Transcrição

As tarefas mais comuns de um programador podem ser desafiadoras no Java. O motivo? A sintaxe com mais de 20 anos, tornando-a uma linguagem burocrática. Felizmente isso mudou significativamente com o Java 8. Um exemplo? Ordenação de objetos.

Vamos usar o Eclipse nos nossos exemplos. Você pode utilizar qualquer outra IDE, ou até mesmo a linha de comando. O antigo Eclipse Kepler 4.3 possui suporte ao Java 8 via download, a partir do Eclipse Luna 4.4, esse suporte já vem ativado. Lembre-se de verificar se você tem o Java 8 instalado, indo ao console/terminal e digitando `java -version`. Deve sair `1.8.0`. Caso contrário, atualize a versão do seu java (http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html?ssSourceSiteId=otnes).

No eclipse, crie um novo projeto chamado `Java8` e, através das propriedades do projeto, escolha a opção Java Compiler e ative a versão 8. Se ela não está disponível e você tem certeza que instalou o Java 8, basta adicionar esse JDK em Windows, Preferences, Java, Installed JREs.

Em uma nova classe `OrdenaStrings`, crie o método `main` e vamos fazer uma lista de strings e trabalhar com ele sem nenhum dos novos recursos da linguagem:

```
List<String> palavras = new ArrayList<>();
palavras.add("alura online");
palavras.add("casa do código");
palavras.add("caelum");
```

COPIAR CÓDIGO

Vale lembrar que podemos criar uma lista de objetos diretamente via `Arrays.asList`, fazendo `List<String> palavras = Arrays.asList("", "", ...)`. A diferença é que não se pode mudar a quantidade de elementos de uma lista devolvida por esse método.

Como fazemos para ordenar essa lista? Podemos fazer isso sem usar nenhuma novidade: com o `Collections.sort`:

```
Collections.sort(palavras);
System.out.println(palavras);
```

COPIAR CÓDIGO

E se quisermos ordenar essas palavras em uma ordem diferente? Por exemplo, pela ordem do tamanho das palavras. Nesse caso, utilizaremos um `Comparator`. Podemos criá-la como uma outra classe, por enquanto apenas o esqueleto:

```
class ComparadorDeStringPorTamanho implements Comparator<String> {

    public int compare(String s1, String s2) {
        return 0;
    }

}
```

COPIAR CÓDIGO

O que preencher aí dentro? Se você lembrar, o contrato da interface `Comparator` diz que devemos devolver um número negativo se o primeiro objeto for menor que o segundo, um número positivo caso contrário e zero se forem equivalentes.

Esse "maior", "menor" e "equivalente" é algo que nós decidimos. No nosso caso, vamos dizer que uma `String` é "menor" que outra se ela tiver menos caracteres. Então podemos fazer:

```
class ComparadorDeStringPorTamanho implements Comparator<String> {
    public int compare(String s1, String s2) {
        if(s1.length() < s2.length())
            return -1;
        if(s1.length() > s2.length())
            return 1;
        return 0;
    }
}
```

COPIAR CÓDIGO

E, para ordenar com esse novo critério de comparação, podemos fazer:

```
Comparator<String> comparador = new ComparadorDeStringPorTamanho();
Collections.sort(palavras, comparador);
```

COPIAR CÓDIGO

Até aqui, nenhuma novidade. No decorrer do curso, você verá como esse código ficará muito, muito menor, mais sucinto e expressivo com cada recurso que formos estudar do Java 8. Vamos ao primeiro deles. Em vez de usar o `Collections.sort`, podemos invocar essa operação na própria `List`! Veja:

```
Comparator<String> comparador = new ComparadorDeStringPorTamanho();
palavras.sort(comparador);
```

COPIAR CÓDIGO

Parece pouco, mas há muita coisa por trás. Em primeiro lugar, esse método `sort` não existia antes na interface `List` (http://docs.oracle.com/javase/8/docs/api/java/util/List.html), nem em suas mães (`Collection` e `Iterable`).

Será então que simplesmente adicionaram um novo método? Se tivessem feito assim, haveria um grande problema: todas as classes que implementam `List` parariam de compilar, pois não teriam o método `sort` . E há muitas, muitas classes que implementam essas interfaces básicas do Java. Há implementações no Hibernate, no Google Collections e muito mais.

Default Methods

Para evitar essa quebra, o Java 8 optou por criar um novo recurso que possibilitasse adicionar métodos em interfaces e implementá-los ali mesmo! Se você abrir o código fonte da interface `List` , verá esse método:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

COPIAR CÓDIGO

É um default method! Um método de interface que você não precisa implementar na sua classe se não quiser, pois você terá já essa implementação default. Repare que ele simplesmente delega a invocação para o bom e velho `Collections.sort` , mas veremos que outros métodos fazem muito mais.

Default methods foi uma forma que o Java encontrou para evoluir interfaces antigas, sem gerar incompatibilidades. Não é uma novidade da linguagem: Scala, C# e outras possuem recursos similares e até mais poderosos. E repare que é diferente de uma classe abstrata: em uma interface você não pode ter atributos de instância, apenas esses métodos que delegam chamadas ou trabalham com os próprios métodos da interface.

foreach, Consumer e interfaces no java.util.functions

Vamos a um outro método default adicionado as coleções do Java: o `forEach` na interface `Iterable`. Como `Iterable` é mãe de `Collection`, temos acesso a esse método na nossa lista.

Se você abrir o JavaDoc ou utilizar o auto complete do Eclipse, verá que `List.forEach` recebe um `Consumer`, que é uma das muitas interfaces do novo pacote `java.util.functions`. Então vamos criar um consumidor de `String`:

```
class ConsumidorDeString implements Consumer<String> {
    public void accept(String s) {
        System.out.println(s);
    }
}
```

COPIAR CÓDIGO

E podemos passar uma instância dessa para o `forEach`:

```
Consumer<String> consumidor = new ConsumidorDeString();
palavras.forEach(consumidor);
```

COPIAR CÓDIGO

Interessante? Ainda não muito. Talvez fosse mais direto e simples escrever um `for(String s : lista)`.

Default methods é o primeiro recurso que conhecemos. Sim, é bastante simples e parece não trazer grandes melhorias. O segredo é utilizá-los junto com lambdas, que você verá a seguir, e trará um impacto significativo para o seu código.

Quais são os novos métodos default em List?

Pesquise os novos métodos `default` adicionados na interface `List`.

Você pode consultar a documentação neste link (http://docs.oracle.com/javase/8/docs/api/java/util/List.html).

Além do `forEach` na interface `Iterable` e `sort` na interface `List`, quais outros você achou interessante?

Opinião do instrutor

Uma adição também interessante foi o `replaceAll`.

Isso sem considerar os diversos outros que `List` herda de `Collection`.

Vantagens dos default methods

O que você achou dessa introdução do Java, os métodos `default` ?

Qual você acha que é a grande vantagem desses métodos? O que eles possibilitam?

Opinião do instrutor

Sem dúvidas um recurso bem interessante, veremos mais pra frente o quanto as novas APIs tiram proveito dele.

A grande vantagem é que agora uma interface pode evoluir sem quebrar compatibilidade.

Transcrição

Vamos retomar o nosso `forEach`, ele precisa da classe que implementa `Consumer`:

```
class ConsumidorDeString implements Consumer<String> {
    public void accept(String s) {
        System.out.println(s);
    }
}
```

COPIAR CÓDIGO

E a invocação:

```
Consumer<String> consumidor = new ConsumidorDeString();
palavras.forEach(consumidor);
```

COPIAR CÓDIGO

Se você já está acostumado com Java há mais tempo, sabe que nesses casos não criamos uma classe isolada. Fazemos tudo ao mesmo tempo, criando a classe e instanciando-a:

```
Consumer<String> consumidor = new Consumer<String>() {
    public void accept(String s) {
```

```
            System.out.println(s);
        }
    };
    palavras.forEach(consumidor);
```

COPIAR CÓDIGO

São as chamadas classes anônimas, que usamos com frequência para implementar listeners e callbacks que não terão reaproveitamento.

Poderíamos até mesmo evitar a criação da variável `consumidor`, passando a classe anônima diretamente para o `forEach`:

```
palavras.forEach(new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s);
    }
});
```

COPIAR CÓDIGO

Quando começamos a aprender Java, essa sintaxe pode intimidar. Ela aparece com frequência, em especial nesses casos onde a implementação é curta e simples.

Lambda para simplificar

Tendo essas dificuldade e verbosidade da sintaxe das classes anônimas em vista, o Java 8 traz uma nova forma de implementar essas interfaces ainda mais sucinta. É a sintaxe do lambda. Em vez de escrever a classe anônima, deixamos de escrever alguns itens que podem ser inferidos.

Como essa interface só tem um método, não precisamos escrever o nome do método. Também não daremos new. Apenas declararemos os argumentos e o bloco a ser executado, separados por `->` :

```
palavras.forEach((String s) -> {
    System.out.println(s);
});
```

COPIAR CÓDIGO

É uma forma bem mais sucinta de escrever! Essa sintaxe funciona para qualquer interface que tenha apenas um método abstrato, e é por esse motivo que nem precisamos falar que estamos implementando o método `accept` , já que não há outra possibilidade. Podemos ir além e remover a declaração do tipo do parâmetro, que o compilador também infere:

```
palavras.forEach((s) -> {
    System.out.println(s);
});
```

COPIAR CÓDIGO

Quando há apenas um parâmetro, nem mesmo os parenteses são necessários:

```
palavras.forEach(s -> {
    System.out.println(s);
});
```

COPIAR CÓDIGO

Dá pra melhorar? Sim. podemos remover as chaves de declaração do bloco, assim como o ponto e vírgula, pois só existe uma única instrução:

```
palavras.forEach(s -> System.out.println(s));
```

COPIAR CÓDIGO

Pronto. Em vez de usarmos classes anônimas, utilizamos o lambda para escrever códigos simples e sucintos nesses casos. Uma interface que possui apenas um método abstrato é agora conhecida como interface funcional e pode ser utilizada dessa forma.

Outro exemplo é o próprio `Comparator` que já vimos. Se utilizarmos a forma de classe anônima, teremos essa situação:

```
palavras.sort(new Comparator<String>() {
    public int compare(String s1, String s2) {
        if (s1.length() < s2.length())
            return -1;
        if (s1.length() > s2.length())
            return 1;
        return 0;
    }
});
```

COPIAR CÓDIGO

Como aplicar a mesma lógica para transformar isso em um lambda? Basta removermos quase tudo da assinatura do método, assim como o `new Comparator` e adicionar o `->` entre os parâmetros e o bloco. Além disso, podemos tirar o tipo dos parâmetros:

```
palavras.sort((s1, s2) -> {
    if (s1.length() < s2.length())
        return -1;
    if (s1.length() > s2.length())
```

```
            return 1;
        return 0;
    });
```

COPIAR CÓDIGO

Melhor? Parece que sim. Mas ainda não muito interessante. O lambda se encaixa melhor quando a expressão dentro do bloco é mais curta. Normalmente com apenas um statement. Conhecendo a API do Java, podemos ver que há um método que compara dois inteiros e retorna negativo/positivo/zero dependendo se o primeiro for menor/maior/igual ao segundo. É o `Integer.compare` . Com ele, reduzimos o lambda para o seguinte:

```
palavras.sort((s1, s2) -> {
    return Integer.compare(s1.length(), s2.length());
});
```

COPIAR CÓDIGO

Dá para fazer melhor. Como há apenas um único statement, podemos remover as chaves. Além disso, o `return` pode ser eliminado que o compilador vai inferir que deve ser retornado o valor que o próprio `compare` devolver:

```
palavras.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

COPIAR CÓDIGO

Compare com a nossa primeira versão. Muito melhor! Claro que poderíamos ter utilizado o `Integer.compare` desde o capítulo anterior, mas a combinação com o lambda deixa tudo mais legível e simples.

Vale lembrar que não é porque digitamos menos linhas que o código é necessariamente mais simples. Às vezes, pouco código pode tornar difícil de entender uma ideia, um algoritmo. Não é o nosso caso.

Threads com lambda!

Considere o seguinte código que executa um `Runnable` em uma `Thread`:

```
new Thread(new Runnable() {

    @Override
    public void run() {
        System.out.println("Executando um Runnable");
    }

}).start();
```

COPIAR CÓDIGO

Como podemos escrevê-lo usando uma expressão lambda?

Opinião do instrutor

Seu código deve ficar parecido com:

```
new Thread(() -> System.out.println("Executando um Runnable")).start()
```

COPIAR CÓDIGO

Transcrição

Com os lambdas e métodos default, conseguimos escrever a ordenação das Strings de uma forma bem mais sucinta:

```
palavras.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

COPIAR CÓDIGO

Podemos ir além.

Métodos default em Comparator

Há vários métodos auxiliares no Java 8. Até em interfaces como o `Comparator`. E você pode ter um método default que é estático. Esse é o caso do `Comparator.comparing`, que é uma fábrica, uma factory, de `Comparator`. Passamos o lambda para dizer qual será o critério de comparação desse `Comparator`, repare:

```
palavras.sort(Comparator.comparing(s -> s.length()));
```

COPIAR CÓDIGO

Veja a expressividade da linha, está escrito algo como "palavras ordene comparando s.length". Podemos quebrar em duas linhas para ver o que esse novo método faz exatamente:

```
Comparator<String> comparador = Comparator.comparing(s -> s.length());
palavras.sort(comparador);
```

COPIAR CÓDIGO

Dizemos que `Comparator.comparing` recebe um lambda, mas essa é uma expressão do dia a dia. Na verdade, ela recebe uma instância de uma interface funcional. No caso é a interface `Function` (http://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) que tem apenas um método, o `apply`. Para utilizarmos o `Comparator.comparing`, nem precisamos ficar decorando os tipos e assinatura do método dessas interfaces funcionais. Essa é uma vantagem dos lambdas. Você também vai acabar programando dessa forma. É claro que, com o tempo, você vai conhecer melhor as funções do pacote `java.util.functions`. Vamos quebrar o código mais um pouco. Não se esqueça de dar os devidos `imports`.

```
Function<String, Integer> funcao = s -> s.length();
Comparator<String> comparador = Comparator.comparing(funcao);
palavras.sort(comparador);
```

COPIAR CÓDIGO

A interface `Function` vai nos ajudar a passar um objeto para o `Comparator.comparing` que diz qual será a informação que queremos usar como critério de comparação. Ela recebe dois tipos genéricos. No nosso caso, recebe uma `String`, que é o tipo que queremos comparar, e um `Integer`, que é o que queremos extrair dessa string para usar como critério. Poderia até mesmo criar uma classe anônima para implementar essa `Function` e seu método `apply`, sem utilizar nenhum lambda. O código ficaria grande e tedioso.

Quisemos quebrar em três linhas para que você pudesse enxergar o que ocorre por trás exatamente. Sem dúvida o `palavras.sort(Comparator.comparing(s -> s.length()))` é mais fácil de ler. Dá para melhorar ainda mais? Sim!

Method reference

É muito comum escrevermos lambdas curtos, que simplesmente invocam um único método. É o exemplo de `s -> s.length()`. Dada uma `String`, invoque e retorne o método `length`. Por esse motivo, há uma forma de escrever esse tipo de lambda de uma forma ainda mais reduzida. Em vez de fazer:

```
palavras.sort(Comparator.comparing(s -> s.length()));
```

COPIAR CÓDIGO

Fazemos uma referência ao método (method reference):

```
palavras.sort(Comparator.comparing(String::length));
```

COPIAR CÓDIGO

São equivalentes nesse caso! Sim, é estranho ver `String::length` e dizer que é equivalente a um lambda, pois não há nem a `->` e nem os parênteses de invocação ao método. Por isso é chamado de method reference. Ela pode ficar ainda mais curta com o `import static`:

```
import static java.util.Comparator.*;
palavras.sort(comparing(String::length));
```

COPIAR CÓDIGO

Vamos ver melhor a semelhança entre um lambda e seu method reference equivalente. Veja as duas declarações a seguir:

```
Function<String, Integer> funcao1 = s -> s.length();
Function<String, Integer> funcao2 = String::length;
```

COPIAR CÓDIGO

Elas ambas geram a mesma função: dada um `String`, invoca o método `length` e devolve este `Integer`. As duas serão avaliadas/resolvidas (*evaluated*) para `Functions` equivalentes.

Quer um outro exemplo? Vejamos o nosso `forEach`, que recebe um `Consumer`:

```
palavras.forEach(s -> System.out.println(s));
```

COPIAR CÓDIGO

Dada uma `String`, invoque o `System.out.println` passando-a como argumento. É possível usar method reference aqui também! Queremos invocar o `println` de `System.out`:

```
palavras.forEach(System.out::println);
```

COPIAR CÓDIGO

Novamente pode parecer estranho. Não há os parênteses, não há a flechinha (`->`), nem os argumentos que o `Consumer` recebe. Fica tudo implícito. Dessa vez, o argumento recebido (isso é, cada palavra dentro da lista `palavras`), não será a variável onde o método será invocado. O Java 8 consegue perceber que tem um `println` que recebe objetos, e invocará esse método, passando a `String` da vez.

Quando usar lambda e quando usar method reference? Algumas vezes não é possível usar method references. Se você tiver, por exemplo, um lambda que dada uma `String`, pega os 5 primeiros caracteres, faríamos `s -> s.substring(0, 5)`. Esse lambda não pode ser escrito como method reference! Pois não é uma simples invocação de métodos onde os parâmetros são os mesmos que os do lambda.

Mudando o critério de comparação

Mude o seu comparator usando algum outro critério de comparação no lugar do tamanho(`length`) da `String` .

Qual foi o critério escolhido?

Opinião do instrutor

Uma forma seria utilizar o `String.CASE_INSENSITIVE_ORDER` :

```
palavras.sort(String.CASE_INSENSITIVE_ORDER);
```

COPIAR CÓDIGO

Transcrição

Para que a gente possa elaborar exemplos mais reais e interessantes, vamos criar uma simples classe que representa um curso do Alura. Com ela, vamos refazer alguns passos dos últimos capítulos do curso, de forma mais rápida, a fim de revê-los. Ela terá apenas dois atributos: seu nome e a quantidade de alunos, além de um construtor e seus respectivos getters:

```
class Curso {
    private String nome;
    private int alunos;

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

    public String getNome() {
        return nome;
    }

    public int getAlunos() {
        return alunos;
    }
}
```

COPIAR CÓDIGO

Em uma classe com o `main`, criamos alguns cursos e inserimos em uma lista:

```
List<Curso> cursos = new ArrayList<Curso>();
cursos.add(new Curso("Python", 45));
cursos.add(new Curso("JavaScript", 150));
cursos.add(new Curso("Java 8", 113));
cursos.add(new Curso("C", 55));
```

COPIAR CÓDIGO

Se quisermos ordenar esses cursos pela quantidade de alunos, podemos utilizar o `Comparator` da forma antiga, fazendo `lista.sort(new Comparator<Curso>() { .... }` e implementando nosso critério de comparação dentro da classe anônima.

E a forma nova, como fazemos? Podemos já utilizar o `Comparator.comparing`. E indicaremos que queremos que o `getAlunos` seja utilizado como critério de comparação:

```
cursos.sort(Comparator.comparingInt(c -> c.getAlunos()));
```

COPIAR CÓDIGO

Esse é o caso que podemos usar um method reference para ficar ainda mais sucinto e legível:

```
cursos.sort(Comparator.comparingInt(Curso::getAlunos));
```

COPIAR CÓDIGO

Pronto! Você já pode fazer um `forEach` e imprimir os nomes dos cursos para ver se o resultado é o esperado.

Streams: trabalhando com coleções no java 8

E se quisermos fazer outras tarefas com essa coleção de cursos? Por exemplo, filtrar apenas os cursos com mais de 100 alunos. Poderíamos fazer um loop que, dado o critério desejado seja atendido, adicionamos este curso em uma nova lista, a lista filtrada.

No Java 8, podemos fazer de uma forma muito mais interessante. Há como invocar um `filter`. Para sua surpresa, esse método não se encontra em `List`, nem em `Collection`, nem em nenhuma das interfaces já conhecidas. Ela está dentro de uma nova interface, a `Stream`. Você pode pegar um `Stream` de uma coleção simplesmente invocando `cursos.stream()`:

```
Stream<Curso> streamDeCurso = cursos.stream();
```

COPIAR CÓDIGO

O que fazemos com ele? O `Stream` devolvido por esse método tem uma dezena de métodos bastante úteis. O primeiro é o `filter`, que recebe um predicado (um critério), que deve devolver verdadeiro ou falso, dependendo se você deseja filtrá-lo ou não. Utilizaremos um lambda para isso:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
```

COPIAR CÓDIGO

Repare que o filtro devolve também um `Stream`! É um exemplo do que chamam de `fluent interface`. Vamos fazer um `forEach` e ver o resultado dos cursos:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
cursos.forEach(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

A saída será:

```
Python
C
Java 8
Java Script
```

COPIAR CÓDIGO

Estranho. Filtramos apenas os que tem mais de 100 alunos, e ele acabou listando todos! Por quê? Pois **modificações em um stream não modificam a coleção/objeto que o gerou**. Tudo que é feito nesse fluxo de objetos, nesse `Stream`, não impacta, não tem efeitos colaterais na coleção original. A coleção original continua com os mesmos cursos!

Para imprimir os cursos filtrados, podemos usar o `forEach` que existe em `Stream`:

```
Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);
streamDeCurso.forEach(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

Ou melhor ainda, podemos eliminar essa variável temporária, fazendo tudo em uma mesma linha:

```
cursos.stream().filter(c -> c.getAlunos() > 100).forEach(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

Por uma questão de legibilidade, vamos espaçar esse código em algumas linhas, mas continuando um único `statement`:

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .forEach(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

Interessante não? E por que criaram uma nova interface e não colocar o método de filtrar dentro de `List`? Há vários motivos. Mas repare já em um primeiro: numa coleção tradicional, o que o `filter` faria? Alteraria a coleção em questão, ou manteria intacta, devolvendo uma nova coleção? Coleções no java podem ser mutáveis e imutáveis, o que complicaria ler esses métodos. No `Stream`, sabemos que esses métodos nunca alterarão a coleção original.

Vamos além. Vamos ver as outras funcionalidades. E se quisermos, dados esses cursos filtrados no nosso fluxo (`Stream`) de objetos, um novo fluxo apenas com a quantidade de alunos de cada um deles? Utilizamos o `map`:

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .map(c -> c.getAlunos());
```

COPIAR CÓDIGO

Se você reparar, esse `map` não devolve um `Stream<Curso>`, e sim um `Stream<Integer>`! Faz sentido. Podemos concatenar a invocação ao `forEach` para imprimirmos os dados:

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .map(c -> c.getAlunos())
   .forEach(x -> System.out.println(x));
```

COPIAR CÓDIGO

Aproveitamos para recapitular o que já vimos: temos a oportunidade de usar o recurso de method references duas vezes. Tanto pra invocação de `getAlunos` quanto a do `println`. Vamos alterar:

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .map(Curso::getAlunos)
   .forEach(System.out::println);
```

COPIAR CÓDIGO

O lambda passado para o `filter` não pode ser representado como um method reference, pois não é uma simples invocação de um único método: ele compara com um número. Pode ser que, no começo, você prefira os lambdas, pela sintaxe ser utilizada mais frequentemente. Com o tempo, verá que o method reference não impõe problema de legibilidade algum, muito pelo contrário.

Outro ponto que podemos notar: nem vimos qual é o tipo de interface que `Map` recebe! É uma `Function`, mas repare que usamos o lambda e nem foi necessário conhecer a fundo quais eram os parâmetros que ele recebia. Foi natural. É claro que, com o tempo, é importante que você domine essa nova API, mesmo que acabe utilizando majoritariamente as interfaces como lambdas.

Streams primitivos

Trabalhar com Streams vai ser frequente no seu dia a dia. Há um cuidado a ser tomado: com os tipos primitivos. Quando fizemos o `map(Curso::getAlunos)`, recebemos de volta um `Stream<Integer>`, que acaba fazendo o autoboxing dos `int`s. Isto é, utilizará mais recursos da JVM. Claro que, se sua coleção é pequena, o impacto será irrisório. Mas é possível trabalhar só com `int`s, invocando o método `mapToInt`:

```
IntStream stream = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .mapToInt(Curso::getAlunos);
```

COPIAR CÓDIGO

Ele devolve um `IntStream`, que não vai gerar autoboxing e possui novos métodos específicos para trabalhar com inteiros. Um exemplo? A soma:

```
int soma = cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .mapToInt(Curso::getAlunos)
    .sum()
```

COPIAR CÓDIGO

Em uma única linha de código, pegamos todos os cursos, filtramos os que tem mais de 100 e somamos todos os alunos. Há também versões para `double` e `long` de `Streams` primitivos. Até mesmo o `Comparator.comparing` possui versões como `Comparator.comparingInt`, que recebe uma `IntFunction` e não necessita do boxing. Em outras palavras, todas as interfaces funcionais do novo pacote `java.util.functions` possuem versões desses tipos primitivos.

`Stream` não é uma `List`, não é uma `Collection`. E se quisermos obter uma coleção depois do processamento de um `Stream`? É o que veremos no próximo capítulo.

Não deixe de praticar bastante, descobrir novos métodos e fazer os exercícios propostos!

Utilizando o método map

Como transformar o nosso `Stream<Curso>` em um `Stream<String>` contendo apenas os nomes dos cursos?

Opinião do instrutor

Podemos fazer essa projeção utilizando o método `map`:

```
Stream<String> nomes = cursos.stream().map(Curso::getNome);
```

COPIAR CÓDIGO

Mais sobre a API de Stream

Pesquise mais sobre a API de Stream em sua documentação:

http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
(http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)

O que achou da API? Que outros métodos você achou interessante?

Opinião do instrutor

Há diversos novos métodos, vamos conhecer mais alguns no proximo capitulo.

Alguns bem interessante são `map`, `collect`, `findFirst` e `findAny`.

Transcrição

Os Streams possibilitam trabalhar com dados de uma maneira funcional. Normalmente, são dados e objetos que vêm de uma collection do Java. Por que não adicionaram esses métodos diretamente nas Collections? Justo para não ser dependente delas, não ter efeitos colaterais e não entupir de métodos as interfaces.

Vamos conhecer outros métodos interessantes dos Streams. Um exemplo seria: quero um curso que tenha mais de 100 alunos! Pode ser qualquer um deles. Há o método `findAny`

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .findAny();
```

COPIAR CÓDIGO

O que será que devolve o `findAny` ? Um `Curso` ? Não! Um `Optional<Curso>` .

Optional

`Optional` é uma importante nova classe do Java 8. É com ele que poderemos trabalhar de uma maneira mais organizada com possíveis valores `null` . Em vez de ficar comparando `if(algumaCoisa == null)` , o `Optional` já fornece uma série de métodos para nos ajudar nessas situações. Por que o `findAny` utiliza esse recurso? Pois pode não haver nenhum curso com mais de 100 alunos! Nesse caso, o que seria retornado? `null` ? uma exception?

Vamos ver as vantagens de se trabalhar com `Optional`. Primeiro vamos atribuir o resultado do `findAny` a uma variável:

```
Optional<Curso> optional = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .findAny();
```

COPIAR CÓDIGO

Dado um `Optional`, podemos pegar seu conteúdo invocando o `get`. Ele vai devolver o `Curso` que queremos. Mas e se não houver nenhum? Uma exception será lançada.

```
Curso curso = optional.get();
```

COPIAR CÓDIGO

Há métodos mais interessantes. O `orElse` diz que ele deve devolver o curso que existe dentro desse optional, *ou então* o que foi passado como argumento:

```
Curso curso = optional.orElse(null);
```

COPIAR CÓDIGO

Nesse caso ou ele devolve o curso encontrado, ou `null`, caso nenhum seja encontrado. Mesmo assim, ainda não está tão interessante. Há como evitar tanto o `null`, quanto as exceptions, quanto os `ifs`. O método `ifPresent` executa um lambda (um `Consumer`) no caso de existir um curso dentro daquele optional:

```
optional.ifPresent(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

Claro que, no dia a dia, não teríamos a variável temporária `curso`. Podemos fazer isso

```
cursos.stream()
    .filter(c -> c.getAlunos() > 100)
    .findAny()
    .ifPresent(c -> System.out.println(c.getNome()));
```

COPIAR CÓDIGO

Outros métodos devolvem `Optional` nos `Streams`. Um deles é o `average` em `IntStream`. Por que? Pois pode não existir nenhum element, e aí a média poderia realizar uma divisão por zero.

Você vai encontrar Optional não somente na API de Streams. Vale a pena conhecer e utilizá-la no seu próprio código e entidades.

Gerando uma coleção a partir de um Stream

Invocar métodos no `stream` de uma coleção não altera o conteúdo da coleção original. Ele não gera efeitos colaterais. Como então obter uma coleção depois de alterar um `Stream`?

Tentar fazer `List<Curso> novaLista = lista.stream().filter(...)` não compila, pois um `Stream` **não é** uma coleção. Para fazer algo parecido com isso, utilizamos o método `collect`, que *coleta* elementos de um `Stream` para produzir um outro objeto, como uma coleção.

O método `Collect` recebe um `Collector`, uma interface não tão trivial de se implementar. Podemos usar a classe `Collectors` (repare o `s` no final), cheio de *factory methods* que ajudam na criação de coletores. Um dos coletores mais utilizados é o retornado por `Collectors.toList()`:

```
List<Curso> resultados = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .collect(Collectors.toList());
```

COPIAR CÓDIGO

Pronto! É através dos coletores que podemos "retornar" de um `Stream` para uma `Collection`. Certamente poderia ter usado a mesma variável, a `List<Curso> cursos` que temos:

```
cursos = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .collect(Collectors.toList());
```

COPIAR CÓDIGO

Pronto. Alteramos a referência antiga para apontar para essa nova coleção, depois de filtrada!

Um exemplo mais complicado? Podemos gerar mapas! Queremos um mapa que, dado o nome do curso, o valor atrelado é a quantidade alunos. Um `Map<String, Integer>`. Utilizamos o `Collectors.toMap`. Ele recebe duas `Functions`. A primeira indica o que vai ser a chave, e a segunda o que será o valor:

```
Map mapa = cursos
.stream()
.filter(c -> c.getAlunos() > 100)
.collect(Collectors.toMap(c -> c.getNome(), c -> c.getAlunos()));
```

COPIAR CÓDIGO

Outras vantagens do Stream

Os Streams foram desenhados de uma forma a tirar proveito da programação funcional. Se você utilizá-los da forma que vimos por aqui, eles nunca gerarão efeitos colaterais. Isso é, apenas o stream será alterado, e nenhum outro objeto será impactado.

Dada essa premissa, podemos pedir para que nosso `stream` seja processado em paralelo. Ele mesmo vai decidir quantas threads usar e fazer todo o trabalho, utilizando APIs mais complicadas (como a de fork join) para ganhar performance. Para fazer isso, basta utilizar `parallelStream()` em vez de `stream()` !

Tome cuidado. Para streams pequenos, o custo de cuidado dessas threads e manipular os dados entre elas é alto e pode ser bem mais lento que o `Stream` tradicional.

Não deixe de investigar a API de `Stream` e conhecer os outros métodos que ela possui. Certamente você vai parar de escrever os diversos `fors` encadeados que estamos acostumados, podendo fazer tudo de uma maneira mais legível, fácil e funcional.

Trabalhando com Optional

Ao utilizar o método `findFirst()` temos como retorno um `Optional<Curso>`. Por quê?

Qual a vantagem de retornar um `Optional` no lugar de retornar um curso diretamente?

Opinião do instrutor

Ganhamos muito com essa nova introdução. Assim não precisamos escrever aqueles diversos `if`s garantindo que o objeto não é nulo, temos uma forma muito mais interessante de representar nossas intenções. A classe `Optional` nos oferece uma variedade imensa de novos métodos que nos permite trabalhar de forma funcional com nossos valores, tirando maior proveito dos novos recursos de `default methods`, `lambda`s e `method reference`. Você pode ler mais sobre essa API em sua documentação:

http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html (http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)

Calculando média de quantidade de alunos

Calcule a quantidade média de alunos de todos os seus cursos utilizando a API de Stream.

Opinião do instrutor

Uma possível solução é:

```
cursos.stream()
    .mapToInt(c -> c.getAlunos())
    .average();
```

COPIAR CÓDIGO

Coletando o resultado do stream em uma List

Depois de filtrar todos os cursos com mais de 50 alunos, temos um `Stream<Curso>` como resultado:

```
Stream<Curso> stream = cursos.stream()
    .filter(c -> c.getAlunos() > 50);
```

COPIAR CÓDIGO

Como podemos transformar esse `Stream<Curso>` filtrado em uma `List<Curso>` ?

Opinião do instrutor

Para isso utilizamos o método `collect` , da seguinte forma:

```
List<Curso> cursosFiltrados = cursos.stream()
    .filter(c -> c.getAlunos() > 50)
    .collect(Collectors.toList());
```

COPIAR CÓDIGO

Transcrição

Já vimos bastante das principais novidades da linguagem. Vamos conhecer agora mais uma grande introdução do java 8, a nova API de datas. Essa é uma API bem grande, que fazia bastante falta e foi por muito tempo esperada pelos desenvolvedores.

Se você já trabalha com Java sabe as dificuldades em trabalhar com as classes `Date` e `Calendar`. Se você ainda não conhece essas classes, não se preocupe, você já pode começar a aprender e utilizar os modelos novos.

Pra começar vamos criar uma classe simples com um método `main`, onde vamos fazer os nossos testes com a nova APi.

```
public class Datas {

    public static void main(String[] args) {

    }
}
```

COPIAR CÓDIGO

Vamos começar criando uma data, a data de hoje. Para representar uma data em java agora eu posso utilizar a classe `LocalDate`, presente no pacote `java.time`. Repare como é facil ter a data atual utilizando o método `now()`:

```
public class Datas {

    public static void main(String[] args) {

        LocalDate hoje = LocalDate.now();
        System.out.println(hoje);

    }
}
```

COPIAR CÓDIGO

Assim como este existem diversos outros métodos estáticos nas novas classes da API de datas. Vamos conhecer vários no decorrer do capítulo.

Repare que o valor impresso neste caso será `2014-05-28`, logo veremos como formatar essa saída de outras formas.

Conhecendo mais da API

Vamos agora criar uma nova data para representar as Olimpíadas do Rio, por exemplo. Para fazer isso podemos utilizar o método `of` passando o dia, mês e ano:

```
LocalDate olimpiadasRio = LocalDate.of(2016, Month.JUNE, 5);
```

COPIAR CÓDIGO

Repare que existe uma enumeração pra representar o mês, mas se você preferir pode utilizar sim um numero inteiro.

Vamos agora calcular a diferença de anos entre essas duas datas. Uma forma de fazer isso na mão seria subtraindo o método `getYear` das datas, algo como:

```
int anos = olimpiadasRio.getYear() - hoje.getYear();
System.out.println(anos);
```

COPIAR CÓDIGO

Trabalhando com Period

Ao executar esse código temos o resultado esperado, que neste caso é 2 anos. Mas e se quisessemos descobrir a diferença de dias e meses também? Daria pra fazer da mesma forma, mas sempre que você tiver que fazer um trabalho dessa na mão você pode ter certeza que já existe algo pronto pra te ajudar de alguma forma. Nesse caso podemos utilizar a classe `Period`.

Para saber a diferença entre duas datas podemos utilizar seu método `between`, da seguinte forma:

```
Period periodo = Period.between(hoje, olimpiadasRio);
System.out.println(periodo);
```

COPIAR CÓDIGO

Repare que a saída desse nosso `println` vai ser um pouco estranha, um exemplo seria: `P2Y8D`.

Isso significa um período de 2 anos e 8 dias. Mas claro, poderíamos imprimir apenas as propriedades, da segunite forma:

```
Period periodo = Period.between(hoje, olimpiadasRio);
System.out.println(periodo.getYears());
System.out.println(periodo.getMonths());
System.out.println(periodo.getDays());
```

COPIAR CÓDIGO

Assim como esses existem diversos getters pra passar informações importantes a respeito desse período.

Incrementando e decrementando suas datas

Outra coisa bem comum em nosso dia a dia é quando queremos saber o dia anterior ou posterior a uma data. Por exemplo como saber qual a data de amanhã? Há diversos métodos pra nos ajudar com isso, vamos encontrar na API diversos métodos `minus` ou `plus` pras diferentes unidades de tempo, como por exemplo:

```
System.out.println(hoje.minusYears(1));
System.out.println(hoje.minusMonths(4));
System.out.println(hoje.minusDays(2));

System.out.println(hoje.plusYears(1));
System.out.println(hoje.plusMonths(4));
System.out.println(hoje.plusDays(2));
```

COPIAR CÓDIGO

Ou seja, pra saber a data de amanhã bastaria fazer `hoje.plusDays(1)` .

Uma API imutável

Sabendo disso podemos escrever o seguinte código para incrementar 4 anos na data atual, para saber quando será a próxima Olimpíada, por exemplo.

```
olimpiadasRio.plusYears(4);
System.out.println(olimpiadasRio);
```

COPIAR CÓDIGO

Mas repare que a saída desse código ainda será a data atual. Porque isso ocorreu? Da mesma forma que as novas API's, como o Stream, os métodos da API de datas sempre vão retornar uma nova instancia da sua data. Portanto precisamos fazer algo

como:

```
LocalDate proximasOlimpiadas = olimpiadasRio.plusYears(4);
System.out.println(proximasOlimpiadas);
```

COPIAR CÓDIGO

Ou seja, toda a API de datas é imutável. Ela nunca vai alterar a data original.

Ao executar esse código, um exemplo da saída seria `2020-06-25`.

Mas note que não é esse o formato que estamos acostumados a trabalhar, podemos então trabalhar com os diversos formatadores de datas existentes.

Formatando suas datas

Para formatar nossas datas podemos utilizar o `DateTimeFormatter`. Existem diversos já prontos, mas há ainda a alternativa de você criar o seu próprio formatador no padrão já conhecido de `dd/MM/yyyy`.

Para fazer isso basta você utilizar o método `ofPattern`:

```
DateTimeFormatter formatador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

COPIAR CÓDIGO

Agora podemos a partir da nossa data, neste caso `proximasOlimpiadas`, chamar o método `format` passando esse formatador:

```
String valorFormatado = proximasOlimpiadas.format(formatador);
System.out.println(valorFormatado);
```

COPIAR CÓDIGO

Agora sim, ao executar temos o resultado `05/06/2020`

Trabalhando com medida de tempo

Por enquanto só estamos trabalhando com datas, fazendo formatações e manipulando seu resultado. Mas é muito comum eu também precisar trabalhar com horas, minutos e segundos. Ou seja, trabalhar com uma medida de data com tempo.

Para isso podemos utilizar a classe `LocalDateTime` , de forma bem similar podemos fazer:

```
LocalDateTime agora = LocalDateTime.now();
```

COPIAR CÓDIGO

Podemos criar um novo formatador para mostrar as horas, minutos e segundos para conseguirmos ver o resultado já formatado:

```
DateTimeFormatter formatadorComHoras = DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm:ss");
LocalDateTime agora = LocalDateTime.now();
System.out.println(agora.format(formatadorComHoras));
```

COPIAR CÓDIGO

Pronto! Agora o resultado será algo como `05/06/2014 12:24:10` .

Lidando com modelos mais específicos

É muito comum ignorarmos valores quando precisamos apenas de algumas medidas de tempo, como por exemplo ano e mês. Nessa caso no lugar de criarmos um `LocalDate` ou algo assim e ignorar o seu valor de dia, podemos trabalhar com os modelos mais específicos da nova API.

Neste exemplo podemos usar o `YearMonth`, da seguinte forma:

```
YearMonth anoEMes = YearMonth.of(2015, Month.JANUARY);
```

COPIAR CÓDIGO

Ou seja, existem diversas novas classes para expressar bem nossas intenções.

Outro exemplo, para trabalharmos apenas com tempo podemos utilizar o `LocalTime`. Representar o horario do nosso intervalo de almoço, por exemplo, poderia ser feito com:

```
LocalTime intervalo = LocalTime.of(12, 30);
System.out.println(intervalo);
```

COPIAR CÓDIGO

Nesse caso a saída seria exatamente `12:30`. Não quer trabalhar com esse formato? Tudo bem, você pode criar um formatador como nós já fizemos, contanto que ele só tenha as medidas de tempo que existem, que neste caso são hora e minuto. Caso contrário uma exception será lançada.