ajudam você a manipular um punhado de objetos.

Para esse curso você estará o tempo todo importando classes e interfaces do pacote `java.util`, **fique atento!** Caso contrário você pode acabar importando classes de outros pacotes que possuem o mesmo nome.

Crie um novo projeto chamado `gerenciador-de-cursos` e vamos programar! Mesmo que você já conheça o conteúdo dessas duas primeiras aulas, que envolvem a utilização dos métodos básicos e ordenação, vale a pena recapitular, para então entrarmos em boas práticas, outras coleções e uso no dia a dia de forma real.

Adicionando elementos em uma lista

Para criar um objeto do tipo `ArrayList`, certamente fazemos como sempre: utilizando o operador `new`. Mas repare que acabamos passando um pouco mais de informações. Ao declarar a referência a uma `ArrayList`, passamos qual o tipo de

objeto com o qual ela trabalhará. Se queremos uma lista de nomes de aulas, vamos declarar `ArrayList<String>`. Crie a classe `TestandoListas`, adicionando os nomes de algumas aulas que teremos nesse curso:

```
import java.util.List;
import java.util.ArrayList;

public class TestandoListas {

    public static void main(String[] args) {

        String aula1 = "Modelando a classe Aula";
        String aula2 = "Conhecendo mais de listas";
        String aula3 = "Trabalhando com Cursos e Sets";

        ArrayList<String> aulas = new ArrayList<>();
        aulas.add(aula1);
        aulas.add(aula2);
        aulas.add(aula3);

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

COPIAR CÓDIGO

Qual é o resultado desse código? Ele mostra as aulas adicionadas em sequência! Por que isso acontece? Pois a classe `ArrayList`, ou uma de suas mães, reescreveu o método `toString`, para que internamente fizesse um `for`, concatenando os seus elementos internos separados por vírgula.

Removendo elementos

Bastante simples! O que mais podemos fazer com uma lista? As operações mais básicas que podemos imaginar, como por exemplo remover um determinado elemento. Usamos o método `remove` e depois mostramos o resultado para ver que a primeira foi removida:

```
aulas.remove(0);
System.out.println(aulas);
```

COPIAR CÓDIGO

Por que `0`? Pois as listas, assim como a maioria dos casos no Java, são indexadas a partir do `0`, e não do `1`.

Percorrendo uma lista

Bem, talvez não seja a melhor das ideias fazer um `System.out.println` na nossa lista, pois talvez queiramos mostrar esses itens de alguma outra forma, como por exemplo um por linha. Como fazer isso? Utilizando o `for` de uma maneira especial, chamada de `enhanced for`, ou popularmente `foreach`. Lembrando que `foreach` não existe no Java como comando, e sim como um caso especial do `for` mesmo. Olhe o código:

```
for (String aula : aulas) {
    System.out.println("Aula: " + aula);
}
```

COPIAR CÓDIGO

Acessando elementos

E se eu quisesse saber apenas a primeira aula? O método aqui é o `get`. Ele retorna o primeiro elemento se passarmos o `0` como argumento:

```
String primeiraAula = aulas.get(0);
System.out.println("A primeira aula é " + primeiraAula);
```

COPIAR CÓDIGO

Você pode usar esse mesmo método para percorrer a lista toda, em vez do tal do `enhanced for`. Para isso, precisamos saber quantos elementos temos nessa lista. Nesse caso, utilizamos o método `size` para limitar o nosso `for`:

```
for (int i = 0; i < aulas.size(); i++) {
    System.out.println("aula : " + aulas.get(i));
}
```

COPIAR CÓDIGO

Fizemos até `i < aulas.size()` pois `size` retorna o total de elementos. Se acessássemos até `i <= aulas.size()` teríamos um problema! Uma exception do tipo `IndexOutOfBoundsException` seria lançada! Quer ver? Vamos imprimir o `size` e faça o teste com o código que temos até aqui:

```
import java.util.List;
import java.util.ArrayList;

public class TestandoListas {

    public static void main(String[] args) {

        String aula1 = "Modelando a classe Aula";
        String aula2 = "Conhecendo mais de listas";
        String aula3 = "Trabalhando com Cursos e Sets";

        ArrayList<String> aulas = new ArrayList<>();
```

```
        aulas.add(aula1);
        aulas.add(aula2);
        aulas.add(aula3);

        System.out.println(aulas);
        System.out.println(aulas.size());

        // cuidado! <= faz sentido aqui?
        for (int i = 0; i <= aulas.size(); i++) {
            System.out.println("Aula: " + aulas.get(i));
        }
    }
}
```

COPIAR CÓDIGO

Mais uma forma de percorrer elementos, agora com Java 8

Percorrer com o `enhanced for` é uma forma bastante indicada. Já o `for` que fizemos utilizando o `get` possui alguns problemas que veremos em uma próxima aula e vai ficar bastante claro.

Uma outra forma de percorrer nossa lista é utilizando as sintaxes e métodos novos incluídos no Java 8. Temos um método (não um comando!) agora que se chama `forEach`. Ele recebe um objeto do tipo `Consumer`, mas o interessante é que você não precisa criá-lo, você pode utilizar uma sintaxe bem mais enxuta, mas talvez assustadora a primeira vista, chamada **lambda**. Repare:

```
aulas.forEach(aula -> {
    System.out.println("Percorrendo:");
    System.out.println("Aula " + aula);
});
```

COPIAR CÓDIGO

Estranho não? Lambda não é o foco desse curso. Existe um curso no Alura (https://cursos.alura.com.br/course/java8-lambdas) que vai tratar apenas desse assunto e é bastante aconselhado. Aqui estamos falando que, para cada `String aula`, determinado bloco de código deve ser executado. Essa variável `aula` poderia ter o nome que você desejasse.

Ordenando a lista

Esse é bastante fácil quando temos uma `List<String>`. A classe `java.util.Collections` (repare o `s` no final, que é diferente da interface `Collection`, que será vista mais para a frente) é um conjunto de métodos estáticos auxiliares as coleções. Dentro dela há o método `sort`:

```
Collections.sort(aulas);
```

COPIAR CÓDIGO

Simples, não? Mas há bastante mágica ocorrendo aqui por trás. Como é que essa classe sabe ordenar listas de `Strings`? E se fosse uma lista de, digamos, `Aula`, também funcionaria? Segure um pouco essas questões e por enquanto vamos fechar esta aula testando esse simples código final:

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

class TestandoListas {

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

```
        String aula1 = "Modelando a classe Aula";
        String aula2 = "Conhecendo mais de listas";
        String aula3 = "Trabalhando com Cursos e Sets";

        ArrayList<String> aulas = new ArrayList<>();
        aulas.add(aula1);
        aulas.add(aula2);
        aulas.add(aula3);

        System.out.println(aulas);

        Collections.sort(aulas);
        System.out.println("Depois de ordenado:");
        System.out.println(aulas);
    }
}
```

COPIAR CÓDIGO

O que aprendemos neste capítulo:

- A implementação `ArrayList`.
- O pacote `java.util`.
- Métodos de manipulação do `ArrayList`.
- `ForEach` do Java 8.

Preparando Ambiente em todos SOs

Apesar de a maior parte dos exercícios deste curso serem executados diretamente por aqui, na plataforma, é importante que você tenha a JDK instalado em sua máquina para que possa testá-los localmente e também para os demais exercícios propostos.

Siga aqui os passos de instalação de uma VM no sistema operacional que você usa: Linux, Mac OSX ou Windows:

Linux

No Ubuntu o processo de instalação mais rápido e simples é o do open-jdk, uma implementação open source do JDK:

```
sudo add-apt-repository ppa:openjdk-r/ppa
sudo apt-get update
sudo apt-get install openjdk-8-jdk
```

COPIAR CÓDIGO

No Linux Fedora você pode instalar com:

```
su -c "yum install java-1.8.0-openjdk"
```

COPIAR CÓDIGO

Após a instalação no seu Linux teste seu java com os comandos:

```
javac -version
```

```
java -version
```

Mac OSX

No Mac OSX você pode baixar a versão 1.8 do Java SDK em http://jdk.java.net/8/ (http://jdk.java.net/8/) Após executar o instalador, será necessário entrar no painel de Preferences, Java Preferences e alterar a versão do Java para a nova que você acaba de instalar.

Após a instalação no seu Mac OSX teste seu java com os comandos :

```
javac -version
```

```
java -version
```

Windows

Por fim, para instalar o SDK do Windows, acesse:
http://www.oracle.com/technetwork/java/ (http://www.oracle.com/technetwork/java/)

Dentre os top downloads, escolha o Java SE:
http://www.oracle.com/technetwork/java/javase/downloads/index.html
(http://www.oracle.com/technetwork/java/javase/downloads/index.html)

Escolha então o JDK (Java Development Kit) e por fim a versão de seu sistema operacional. Execute o arquivo jdk-versão-windows-arquitetura-p.exe e passe pelo wizard de instalação. O registro de sua VM no site da Oracle é opcional.

Clique agora com o botão da direita sobre o `Computador`, escolha `Propriedades`. Na aba `Configurações Avançadas do Sistema` clique em `Variáveis de Ambiente`.

Clique no botão `Novo` para adicionar uma nova variável: seu nome é `JAVA_HOME` (tudo maiúsculo) e seu valor será o diretório onde instalou o Java (provavelmente algo como C:\Program Files\Java\jdk1.8.0_03).

Clique no botão `Novo` para adicionar uma nova variável: seu nome é `PATH` (tudo maiúsculo) e seu valor será o `%JAVA_HOME%`.

Agora vamos alterar a variável PATH. **Não crie uma variável nova, altere a variável PATH que já existe**. Escolha ela e clique em `Editar`. No final do valor atual **complemente** com o valor `%JAVA_HOME%\bin`. Não se esqueça do ponto e vírgula que separa o path anterior desse novo path que estamos colocando.

Pronto, feche todas as janelas e abra o prompt, indo em `Iniciar`, `Executar` e digite `cmd`.

Após a instalação no seu Windows teste seu java com os comandos

```
java -version
```

```
javac -version
```

Após configurar a VM e o SDK, o resultado do comando *javac -version* deve ser a versão de seu Java, como por exemplo 1.8.03 (depende somente da versão instalada na sua máquina).

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;
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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]
```

COPIAR CÓDIGO

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]";
    }
}
```

COPIAR CÓDIGO

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

```java
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]";
    }
}
```

COPIAR CÓDIGO

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);
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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?
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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));
```

COPIAR CÓDIGO

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));
```

COPIAR CÓDIGO

O que aprendemos neste capítulo:

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

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;
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

O resultado é o esperado por muitos:

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

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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());
    }
}
```

COPIAR CÓDIGO

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);
```

```
    }
}
```

COPIAR CÓDIGO

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());
    }
}
```

COPIAR CÓDIGO

```
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);
    }
}
```

COPIAR CÓDIGO

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

COPIAR CÓDIGO

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)](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.

Diferença entre ArrayList e LinkedList

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. A `ArrayList`, como diz o nome, internamente usa uma array para guardar os elementos. Ela consegue fazer umas operações de maneira muito eficiente, como invocar o método `get(indice)`. Se você precisa pegar o décimo quinto elemento, ele te devolve isso bem rápido. Onde uma `ArrayList` é lenta? 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... chamamos isso em computação de consumo de tempo linear.

Já a `LinkedList` possui uma grande vantagem aqui. Ela utiliza a estrutura de dados chamada lista ligada. Ela é muito rápida para adicionar e remover elementos na *cabeça* da lista, isso é, na primeira posição. Mas ela é lenta 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.

Ainda está confuso? Veja código abaixo que testa a inserção de 1 milhão de elementos com `ArrayList` e `LinkedList`, medindo o tempo. Além disso, estamos removendo 100 elementos, sempre tirando do início da lista. Execute e veja a diferença:


```
ort java.util.ArrayList;
ort java.util.LinkedList;
ort java.util.List;

lic class TesteListas {

 public static void main(String[] args) {

     System.out.println("**** ArrayList vs LinkedList ***");

     List<Integer> numerosArrayList = new ArrayList<>();
     List<Integer> numerosLinkedList = new LinkedList<>();
     int quantidadeElementos = 1000000;

     long tempoArrayList  = insereElementosNo(numerosArrayList, qu
     long tempoLinkedList = insereElementosNo(numerosLinkedList, q

     System.out.println("Inserção na ArrayList demorou  " + tempoA
     System.out.println("Inserção na LinkedList demorou " + tempoL

     tempoArrayList = removePrimeirosElementos(numerosArrayList);
     tempoLinkedList = removePrimeirosElementos(numerosLinkedList)
```

```
        System.out.println("Remoção da ArrayList demorou  " + tempoAr
        System.out.println("Remoção da LinkedList demorou " + tempoLi
    }

    /*
     * removendo 100 elementos sempre na primeira posição
     */
    private static long removePrimeirosElementos(List<Integer> numerc
        long ini = System.currentTimeMillis();

        for (int i = 0; i < 100; i++) {
            numeros.remove(0); //removendo sempre o primeiro elementc
        }
        long fim = System.currentTimeMillis();
        return fim-ini;
    }

    private static long insereElementosNo(List<Integer> numeros, int
        long ini = System.currentTimeMillis();
        for (int i = 0; i < quantidade; i++) {
            numeros.add(i);
        }
        long fim = System.currentTimeMillis();
        return fim-ini;
    }
```

COPIAR CÓDIGO

29%

ATIVIDADES
6 de 6

FÓRUM DO CURSO

VOLTAR PARA DASHBOARD

MODO NOTURNO

ABRIR CADERNO

L

73.2k xp

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));

    }
}
```

COPIAR CÓDIGO

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);

    }
}
```

COPIAR CÓDIGO

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: Modela
```

COPIAR CÓDIGO

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:

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

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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);
    }
}
```

COPIAR CÓDIGO

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: Modela
[[Aula: Criando uma Aula, 20 minutos], [Aula: Modelando com coleções, 24 minutos], [Aula: Trabalhan
```

COPIAR CÓDIGO

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;
}
```

COPIAR CÓDIGO

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

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

COPIAR CÓDIGO

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();
}
```

COPIAR CÓDIGO

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();
}
```

COPIAR CÓDIGO

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;
    }
}
```

COPIAR CÓDIGO

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 + "]";
    }
}
```

COPIAR CÓDIGO

E temos o seguinte resultado:

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

COPIAR CÓDIGO

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.

Outros métodos de Collections.

Além do método `sort()` que vimos neste capítulo, a classe `Collections` também possui muitos outros métodos interessantes. Vamos dar uma olhada em alguns:

`Collections.reverse()`

O método `reverse()` serve para inverter a ordem de uma lista. As vezes precisamos imprimir uma lista de nomes do último para o primeiro, ou uma lista de `ids` do maior para o menor e é nestas horas que utilizamos o reverse para inverter a ordem natural da lista para nós.

`Collections.shuffle()`

O método `shuffle()` serve para embaralhar a ordem de uma lista. Por exemplo em um caso de um sistema de sorteio, em que precisamos de uma ordem aleatória na nossa lista, utilizamos o método shuffle para embaralhá-la.

`Collections.singletonList()`

O método `singletonList()` nos devolve uma lista imutável que contêm um único elemento especificado. Ele é útil quando precisamos passar um único elemento para uma API que só aceita uma Collections daquele elemento.

Collections.nCopies()

O método `nCopies()` nos retorna uma lista imutável com a quantidade escolhida de um determinado elemento. Se temos uma lista específica e precisamos obter uma outra lista imutável , contendo diversas cópias de um destes objetos, utilizamos o método nCopies(). O bom deste método é que mesmo que nós solicitemos uma lista com um número grande, como 10000 objetos, ele na verdade se referencia a apenas um, ocupando assim um pequeno espaço.

Este método também é utilizado para inicializar Listas recém criadas com Null, já que ele pode rapidamente criar diversos objetos, deste modo:

```
List<Type> list = new ArrayList<Type>(Collections.nCopies(1000,
(Type)null));
```

Estes são apenas alguns exemplos dos diversos métodos da classe **Collections**.

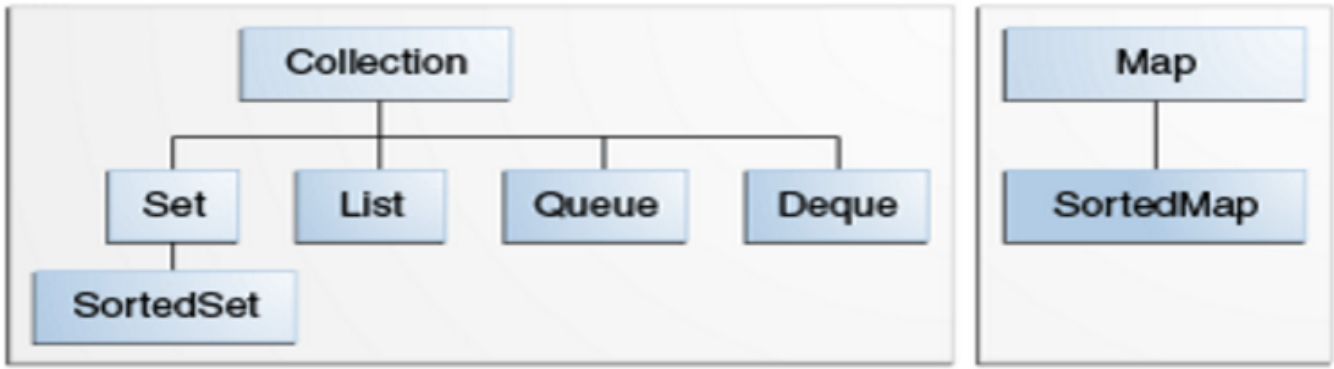
Transcrição

Vamos continuar com o curso de *Collections*, neste capítulo vamos conhecer uma nova coleção e finalmente entender o que seria uma coleção, qual sua diferença para lista e assim por diante.

Uma nova coleção, o Set

Vamos aumentar o nosso modelo para começar a trabalhar com alunos, pois um curso tem alunos. Se queremos guardar os alunos, que pertencem a determinado curso, podemos muito bem criar uma lista de alunos na classe `Curso`, assim como temos uma lista de aulas. Mas queremos dar um passo além, mostrar que dentro da biblioteca de coleções há outras opções possíveis, em vez de `List`, que podem nos ajudar em um caso específico.

Se formos abrir a documentação do Java, vocês podem perceber que várias vezes podem aparecer a seguinte imagem:



Isso é a herança das interfaces, dentro da API de coleções. Até agora só trabalhamos com a interface `List`. Sim, ela é uma interface, tanto que nunca demos `new` em uma `List`, sempre em um `ArrayList` ou `LinkedList`. O interessante é que existem outras coleções, existe uma interface `Collection`, que é a "mãe" das outras coleções, que veremos daqui para frente.

A coleção que veremos neste capítulo é a segunda coleção mais utilizada, o `Set`, que lembra muito um conjunto matemático. Então, em vez de criarmos uma lista, criaremos um *set*. Para testá-lo, usaremos a classe `TestaAlunos` para testar os futuros alunos do nosso modelo (depois criaremos a classe `Aluno`). Como `Set` é uma interface, não podemos usar o `new`, então vamos dar `new` na implementação mais utilizada dela, o `HashSet`, que iremos entender com o tempo como é o seu funcionamento e quais suas grandes vantagens, comparados com o `ArrayList`:

```
public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
    }
}
```

COPIAR CÓDIGO

Agora vamos adicionar três alunos e imprimi-los, usando o já conhecido método `add`:

```
public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
```

```
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");

        System.out.println(alunos);

    }
}
```

COPIAR CÓDIGO

Mas até aqui funcionou exatamente como uma lista, então qual seria a diferença? Para mostrar uma delas, vamos adicionar mais três alunos. Teste e observe o resultado:

```
import java.util.*;

public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");
        alunos.add("Sergio Lopes");
        alunos.add("Renan Saggio");
        alunos.add("Mauricio Aniche");

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

COPIAR CÓDIGO

A ordem da impressão saiu meio estranha. Os alunos não foram impressos na ordem em que foram adicionados. E é essa a primeira característica que podemos perceber quando estamos utilizando um conjunto, um *set*, não temos garantia da ordem em que os elementos vão ficar dentro desse conjunto, desse "saco de objetos". Um conjunto (diferente de uma lista, que representa uma sequência de objetos) é uma "sacola", e lá dentro está cheio de objetos, e você não sabe em que ordem eles estão.

Mas aí você pode pensar então que um conjunto é uma opção pior que uma lista. Não necessariamente. Muitas vezes, e você vai perceber que são mais do que imagina, não é necessário que haja uma ordem entre os elementos da coleção. Podemos simplesmente querer saber quais alunos estão matriculados no curso, não nos importa quem foi o primeiro aluno a se matricular, não temos essa necessidade neste caso. Mas se tivermos essa necessidade, usaríamos uma lista.

E não é por acaso que um conjunto não tenha os métodos de acesso que utilizam a ordem do elemento, como o método `get`, por exemplo. Claro, como não temos garantia da ordem dos elementos, não podemos invocar o `get` pedindo o quarto elemento, já que como não existe ordem, não existe esse quarto elemento.

Mas e para acessar esses elementos? Podemos fazer um `foreach`:

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

    Set<String> alunos = new HashSet<>();
    alunos.add("Rodrigo Turini");
    alunos.add("Alberto Souza");
    alunos.add("Nico Steppat");
    alunos.add("Sergio Lopes");
    alunos.add("Renan Saggio");
    alunos.add("Mauricio Aniche");

    for (String aluno : alunos) {
        System.out.println(aluno);
```

```
        }

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

COPIAR CÓDIGO

Com isso, imprimimos cada uma das `String` s que estão nesse conjunto.

Vantagens do uso do HashSet

Por enquanto nós "perdemos" a ordem, se compararmos o conjunto com a lista. Então quais são as vantagens?

A primeira vantagem é que ele não aceita elementos repetidos. Podemos testar isso adicionando duas `String` s iguais e depois imprimi-las. Faça o teste:

```
import java.util.*;

public class TestaAlunos {

    public static void main(String[] args) {

        Set<String> alunos = new HashSet<>();
        alunos.add("Rodrigo Turini");
        alunos.add("Alberto Souza");
        alunos.add("Nico Steppat");
        alunos.add("Nico Steppat"); // outro Nico Steppat, exatamente igual ao anterior

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

```
    }
}
```

COPIAR CÓDIGO

Todos os `Set`s do Java garantem para nós que só haverá um objeto dentro do conjunto, nenhum outro igual. Ele ignorará todos os outros elementos iguais, isso pode ser comprovado se imprimirmos o tamanho do conjunto, invocando o método `size`. No caso do exemplo acima, o resultado será **3**.

Mas a grande vantagem de se utilizar o conjunto é a velocidade de performance, quando utilizamos métodos que procuram objetos dentro de uma coleção (por exemplo, o método `contains`).

Toda coleção possui o método `contains`, isso porque esse método é da interface "mãe" das coleções, a `Collection`, logo uma lista também possui esse método.

O `contains` retorna um booleano dizendo se a coleção possui ou não determinado objeto que passamos para o método. Exemplo:

```
boolean pauloEstaMatriculado = alunos.contains("Paulo Silveira");
```

COPIAR CÓDIGO

E esse método é extremamente rápido quando executado em um `HashSet`. Muito mais rápido que em uma lista. Por quê?

Baseado no exemplo, o `contains` de uma lista faz uma busca linear, ou seja, busca elemento por elemento, para verificar que "Paulo Silveira" não se encontra no meio dos objetos da coleção. Já o `HashSet` utiliza uma **tabela de espalhamento** para tentar fazer a busca em tempo constante, tornando a busca mais rápida.

Em um conjunto com 10 mil, 100 mil objetos, realizando buscas frequentes, a diferença do tempo de execução entre o `HashSet` e uma lista é notável.

Quando usar cada um?

Essa questão varia de acordo com a necessidade de cada um, o que é interessante é que podemos ser ainda mais genéricos quando declaramos as nossas coleções. `HashSet` implementa `Set`, que por sua vez implementa `Collection`, então podemos declarar um `HashSet` da seguinte forma:

```
Collection<String> alunos = new HashSet<>();
```

COPIAR CÓDIGO

Com isso, o nosso código continua compilando, já que a maioria dos métodos que vimos até aqui pertencem à interface `Collection`, assim o nosso código fica mais flexível.

Mas podemos perceber que ainda não podemos utilizar métodos que envolvam a ordem dos elementos. Para isso, podemos utilizar um recurso que utilizamos no capítulo passado, criar uma lista, passando a coleção por parâmetro para o construtor:

```
List<String> alunosEmLista = new ArrayList<>(alunos);
```

COPIAR CÓDIGO

Agora conseguimos ordenar essa lista, buscar pelo índice, e assim por diante. É comum utilizarmos várias coleções ao mesmo tempo, que compartilham os elementos entre si, para trabalharmos com eles da melhor forma.

O que aprendemos neste capítulo:

- Uma nova coleção: `Set`.
- A implementação `HashSet`.
- Vantagens e desvantagens do `Set`.

- Mais sobre a interface `Collection`.

Velocidade de busca das listas e conjuntos

Crie a classe `TestaPerformance`, com um método `main` e um código que insere 50 mil números em uma `ArrayList` e os pesquisa. Vamos usar o método `currentTimeMillis()`, de `System`, para cronometrar o tempo gasto com a adição e pesquisa dos elementos:

```
public class TestaPerformance {

    public static void main(String[] args) {

        Collection<Integer> numeros = new ArrayList<Integer>();

        long inicio = System.currentTimeMillis();

        for (int i = 1; i <= 50000; i++) {
            numeros.add(i);
        }

        for (Integer numero : numeros) {
            numeros.contains(numero);
        }

        long fim = System.currentTimeMillis();

        long tempoDeExecucao = fim - inicio;
```

```
        System.out.println("Tempo gasto: " + tempoDeExecucao);

    }

}
```

COPIAR CÓDIGO

Troque o `ArrayList` por `HashSet` e verifique o tempo que vai demorar:

```
Collection<Integer> numeros = new HashSet<>();
```

COPIAR CÓDIGO

O que é lento? A inserção dos 50 mil elementos ou as 50 mil buscas? Descubra computando o tempo gasto em cada `for` separadamente.

Se você passar de 50 mil para um número maior, como 100 mil, verá que isso inviabiliza por completo o uso de uma `List` em casos que quisermos utilizá-la essencialmente para pesquisas.

Opinião do instrutor

No caso do `ArrayList`, a inserção é bem rápida e a busca **muito lenta**!

No caso do `HashSet`, a inserção ainda é rápida, embora um pouco mais lenta do que a das listas. Mas a busca é **muito rápida**!

Transcrição

Vamos continuar com o nosso modelo? Agora que conhecemos um pouco de `Set`, vamos colocar em prática em uma classe nova. Criaremos a classe `Aluno`, com os atributos `nome` e `matricula`, sem esquecer do construtor e dos *getters* (lembre-se de utilizar o atalho do Eclipse):

```
public class Aluno {

    private String nome;
    private int numeroMatricula;

    public Aluno(String nome, int numeroMatricula) {
        this.nome = nome;
        this.numeroMatricula = numeroMatricula;
    }

    public String getNome() {
        return nome;
    }

    public int getNumeroMatricula() {
        return numeroMatricula;
    }
}
```

COPIAR CÓDIGO

Para testar, vamos criar uma nova classe também: `TestaCursoComAluno`, porque agora vamos colocar uma coleção de alunos dentro da classe `Curso`.

Para fazer isso, precisaremos decidir qual coleção usar. Pode surgir aqui um debate. Uma `Lista`, por exemplo, pode ser uma boa escolha, porque você quer guardar a ordem, pode existir algum aluno repetido... Porém, estudando nosso domínio, resolvemos ficar com um `Set` de alunos.

Então, na classe `Curso`, vamos adicionar:

```
private Set<Aluno> alunos = new HashSet<>();
```

COPIAR CÓDIGO

Vamos começar a testar? Podemos copiar de `TestaCurso2` algumas linhas de código, na qual criamos um curso e adicionamos aulas. Precisamos de alguns alunos também, já que queremos colocá-los num curso. Então:

```
public class TestaCursoComAluno {

    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);
    }
}
```

COPIAR CÓDIGO

Agora podemos fazer uso de algo bem comum no dia a dia da programação Java, que é o TDD (***Test Driven Development***, que traduzido para o português, significa **Desenvolvimento guiado por testes**). Fica muito bom para nós, por exemplo, ter uma maneira de matricular nossos alunos no curso. "Ter uma maneira" significa sempre "criar um método". Então vamos escrevê-lo! Nós podemos fazer isto, mesmo sem ter o método:

```
javaColecoes.matricula(a1);
```

COPIAR CÓDIGO

O Eclipse vai nos avisar que esse método não existe, mas isso já sabíamos! Podemos dar `CTRL + 1` nessa linha e selecionar a opção de criá-lo dentro da classe `Curso`. Dentro do método `matricula`, podemos agora deixar o código bem encapsulado, e colocar:

```
public void matricula(Aluno aluno){
    this.alunos.add(aluno);
}
```

COPIAR CÓDIGO

Agora o Eclipse deixa de reclamar. Podemos matricular os três alunos no curso:

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

```
javaColecoes.matricula(a3);
```

COPIAR CÓDIGO

Mas para mostrar os alunos do curso, como faremos? Agora é uma boa hora para criarmos o *getter* de alunos em `Curso`. Mas qual é a boa prática? Devolver `Collections.unmodifiableSet()`, o mesma que fizemos com as aulas, assim será impossível que alguém fora da classe possa adicionar ou remover objetos:

```
public Set<Aluno> getAlunos() {
    return Collections.unmodifiableSet(alunos);
}
```

COPIAR CÓDIGO

Agora vamos mostrar todos os nossos alunos. Podemos tanto usar o `forEach` comum quanto o especial, com o `lambda` do Java 8, ficando assim:

```
System.out.println("Todos os alunos matriculados: ");
javaColecoes.getAlunos().forEach(aluno -> {
    System.out.println(aluno);
});
```

COPIAR CÓDIGO

Vamos rodar e... Apenas objetos! Podemos resolver isso facilmente colocando `aluno.getNome()`, porém, podemos solucionar isso sobrescrevendo o método `toString()` da classe `Aluno`, como já fizemos neste curso:

```
@Override
public String toString() {
    return "[Aluno: " + this.nome + ", matricula: " + this.numeroMatricula + "]";
}
```

COPIAR CÓDIGO

Rodando novamente, temos tudo certinho e formatado! Mas perceba que ele não manteve a ordem... Sabemos o porquê disso: O `Set` não garante a ordem dos elementos inseridos. A ordem pode até ser igual com um pouco de sorte e poucos testes, porém não é isso que acontece no mundo real, pois como foi dito, o `Set` não guarda a sequência. Caso você realmente precise de uma ordem fiel, começando necessariamente pelo primeiro elemento inserido, então a melhor opção seria você utilizar uma `List`.

O que aprendemos neste capítulo:

- Aplicação do `Set` no nosso modelo.
- Programação defensiva com conjuntos.

Collections.emptySet()

No código abaixo usamos mais um método da classe `Collections`. Nesse caso criamos um conjunto vazio.

Será que podemos adicionar um elemento nesse conjunto? Execute o código a seguir para ter certeza!

```
import java.util.*;

public class TesteEmptySet {

    public static void main(String[] args) {

        Set<String> nomes = Collections.emptySet();
        nomes.add("Paulo"); //o que acontece aqui?
    }

}
```

COPIAR CÓDIGO

Opinião do instrutor

Também não podemos! Recebemos a mesma exceção:

```
UnsupportedOperationException
```

Um conjunto destinado a ser vazio não pode ter um elemento, certo? Então faz sentido receber essa exceção. Mas para que um conjunto vazio poderia ser útil?

Por exemplo, imagina que você precisa representar um curso que foi cancelado pois não teve matriculas. Nesse caso faria todo sentido devolver um `Collections.emptySet()`

Para saber mais: Coleções threadsafe

Já usamos bastante a classe `java.util.Collections`. Vimos os métodos `Collections.unmodifiableSet(..)` e `Collections.emptySet(..)` entre vários outros. Vale muito explorar essa classe!

Tem um método que gostaria de chamar atenção:

```
Set<Aluno> alunosSincronizados = Collections.synchronizedSet(alu
```

COPIAR CÓDIGO

Tente descobrir qual é o objetivo desse método. Segue o link do Javadoc Collections (https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html).

Opinião do instrutor

Uma das características mais interessantes de JVM é que ela sabe trabalhar em paralelo. Internamente isso é feito por meio de **Threads** que funcionam como pequenos processos dentro da JVM.

O problema é que as coleções que estamos usando até agora não foram feitas para serem manipuladas em paralelo. No entanto, nada impede que usemos um método da classe `Collections` para transformar uma coleção comum em uma coleção para threads. É justamente isso que o método faz, retorna um nova coleção que pode ser compartilhada entre threads sem perigos.

O tópico de Threads é importante pois elas são muito utilizadas dentro das bibliotecas que rodam no mundo Java. Na Alura, temos dois treinamentos dedicados a threads:

- https://cursos.alura.com.br/course/threads-java-1 (https://cursos.alura.com.br/course/threads-java-1)
- https://cursos.alura.com.br/course/threads-java-2 (https://cursos.alura.com.br/course/threads-java-2)

Transcrição

Agora, queremos usufruir da grande vantagem dos `Set` s: **a velocidade**. Queremos perguntar para coleção, por exemplo, se determinado aluno está matriculado:

```
System.out.println("O aluno " + a1.getNome() + " está matriculado?");
System.out.println(javaColecoes.estaMatriculado(a1));
```

COPIAR CÓDIGO

Novamente, usaremos o TDD para criar o método automaticamente, basta usar o comando `CTRL + 1` e o Eclipse criará para você. Porém precisamos alterar o retorno para `boolean` . Já dentro do método, utilizaremos-nos de um outro método que está presente em todas as classes que implementam a interface `Collection` , o `contains` . Com isso, vamos delegar a funcionalidade do método `estaMatriculado` para um já existente:

```
public boolean estaMatriculado(Aluno aluno) {
    return this.alunos.contains(aluno);
}
```

COPIAR CÓDIGO

Agora, o `contains` utilizará a estrutura bem implementada da **tabela de espalhamento**, e irá retornar rapidamente `true` ou `false` para nós. Testando... Funciona! O aluno inserido algumas linhas antes, `a1` , está matriculado no curso. Se comentarmos a linha de inserção, o método nos retornará `false` .

O método equals

Porém, existe um grande problema, bastante comum ao trabalhar com conjuntos, o problema do `equals`. Imagine que estamos nos utilizando de um *web service* e ele possui um formulário perguntando quem estamos procurando. Se vamos digitar no formulário, o seu retorno será uma String com o nome do aluno procurado:

```
String alunoProcurado = "Rodrigo Turini";
```

COPIAR CÓDIGO

Não podemos procurá-lo com o nosso método anterior, pois o método `estaMatriculado` recebe um objeto do tipo Aluno como parâmetro. Então vamos criar um objeto exatamente igual ao aluno `a1` criado anteriormente, e passá-lo na função, para saber se ele está ou não dentro do curso:

```
Aluno turini = new Aluno("Rodrigo Turini", 34672);
System.out.println("E esse Turini, está matriculado?");
System.out.println(javaColecoes.estaMatriculado(turini));
```

COPIAR CÓDIGO

Testando e... Deu `false`! Esse `turini` não é o mesmo aluno que adicionamos no curso como `a1`. Mas isso já sabíamos da orientação a objetos, se dermos um `new`, mesmo que o objeto contenha tudo igual, ele não fará referência ao primeiro, e portanto, são diferentes. Você pode testar executando o exemplo abaixo:

```
public class TestaCursoComAluno {

    public static void main(String[] args) {

        Aluno aluno = new Aluno("Douglas Quintanilha", 11824763);
```

```
        Aluno alunoQueVeioDoFormulario = new Aluno("Douglas Quintanilha", 11824763);

        System.out.println("O aluno e igual ao aluno que veio do formulario?");
        System.out.println(aluno == alunoQueVeioDoFormulario);
    }
}

public class Aluno {

    private String nome;
    private int numeroMatricula;

    public Aluno(String nome, int numeroMatricula) {
        this.nome = nome;
        this.numeroMatricula = numeroMatricula;
    }
}
```

COPIAR CÓDIGO

Olhando a documentação da interface `Collection` e indo no método `contains`, veremos que ele utiliza o método `equals`. Sabemos que a definição do `equals` usada pelo Java nem sempre é a que queremos. Usando nosso caso de alunos, sabemos que `a1.equals(turini) == true`, porém isso não é verdadeiro para o Java.

Por isso, precisamos reescrever o método `equals` na nossa classe `Aluno`. Para nós, dois alunos são iguais se ambos tiverem o mesmo nome, então vamos ao trabalho:

```
@Override
public boolean equals(Object obj) {
    Aluno outroAluno = (Aluno) obj;
```

```
    return this.nome.equals(outroAluno.nome);
}
```

COPIAR CÓDIGO

Lembrando que é preciso ter cuidado nos casos em que o nome seja `null`. Podemos nos defender disso colocando uma condição no construtor em que só seja possível criar o objeto se o nome não for `null`:

```
public Aluno(String nome, int numeroMatricula) {
    if (nome == null) {
        throw new NullPointerException("Nome não pode ser nulo");
    }
    this.nome = nome;
    this.numeroMatricula = numeroMatricula;
}
```

COPIAR CÓDIGO

E assim garantimos que no método `equals`, não teremos problemas com `NullPointerException`. Vamos testar agora e ver se "a1 é equals ao Turini":

```
System.out.println("O a1 é equals ao Turini?");
System.out.println(a1.equals(turini));
```

COPIAR CÓDIGO

E sim, é `true`! Porém, nossa comparação de cima ainda não funciona:

```
Aluno turini = new Aluno("Rodrigo Turini", 34672);
System.out.println("E esse Turini, está matriculado?");
System.out.println(javaColecoes.estaMatriculado(turini));
```

COPIAR CÓDIGO

Temos `false` como resultado. No entanto, se mudamos o `equals`, por que ele continua dizendo que `turini` não está matriculado? Ao comparar `turini` com `a1`, o resultado é `true` (como visto no nosso teste), porém o `estaMatriculado` nos retorna `false`.

O método hashCode

Vamos à explicação: a estrutura `Set` usa uma **tabela de espalhamento** para realizar mais rapidamente suas buscas. Imagine que cada vez que você adiciona algo dentro do seu `Set` para espalhar os objetos, um número mágico é gerado e todos os objetos que o tenham são agrupados. E ao buscar, em vez de comparar o objeto com todos os outros objetos contidos dentro do `Set` (isso daria muitas comparações), ele gera novamente o mesmo número mágico, e compara apenas com aqueles que também tiveram como resultado esse número. Ou seja, ele compara apenas dentro do grupo de semelhança. No caso da matricula não reconhecida, o aluno `a1` estava num grupo diferente de `turini`, e por isso o método `contains` não conseguia encontrá-lo.

Como é gerado esse número mágico? Utilizando o método `hashCode`, por isso precisamos sobrescrevê-lo, mudando-o para quando criarmos um objeto `Aluno` com o mesmo nome, que esses objetos gerem o mesmo `hashCode` e portanto, fiquem no mesmo grupo. Podemos por exemplo pegar o primeiro caractere do nome. Dessa maneira, estaremos dividindo os grupos em grupos de alunos que começam com **A, B, C, D, ...**, e `Rodrigo Turini` tanto em `a1` quanto em `turini` estarão no grupo **R**:

```
@Override
public int hashCode(){
    return this.nome.charAt(0);
}
```

COPIAR CÓDIGO

Testando, vemos que funciona! Mas temos outro probleminha... O espalhamento é feito para que se tenha o menor número possível de objetos dentro de um grupo, e separando alfabeticamente como estamos fazendo, não é a maneira mais eficiente. Entrando na classe `String` do Java, vemos que ela tem o método `hashCode` implementado, e ele já faz uma conta bem difícil, para que haja o melhor espalhamento e assim, a busca seja bastante eficiente. Então, podemos fazer com que o nosso `hashCode` devolva o `hashCode` da `String` `nome`:

```
@Override
public int hashCode(){
    return this.nome.hashCode();
}
```

COPIAR CÓDIGO

Se rodarmos o código novamente, temos `true` em todos os testes. Considere a seguinte **regra**: caso você sobrescreva o método `equals`, obrigatoriamente deverá sobrescrever o método `hashCode`.

O que aprendemos neste capítulo:

- Implementação das nossas próprias regras de comparação entre objetos de uma mesma classe.
- Sobrescrita do método `equals`.
- A necessidade de sobrescrever o método `hashCode` quando o `equals` for sobrescrito.

Considerando hashCode e Equals

Sobre as afirmativas abaixo, diga quais são as verdadeiras:

A classe String já possui um método para gerar um hashCode a partir do seu conteúdo. Este método é bem eficiente e confiável.

a classe String já possui um método próprio para gerar um hashCode e a não ser que estejamos trabalhando em um caso específico, podemos sempre usá-lo quando preciso.

B

Quando estamos verificando se um aluno pertence a um **Set** de alunos, é interessante porém **não essencial** sobrescrever o método hashCode da classe Aluno. O método `.contains()` de **Set** também pode comparar o `.equals()` caso o hashCode não tenha sido sobrescrito.

É uma boa prática - pra não dizer que é quase obrigatório - sempre que reescrevemos o método equals também reescrevermos o método hashCode, já que mesmo que no nosso código não utilize nenhum Set , existem diversos códigos que o utilizam, e caso não sobrescrevemos este método podemos esbarrar em bugs não esperados.

Apesar de ser perigoso, se estamos verificando se um elemento pertence a uma implementação de **List**, só precisamos reescrever o método `equals()` , já que o método `.contains()` de List só utiliza o equals para comparação.

Quando estamos usando o método `.contains()` de **List** ele utiliza apenas o `.equals()` para comparar dois objetos, por isso a afirmativa é verdadeira.

Para saber mais: O contrato do método equals

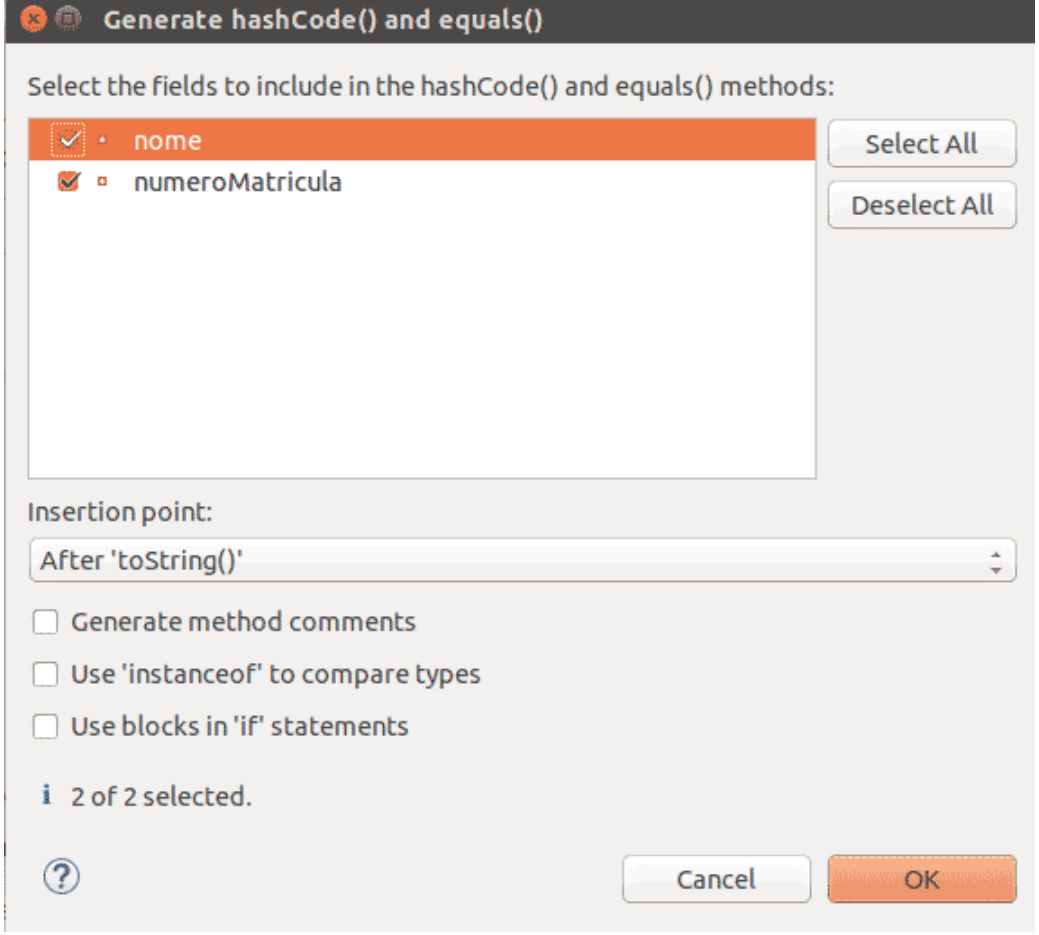
Nossa implementação do método `equals` é funcional, porém em alguns casos mais específicos podemos ter alguns problemas. Existe um contrato mais avançado que devemos seguir para implementar um método `equals` eficiente:

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

Por conta dessas propriedades uma implementação sofisticada do método `equals` pode ser bem trabalhosa, por essa razão que as IDE's fornecem recursos que implementam esse método para nós.

No Eclipse você pode pressionar **CTRL + 3** e digitar **equals**.

Podemos ainda escolher os atributos que queremos utilizar na comparação:



Sendo assim, utilize esse recurso para implementar os métodos *equals* e *hashCode* da classe `Aluno`.

Opinião do instrutor

A implementação deve ficar parecida com essa:

```
rride
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((nome == null) ? 0 : nome.hashCode
    result = prime * result + numeroMatricula;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Aluno other = (Aluno) obj;
    if (nome == null) {
        if (other.nome != null)
            return false;
    } else if (!nome.equals(other.nome))
        return false;
    if (numeroMatricula != other.numeroMatricula)
        return false;
    return true;
}
```

COPIAR CÓDIGO

Transcrição

Nós vimos no curso, duas implementações de `List`, o `ArrayList` e o `LinkedList`. Para representar os alunos do `Curso`, utilizamos o `HashSet`, uma das implementações de `Set`, que como foi dito, não garante a ordem dos elementos conforme estes sejam adicionados no conjunto.

Outras implementações de Set

Mas existe uma implementação de `Set` que guarda a ordem em que os elementos foram adicionados, o `LinkedHashSet`. Podemos testá-lo, utilizando-o na classe `Curso`:

```
private Set<Aluno> alunos = new LinkedHashSet<>();
```

COPIAR CÓDIGO

E executando a classe `TestaCursoComAluno`. Repare no resultado da impressão dos alunos:

```
Todos os alunos matriculados:
[Aluno: Rodrigo Turini, matricula: 34672]
[Aluno: Guilherme Silveira, matricula: 5617]
[Aluno: Mauricio Aniche, matricula: 17645]
```

COPIAR CÓDIGO

Os alunos foram impressos exatamente na mesma ordem em que foram adicionados no `Set`, diferentemente do `HashSet`.

Então há implementações de conjuntos que guardam a ordem que os elementos foram adicionados. Você pode pensar: "Mas foi falado que a ordem não é preservada". E ela não é! O `Set` não guarda a ordem dos elementos, nem tem como acessá-la, a própria interface diz isso. Por isso, mesmo utilizando `LinkedHashSet`, não conseguimos invocar o método `get`, mas na hora de percorrer o conjunto com um `foreach` os seus elementos virão na ordem em que foram adicionados.

Existe ainda uma outra implementação de `Set`, chamada `TreeSet`:

```
private Set<Aluno> alunos = new TreeSet<>();
```

COPIAR CÓDIGO

Mas se executarmos a classe `TestaCursoComAluno`, receberemos uma *exception*. Isso acontece porque o `TreeSet` só funciona para classes que implementam a interface `Comparable`, porque internamente o `TreeSet` guarda os objetos na sua ordem natural, que é a ordem implementada por meio do `Comparable`, mas se olharmos a classe `Aluno`, veremos que ela não implementa essa interface, por isso a exceção é lançada.

Vamos voltar a utilizar o `HashSet`, o importante aqui é saber que as coleções possuem várias implementações, tanto as listas quanto os `Set`s.

Código legado de coleções

Como o `Set` não possui um método `get`, vimos como acessar seus elementos usando o `foreach` do Java 8:

```
javaColecoes.getAlunos().forEach(aluno -> {
    System.out.println(aluno);
});
```

COPIAR CÓDIGO

E antes do Java 8? Tinha o outro `for`, que também vimos no curso:

```
for (Aluno aluno : javaColecoes.getAlunos()) {
    System.out.println(aluno);
}
```

COPIAR CÓDIGO

Só que esse `for` existe desde o Java 5, então como se acessava os elementos de um `Set` antes do Java 5? Era utilizado um objeto bem antigo, o `Iterator`. Ele é um objeto que todas as coleções nos dão acesso, que serve para **iterar** entre os elementos dentro da coleção, selecionando sempre o próximo objeto da coleção.

E a ordem? Se for uma lista, o `Iterator` selecionará os elementos na ordem em que estiverem nela, mas essa ordem não estará garantida no `Set`.

Então vamos mostrar agora como utilizar o `Iterator`, já que algum dia vocês poderão se deparar com um código legado que o utilize. Ainda no nosso exemplo, vamos pegar o `Set` de alunos:

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

COPIAR CÓDIGO

Como foi falado, toda coleção possui `Iterator`, podemos pegá-lo usando o método de mesmo nome:

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

COPIAR CÓDIGO

Com o iterador em mãos, existem dois métodos que costumamos usar. O primeiro é o método `hasNext`, que devolve um booleano dizendo se há ou não um próximo elemento na coleção. Então a primeira pergunta que sempre fazemos para o iterador é: "tem um próximo elemento na coleção?". Até porque se não houver um próximo elemento, não iremos querer pegá-lo. O segundo método é o `next`, que justamente devolve o próximo elemento.

Normalmente colocamos isso dentro de um `while`:

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

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

COPIAR CÓDIGO

Enquanto a coleção tem um próximo elemento, nós vamos imprimindo-o.

Então é muito comum acharmos um código legado que utiliza esse `Iterator`, por isso é importante conhecê-lo.

Se precisarmos percorrer mais uma vez a coleção, precisaríamos ir na coleção e pedir um novo `Iterator`, com o método `iterator`, como fizemos anteriormente.

Um outro objeto antigo que pode ser citado é o `Vector`, que era utilizado antes da interface `Collection` existir (`Collection` existe desde o Java 1.2):

```
Vector<Aluno> vetor = new Vector<>();
```

COPIAR CÓDIGO

Essa classe é muito antiga e se parece com o `ArrayList`, inclusive ela implementa `List` atualmente. A diferença é que ela pode ser utilizada por várias *threads* simultaneamente, chamado de *thread safe*. Não vamos entrar em muitos detalhes, mas o que pode ser dito é que todas as coleções faladas aqui até agora não são seguras quando utilizadas em várias threads, simultaneamente. Isso quer dizer que invocar vários métodos `add`, `get`, etc, pode acarretar algo que não esperamos, como que um "atropele" o outro, elementos sumirem, *exceptions*... A `Vector` não, ela sempre funciona, mesmo assim não é recomendada a sua utilização, já que existem outras formas de se trabalhar com coleções de maneira *thread safe*.

O que aprendemos neste capítulo:

- As implementações `LinkedHashSet` e `TreeSet`.
- Iteração de uma coleção utilizando o `Iterator`.
- A antiga classe `Vector`.

TreeSet e Comparator

Temos a seguinte classe:

```
public class Funcionario {

    private String nome;
    private int idade;

    public Funcionario(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    public String getNome() {
        return this.nome;
    }

    public void setNome(String nome) {

        this.nome = nome;

    }

    public int getIdade() {
        return this.idade;
    }
```

```
    public void setIdade(int idade) {
        this.idade = idade;
    }
}
```

COPIAR CÓDIGO

Crie uma classe que implemente a interface `Comparator` usando como critério de comparação a idade de funcionários. Crie três funcionários adicionando-os em um `TreeSet`. Em seguida, itere sobre a lista usando um `Iterator` para imprimir o nome de cada funcionário.

```
public class OrdenaPorIdade implements Comparator<Funcionario>{
    // metodo compare
}

public class Teste {

    public static void main(String args[]) {

        Funcionario f1 = new Funcionario("Barney", 12);
        Funcionario f2 = new Funcionario("Jonatan", 9);
        Funcionario f3 = new Funcionario("Guaraciara", 13);

        Set<Funcionario> funcionarios = new TreeSet<>(new Ordena
        funcionarios.add(f1);
        funcionarios.add(f2);
        funcionarios.add(f3);

        Iterator<Funcionario> iterador = funcionarios.iterator()
```

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

    public class Funcionario {

        private String nome;
        private int idade;

        public Funcionario(String nome, int idade) {
            this.nome = nome;
            this.idade = idade;
        }

        public String getNome() {
            return this.nome;
        }

        public void setNome(String nome) {

            this.nome = nome;

        }

        public int getIdade() {
            return this.idade;
        }

        public void setIdade(int idade) {
            this.idade = idade;
```

```
    }
}
```

COPIAR CÓDIGO

Opinião do instrutor

Uma implementação possível é:

Criando o comparator:

```
import java.util.Comparator;

public class OrdenaPorIdade implements Comparator<Funcionario>{

    @Override
    public int compare(Funcionario funcionario, Funcionario outr
        return funcionario.getIdade() - outroFuncionario.getIdac
    }

}
```

COPIAR CÓDIGO

Implementação o que foi pedido pelo exercício!

```java
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class Teste {

    public static void main(String args[]) {

        Funcionario f1 = new Funcionario("Barney", 12);
        Funcionario f2 = new Funcionario("Jonatan", 9);
        Funcionario f3 = new Funcionario("Guaraciara", 13);

        Set<Funcionario> funcionarios = new TreeSet<>(new OrdenaI
        funcionarios.add(f1);
        funcionarios.add(f2);
        funcionarios.add(f3);

        Iterator<Funcionario> iterador = funcionarios.iterator()

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

COPIAR CÓDIGO

77%

ATIVIDADES
7 de 7

FÓRUM DO CURSO

VOLTAR PARA DASHBOARD

MODO NOTURNO

ABRIR CADERNO

L

75.5k xp

a

Transcrição

Começando daqui? Você pode fazer o download (https://github.com/alura-cursos/java-collections/archive/aula9.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

No decorrer do curso, vocês podem ter tido a seguinte dúvida: O que é uma `Collection` para o Java?

Uma coleção é todo mundo que implementa a interface `Collection`. É ela que possui os métodos que utilizamos durante o treinamento, como `add`, `contains`, `remove` e `size`. Esses quatro métodos são os principais dessa interface e foram os que mais utilizamos.

Todas as classes que forem "filhas", ou implementação de `Collection`. Quais *collections* que vimos até agora? Duas interfaces "filhas", `List` e `Set`. Mas vimos outras implementações também, em especial o `ArrayList` e o `HashSet`.

Então uma `Collection` é uma interface que define métodos e trabalha com uma coleção, com um punhado de objetos. E existem várias formas de trabalhar com um punhado de objetos, como uma forma que pode ter objetos repetidos e estarão em uma sequência de ordem que nós definirmos, que é a `List`, ou onde não pode haver objetos repetidos e que não sabemos, independentemente da ordem dos objetos, que é o `Set`, entre outras formas.

Qual vamos usar? Depende da necessidade de cada um.

Para demonstrar que varia para cada caso. Para representar as aulas da Alura, fazia sentido usarmos uma lista para termos uma ordem, saber qual viria antes ou depois. Já para os alunos, não precisaremos saber qual virá antes ou depois, e nem poderemos ter elementos repetidos, então utilizamos um conjunto, um `Set`.

Mas se você ainda não sabe qual *collection* utilizar, pode-se declarar o atributo como `Collection`. Vamos criar uma classe chamada `QualColecaoUsar`, para demonstrar isso:

```
public class QualColecaoUsar {

    public static void main(String[] args) {

        Collection<Aluno> alunos;
    }
}
```

COPIAR CÓDIGO

Ainda não demos `new`, porque ainda não sabemos em quem queremos dar `new`. Em quem podemos dar `new`? Em todas as implementações que vimos! Pois todos são "filhos", "netos", e assim por diante, de `Collection`:

```
public class QualColecaoUsar {

    public static void main(String[] args) {

        Collection<Aluno> alunos = new ArrayList<>();
    }
}
```

COPIAR CÓDIGO

Com isso, nós conseguimos ainda utilizar os métodos `add`, `remove`, `size` ... Mas não conseguimos utilizar o `get`, porque ele não pertence à interface `Collection`. Entra aí a questão da necessidade. Se precisamos do método `get`, não faz muito sentido utilizarmos a interface `Collection` e sim a interface `List`.

E isso vai ficando um pouco mais claro com o tempo, no começo queremos utilizar `ArrayList` em tudo e com o tempo podemos ver que em alguns casos poderíamos ter utilizado o `HashSet`, que possui a vantagem de performance quando pesquisarmos elementos nele. E se não sabe ainda o que quer, declare simplesmente como `Collection`, conforme suas necessidades você vai definindo qual interface utilizar.

Mesmo na classe utilitária `Collections`, alguns dos seus métodos recebem `Collection` por parâmetro, para ser o mais genérico possível.

O que aprendemos neste capítulo:

- Declaração de atributos utilizando a interface `Collection`.

A implementações de Collection

Quais dos códigos a seguir estão corretos?

A

```
Collection<Aluno> alunos = new List<>();
```

uma vez que apesar de `List` estender `Collection`, não podemos instanciar um objeto de uma interface;

```
List<Aluno> alunos = new ArrayList<>();
```

`ArrayList` é uma classe que implementa `List`, logo pode ser referenciada pela sua interface

C

```
List<Aluno> alunos = new Collection<>();
```

Há dois erros. O primeiro é que não podemos instanciar uma interface desta maneira. O segundo é que `List` estende `Collection`, e não o contrário.

```
ArrayList<Aluno> alunos = new ArrayList<>();
```

Está correto, porém não está se aproveitando do famoso polimorfismo

```
Collection<Aluno> alunos = new HashSet<>();
```

Está correto, e usando bastante seu polimorfismo, declarando o atributo com a "avó", a interface `Collection`;

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);
    }
}
```

COPIAR CÓDIGO

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);
```

COPIAR CÓDIGO

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");
}
```

COPIAR CÓDIGO

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
}
```

COPIAR 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);
}
```

COPIAR CÓDIGO

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);
}
```

COPIAR CÓDIGO

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);
```

COPIAR CÓDIGO

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`.

08

Como continuar?

O grande foco desse treinamento foi o uso da API de Collections, mas a viagem dentro da plataforma não para por aqui.

Se você tiver interesse em aprender mais sobre o mundo OO e design de classes, os cursos sobre SOLID e padrões de projetos podem ser interessantes para você:

- OO: Melhores técnicas com Java (https://cursos.alura.com.br/course/orientacao-objetos-java)
- SOLID com Java (https://cursos.alura.com.br/course/orientacao-a-objetos-avancada-e-principios-solid)
- Design Patterns Java I (https://cursos.alura.com.br/course/design-patterns)
- Design Patterns Java II (https://cursos.alura.com.br/course/design-patterns-2)

Ou ainda, deseja aprofundar o conhecimento nas estruturas de dados abordadas no decorrer do curso:

- Estrutura de Dados (https://cursos.alura.com.br/course/estrutura-de-dados/)

Para quem quer entender melhor como funcionam as collections *thread-safe* e threads em geral, sugiro os seguintes cursos:

- Threads I (https://cursos.alura.com.br/course/threads-java-1)
- Threads (https://cursos.alura.com.br/course/threads-java-2)

E claro, tem muito mais, por exemplo a API de Reflection (https://cursos.alura.com.br/course/java-reflection-meta-programacao) ou treinamentos sobre o mundo Java Web e backend em geral.