

Java 8 Prático

Lambdas, Streams e os novos recursos
da linguagem



Casa do
Código

PAULO SILVEIRA
RODRIGO TURINI

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil



Casa do Código
Livros para o programador

**Uma editora de livros técnicos
feita por desenvolvedores
para desenvolvedores.**



Inscreva-se em nossa newsletter e
receba novidades e lançamentos

www.casadocodigo.com.br/newsletter



Curta nossa fanpage no Facebook

www.facebook.com/casadocodigo



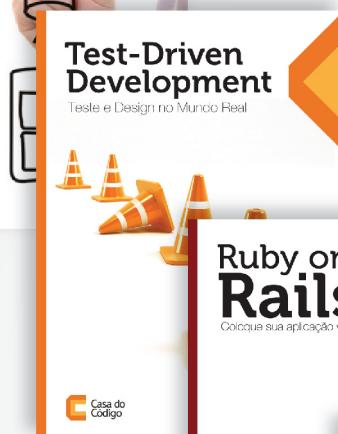
**Caelum:
Cursos de TI presenciais e online**

www.caelum.com.br



Dê seu feedback sobre o livro. Escreva para contato@casadocodigo.com.br

E-book gerado especialmente para Helton Simionato - helton5150@yahoo.com.br



E muito mais em:
www.casadocodigo.com.br



“À família Alexandre Vulcano”

– Paulo Silveira

“À minha amada esposa”

– Rodrigo Turini

Agradecimentos

Foi um desafio escrever um livro durante seis meses em que o JDK8 final ainda não existia.

Fica um agradecimento a todos que nos ajudaram com dúvidas, sugestões e participações na lista. Alberto Souza, Francisco Sokol, Guilherme Silveira, Michael Nascimento e Osvaldo Duoderlein são alguns deles. Agradecimento especial a Alexandre Aquiles, pelas correções e sugestões importantes durante a detalhada revisão.

Um muito obrigado a todos os desenvolvedores do JDK8, que em muitas vezes responderam nossas dúvidas prontamente. Destaque para o Brian Goetz, líder do projeto Lambda e sempre muito solícito.

Um abraço a todos da Caelum, do Alura e da Casa do Código. São equipes que nos incentivam todos os dias a investigar novas tecnologias, paradigmas e bibliotecas.

Sumário

1	Java 8	1
1.1	Um balde de água morna?	2
1.2	Acesse o código desse livro e discuta com a gente!	3
2	Olá, Lambda!	5
2.1	Loops da maneira antiga e da maneira nova	5
2.2	Que entre o Lambda!	8
3	Interfaces funcionais	11
3.1	Outro exemplo: listeners	13
3.2	Sua própria interface funcional	14
3.3	A anotação @FunctionalInterface	15
3.4	Indo mais a fundo: primeiros detalhes	16
4	Default Methods	19
4.1	O método forEach na interface Iterable	19
4.2	A interface Consumer não tem só um método!	20
4.3	Mais um novo método em Collection: removeIf	22
4.4	Herança múltipla?	23
5	Ordenando no Java 8	25
5.1	Comparators como lambda	25
5.2	O método List.sort	26
5.3	Métodos estáticos na interface Comparator	27
5.4	Conhecendo melhor o Comparator.comparing	28
5.5	Ordenando por pontos e o autoboxing	29

6 Method References	31
6.1 Tornando todos os usuários moderadores	31
6.2 Comparando de uma forma ainda mais enxuta	32
6.3 Compondo comparators	33
6.4 Referenciando métodos de instância	34
6.5 Referenciando métodos que recebem argumentos	35
6.6 Referenciando construtores	36
6.7 Outros tipos de referências	38
7 Streams e Collectors	41
7.1 Tornando moderadores os 10 usuários com mais pontos	41
7.2 Streams: tornando moderadores os usuários com mais de 100 pontos	42
7.3 Como obter de volta uma Lista?	46
7.4 Collectors	47
7.5 Avançado: por que não há um toList em Stream?	48
7.6 Liste apenas os pontos de todos os usuários com o map	49
7.7 IntStream e a família de Streams	50
7.8 O Optional em java.util	51
8 Mais operações com Streams	55
8.1 Ordenando um Stream	55
8.2 Muitas operações no Stream são lazy!	56
8.3 Qual é a vantagem dos métodos serem lazy?	57
8.4 Enxergando a execução do pipeline com peek	58
8.5 Operações de redução	59
8.6 Conhecendo mais métodos do Stream	61
8.7 Streams primitivos e infinitos	63
8.8 Praticando o que aprendemos com java.nio.file.Files	67
8.9 FlatMap	68
9 Mapeando, particionando, agrupando e paralelizando	71
9.1 Coletores gerando mapas	71
9.2 groupingBy e partitioningBy	74
9.3 Executando o pipeline em paralelo	78
9.4 Operações não determinísticas e ordered streams	79

10 Chega de Calendar! Nova API de datas	83
10.1 A java.time vem do Joda Time	83
10.2 Trabalhando com datas de forma fluente	84
10.3 Enums no lugar de constantes	88
10.4 Formatando com a nova API de datas	89
10.5 Datas inválidas	90
10.6 Duração e Período	91
10.7 Diferenças para o Joda Time	94
11 Um modelo de pagamentos com Java 8	97
11.1 Uma loja de digital goodies	97
11.2 Ordenando nossos pagamentos	101
11.3 Reduzindo BigDecimal em somas	101
11.4 Produtos mais vendidos	103
11.5 Valores gerados por produto	105
11.6 Quais são os produtos de cada cliente?	106
11.7 Qual é nosso cliente mais especial?	108
11.8 Relatórios com datas	109
11.9 Sistema de assinaturas	110
12 Apêndice: mais Java 8 com reflection, JVM, APIs e limitações	113
12.1 Novos detalhes na linguagem	113
12.2 Qual é o tipo de uma expressão Lambda?	117
12.3 Limitações da inferência no lambda	119
12.4 Fim da Permgen	122
12.5 Reflection: parameter names	122
13 Continuando seus estudos	125
13.1 Como tirar suas dúvidas	125
13.2 Bibliotecas que já usam ou vão usar Java 8	126

Versão: 17.0.31

CAPÍTULO 1

Java 8

São praticamente 20 anos de Java desde o lançamento de sua primeira versão.

Apenas em 2004, com a chegada do Java 5, houve mudanças significativas na linguagem. Em especial generics, enums e anotações.

Com a chegada do Java 8, em 2014, isso acontece mais uma vez. Novas possibilidades surgem com a entrada do lambda e dos method references, além de pequenas mudanças na linguagem. A API de Collections, na qual as interfaces principais são as mesmas desde 1998, recebe um significativo upgrade com a entrada dos Streams e dos métodos default.

Tudo isso será extensivamente praticado durante o livro. É hora de programar. Você deve baixar e instalar o Java 8:

<http://www.oracle.com/technetwork/java/javase/downloads/>

E pode acessar seu javadoc aqui:

<http://download.java.net/jdk8/docs/api/index.html>

O Eclipse possui suporte para o Java 8 a partir da versão Luna (4.4). O Kepler (4.3) precisa do seguinte update:

https://wiki.eclipse.org/JDT/Eclipse_Java_8_Support_For_Kepler

O Eclipse ainda possui alguns pequenos bugs para realizar inferências mais complicadas. Netbeans e IntelliJ tem suas versões atualizadas para Java 8.

Para fixar a sintaxe, você pode optar por realizar os testes e exemplos do livro com um simples editor de texto.

1.1 UM BALDE DE ÁGUA MORNA?

Se você está esperando algo tão poderoso quanto em Scala, Clojure e C#, certamente sairá decepcionado. O legado e a idade do Java, além da ausência de value types e reificação nos generics, impedem o uso de algumas estratégias. O time de desenvolvedores da linguagem tem também grande preocupação em deixar a sintaxe sempre simples, evitando formas obscuras que trariam pequenos ganhos. Na nossa opinião, faz bastante sentido.

Ao mesmo tempo é impressionante o que foi possível atingir com o lançamento dessa nova versão. Você vai se surpreender com alguns códigos e abordagens utilizadas. O foco é não quebrar a compatibilidade do código antigo e ser o menos intrusivo possível nas antigas APIs. Os Streams, que serão vistos quase que a exaustão, certamente têm um papel crucial nessa elegante evolução.

O que ficou de fora do Java 8?

Para quebrar melhor a especificação do Java 8 em tarefas menores, foram criadas as JEPs: JDK Enhancement Proposals. É uma ideia que nasceu dos PEPs, proposta similar da comunidade Python. A JEP o é uma lista com todas essas propostas:

<http://openjdk.java.net/jeps/0>

Como você pode ver, são muitas as novidades no JDK8. Infelizmente nem todas tiveram tempo suficiente para amadurecer. Entre as JEPs propostas, os Value Objects ficaram de fora:

<http://openjdk.java.net/jeps/169>

Assim como o uso de literais para trabalhar com coleções:

<http://openjdk.java.net/jeps/186>

Entre outras ideias que ficaram de fora, temos diversas melhorias aos Garbage Collectors já embutidos, assim como a possível reificação dos generics.

De qualquer maneira, a maioria absoluta das JEPs sobreviveu até o lançamento da versão final. Veremos as principais mudanças da linguagem, assim como as novas APIs, no decorrer do livro.

1.2 ACESSE O CÓDIGO DESSE LIVRO E DISCUTA COM A GENTE!

O código-fonte para cada capítulo pode ser encontrado aqui:

<https://github.com/peas/java8>

É claro que indicamos que você mesmo escreva todo o código apresentado no livro, para praticar a API e a sintaxe, além de realizar testes diferentes dos sugeridos.

Há uma lista de discussão por onde você pode conversar com a gente, mandar sugestões, críticas e melhorias:

<https://groups.google.com/forum/#!forum/java8-casadocodigo>

Se preferir, você pode tirar dúvidas no fórum do GUJ:

<http://www.guj.com.br/>

CAPÍTULO 2

Olá, Lambda!

Em vez de começar com uma boa dose de teoria, é importante que você sinta na prática como o Java 8 vai mudar a forma com que você programa.

2.1 LOOPS DA MANEIRA ANTIGA E DA MANEIRA NOVA

É importante que você acompanhe o livro recriando o código daqui. Só assim a sintaxe tornar-se-á natural e mais próxima de você.

Abra seu editor preferido. Vamos criar uma entidade para poder rodar exemplos baseados nela. Teremos a classe `Usuario`, com três atributos básicos: `pontos`, `nome` e um `boolean moderador`, indicando se aquele usuário é um moderador do nosso sistema. Simples assim:

```
class Usuario {  
    private String nome;  
    private int pontos;  
    private boolean moderador;
```

```
public Usuario(String nome, int pontos) {
    this.pontos = pontos;
    this.nome = nome;
    this.moderador = false;
}

public String getNome() {
    return nome;
}

public int getPontos() {
    return pontos;
}

public void tornaModerador() {
    this.moderador = true;
}

public boolean isModerador() {
    return moderador;
}
}
```

Não declaramos a classe como pública propositalmente. Caso você esteja em um editor de texto simples, poderá criar seus testes em uma classe pública dentro do mesmo arquivo.

Vamos manipular alguns usuários, com nomes e pontos diferentes, e imprimir cada um deles. Faremos isso da maneira clássica que já conhecemos, sem nenhuma novidade da nova versão da linguagem.

```
public class Capitulo2 {
    public static void main(String ... args) {
        Usuario user1 = new Usuario("Paulo Silveira", 150);
        Usuario user2 = new Usuario("Rodrigo Turini", 120);
        Usuario user3 = new Usuario("Guilherme Silveira", 190);

        List<Usuario> usuarios = Arrays.asList(user1, user2, user3);

        for(Usuario u : usuarios) {
            System.out.println(u.getNome());
```

```
    }  
}  
}
```

Omitimos dois imports: do `java.util.List` e do `java.util.Arrays`. Durante o livro, eles não vão aparecer, mas notaremos sempre que aparecerem novos pacotes do Java 8.

`Arrays.asList` é uma maneira simples de criar uma `List` imutável. Mas você poderia sim ter criado uma nova `ArrayList` e adicionado cada um dos usuários.

O `for` que fazemos é trivial. Desde o Java 5, podemos navegar assim em qualquer array ou coleção (na verdade, em qualquer tipo de objeto que implemente a interface `java.lang.Iterable`).

Um novo método em todas as coleções: `forEach`

A partir do Java 8 temos acesso a um novo método nessa nossa lista: o `forEach`. De onde ele vem? Veremos mais adiante. Iniciaremos por utilizá-lo. Podemos fazer algo como:

```
usuarios.forEach(...);
```

Para cada usuário, o que ele deve fazer? Imprimir o nome. Mas qual é o argumento que esse método `forEach` recebe?

Ele recebe um objeto do tipo `java.util.function.Consumer`, que tem um único método, o `accept`. Ela é uma nova interface do Java 8, assim como todo o pacote do `java.util.function`, que conheceremos no decorrer do livro.

Vamos criar esse consumidor, antes de usar o novo `forEach`:

```
class Mostrador implements Consumer<Usuario> {  
    public void accept(Usuario u) {  
        System.out.println(u.getNome());  
    }  
}
```

Criamos uma classe que implementa essa nova interface do Java 8. Ela é bem trivial, possuindo o único método `accept` responsável por pegar um objeto do tipo `Usuario` e consumi-lo. ‘Consumir’ aqui é realizar alguma tarefa que faça sentido pra você. No nosso caso, é mostrar o nome do usuário na saída padrão. Depois disso, podemos instanciar essa classe e passar a referência para o esperado método `forEach`:

```
Mostrador mostrador = new Mostrador();
usuarios.forEach(mostrador);
```

Sabemos que é uma prática comum utilizar classes anônimas para essas tarefas mais simples. Em vez de criar uma classe `Mostrador` só pra isso, podemos fazer tudo de uma tacada só:

```
Consumer<Usuario> mostrador = new Consumer<Usuario>() {
    public void accept(Usuario u) {
        System.out.println(u.getNome());
    }
};

usuarios.forEach(mostrador);
```

Isso vai acabar gerando um `.class` com nome estranho, como por exemplo `Capitulo2$1.class`. Como não podemos nos referenciar a um nome dessa classe, chamamo-la de classe anônima, como já deve ser de seu conhecimento.

O código ainda está grande. Parece que o `for` de maneira antiga era mais suínto. Podemos reduzir um pouco mais esse código, evitando a criação da variável local `mostrador`:

```
usuarios.forEach(new Consumer<Usuario>() {
    public void accept(Usuario u) {
        System.out.println(u.getNome());
    }
});
```

Pronto! Um pouco mais enxuto, mas ainda assim bastante verboso.

2.2 QUE ENTRE O LAMBDA!

Simplificando bastante, um lambda no Java é uma maneira mais simples de implementar uma interface que só tem um único método. No nosso caso, a interface `Consumer` é uma boa candidata.

Isto é, em vez de escrever:

```
Consumer<Usuario> mostrador = new Consumer<Usuario>() {
    public void accept(Usuario u) {
        System.out.println(u.getNome());
    }
};
```

Podemos fazer de uma maneira mais direta. Repare a mágica:

```
Consumer<Usuario> mostrador =  
    (Usuario u) -> {System.out.println(u.getNome());};
```

O trecho `(Usuario u) -> {System.out.println(u.getNome());}` é um lambda do Java 8. O compilador percebe que você o está atribuindo a um `Consumer<Usuario>` e vai tentar jogar esse código no único método que essa interface define. Repare que não citamos nem a existência do método `accept!` Isso é inferido durante o processo de compilação.

Podemos ir além. O compilador consegue também inferir o tipo, sem a necessidade de utilizar `Usuario`, nem parênteses:

```
Consumer<Usuario> mostrador =  
    u -> {System.out.println(u.getNome());};
```

Não está satisfeito? Caso o bloco dentro de `{ }` contenha apenas uma instrução, podemos omiti-lo e remover também o ponto e vírgula:

```
Consumer<Usuario> mostrador =  
    u -> System.out.println(u.getNome());
```

Agora fica até possível de escrever em uma única linha:

```
Consumer<Usuario> mostrador = u -> System.out.println(u.getNome());
```

Então `u -> System.out.println(u.getNome())` infere pro mesmo lambda que `(Usuario u) -> {System.out.println(u.getNome());}`, se forem atribuídos a um `Consumer<Usuario>`. Podemos passar esse trecho de código diretamente para `usuarios.forEach` em vez de declarar a variável temporária `mostrador`:

```
usuarios.forEach(u -> System.out.println(u.getNome()));
```

Difícil? Certamente não. Porém pode levar algumas semanas até você se habituar com a sintaxe, além das diferentes possibilidades de utilizá-la. Vamos conhecer algumas nuances e trabalhar bastante com a API, além de saber como isso foi implementado, que é um pouco diferente das classes anônimas. Se você observar os `.class` gerados, poderá perceber que o compilador não criou os diversos arquivos `Capitulo2$N.class`, como costuma fazer pra cada classe anônima.

Vamos a mais um exemplo essencial. Você pode, em vez de imprimir o nome de todos os usuários, tornar todos eles moderadores!

```
usuarios.forEach(u -> u.tornaModerador());
```

Vale lembrar que essa variável `u` não pode ter sido declarada no mesmo escopo da invocação do `forEach`, pois o lambda pode *capturar* as variáveis de fora, como veremos mais adiante.

No próximo capítulo vamos trabalhar mais com lambda, para exercitar a sintaxe, além de conhecer outros cenários básicos de seu uso junto com o conceito de interface funcional.

CAPÍTULO 3

Interfaces funcionais

Repare que a interface `Consumer<Usuario>`, por exemplo, tem apenas um único método abstrato, o `accept`. É por isso que, quando faço o `forEach` a seguir, o compilador sabe exatamente qual método deverá ser implementado com o corpo do meu lambda:

```
usuarios.forEach(u -> System.out.println(u.getNome()));
```

Mas e se a interface `Consumer<Usuario>` tivesse dois métodos? O fato de essa interface ter apenas um método não foi uma coincidência, mas sim um requisito para que o compilador consiga traduzi-la para uma expressão lambda. Podemos dizer então que toda interface do Java que possui **apenas um método abstrato** pode ser instanciada como um código lambda!

Isso vale até mesmo para as interfaces antigas, pré-Java 8, como por exemplo o `Runnable`:

```
public interface Runnable {  
    public abstract void run();  
}
```

Apenas um lembrete: por padrão, todos os métodos de uma interface no Java são públicos e abstratos. Veremos, mais à frente, que há um novo tipo de método nas interfaces.

Normalmente, escrevemos o seguinte trecho para instanciar uma `Thread` e um `Runnable` que conta de 0 a 1000:

```
Runnable r = new Runnable(){
    public void run(){
        for (int i = 0; i <= 1000; i++) {
            System.out.println(i);
        }
    }
};

new Thread(r).start();
```

A interface `Runnable` tem apenas um único método abstrato. Uma interface que se enquadre nesse requisito é agora conhecida como uma **interface funcional!**. Ela sempre pode ser instanciada através de uma expressão lambda:

```
Runnable r = () -> {
    for (int i = 0; i <= 1000; i++) {
        System.out.println(i);
    }
};

new Thread(r).start();
```

Poderíamos ir além e fazer tudo em um único *statement*, com talvez um pouco menos de legibilidade:

```
new Thread(() -> {
    for (int i = 0; i <= 1000; i++) {
        System.out.println(i);
    }
}).start();
```

Como você já sabe, existe um novo pacote no Java 8, o `java.util.function`, com uma série de interfaces funcionais que podem e devem ser reaproveitadas. Conheceremos diversas delas ao decorrer do nosso estudo.

3.1 OUTRO EXEMPLO: LISTENERS

Um outro uso bem natural de classe anônima é quando precisamos adicionar uma ação no click de um objeto do tipo `java.awt.Button`. Para isso, precisamos implementar um `ActionListener`. Você já pode ter visto um código parecido com este:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("evento do click acionado");
    }
});
```

Usamos a interface `ActionListener` como nada mais do que uma função de retorno, e pela sua estrutura de um único método ela se enquadra no requisito de interface funcional:

```
public interface ActionListener extends EventListener {
    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);
}
```

Assim como toda interface funcional, também podemos representá-la como uma expressão lambda:

```
button.addActionListener( event) -> {
    System.out.println("evento do click acionado");
};
```

Como já vimos, essa expressão pode ficar ainda mais simples, sem parênteses no único argumento e podemos também remover o `{ } e ; :`

```
button.addActionListener(
    event -> System.out.println("evento do click acionado"));
```

Pronto! Agora estamos dando a mesma ação ao `button`, só que substituindo as 5 linhas que usamos com uma classe anônima por uma única linha de expressão lambda.

Assim como a `ActionListener` que já existe e é comumente usada nas versões pré-Java 8, existem diversas outras interfaces do Java que possuem essa mesma estrutura de um único método, como por exemplo as interfaces `java.util.Comparator`, `java.util.concurrent.Callable` e `java.io.FileFilter`. Além do `java.lang.Runnable`, que já vimos.

Mesmo sem terem alterado nada em sua estrutura interna, todas essas interfaces podem ser chamadas, a partir dessa nova versão da linguagem, de interfaces funcionais!

3.2 SUA PRÓPRIA INTERFACE FUNCIONAL

Você não precisa fazer nada de especial para que uma interface seja considerada funcional. O compilador já identifica esse tipo de interface pela sua estrutura.

Imagine que temos uma interface de `Validador<T>`, com um método que `valida(T t)` e que devolve `boolean`:

```
interface Validador<T> {  
    boolean valida(T t);  
}
```

Costumamos utilizá-la criando uma classe anônima, desta forma:

```
Validador<String> validadorCEP = new Validador<String>() {  
    public boolean valida(String valor) {  
        return valor.matches("[0-9]{5}-[0-9]{3}");  
    }  
};
```

E como podemos usar essa interface com Lambda a partir do Java 8? O que precisamos modificar em sua declaração?

A resposta é fácil: absolutamente nada. Como já havíamos falado, bastando a interface possuir um único método abstrato, ela é considerada uma interface funcional e pode ser instanciada através de uma expressão lambda! Veja:

```
Validador<String> validadorCEP =  
    valor -> {  
        return valor.matches("[0-9]{5}-[0-9]{3}");  
    };
```

Para testar você pode executar em um método main simples, o código

```
validadorCEP.valida("04101-300");
```

Nosso lambda está um pouco grande. Vimos mais de uma vez que, quando há uma única instrução, podemos resumi-la. E isso acontece mesmo se ela for uma instrução de `return!` Podemos remover o próprio `return`, assim como o seu ponto e vírgula e as chaves delimitadoras:

```
Validador<String> validadorCEP =  
    valor -> valor.matches("[0-9]{5}-[0-9]{3}");
```

3.3 A ANOTAÇÃO @FUNCTIONALINTERFACE

Podemos marcar uma interface como funcional explicitamente, para que o fato de ela ser uma interface funcional não seja pela simples coincidência de ter um único método. Para fazer isso, usamos a anotação `@FunctionalInterface`:

```
@FunctionalInterface  
interface Validador<T> {  
    boolean valida(T t);  
}
```

Se você fizer essa alteração e compilar o código do nosso `Validador<T>`, vai perceber que nada mudou. Mas diferente do caso de não termos anotado nossa interface com `@FunctionalInterface`, tente alterá-la da seguinte forma, adicionando um novo método:

```
@FunctionalInterface  
interface Validador<T> {  
    boolean valida(T t);  
    boolean outroMetodo(T t);  
}
```

Ao compilar esse código, recebemos o seguinte erro:

```
java: Unexpected @FunctionalInterface annotation  
      Validador is not a functional interface  
      multiple non-overriding abstract methods found in interface Exemplo
```

Essa anotação serve apenas para que ninguém torne aquela interface em não-funcional acidentalmente. Ela é opcional justamente para que as interfaces das antigas bibliotecas possam também ser tratadas como lambdas, independente da anotação, bastando a existência de um único método abstrato.

3.4 INDO MAIS A FUNDO: PRIMEIROS DETALHES

Interfaces funcionais são o coração do recurso de Lambda. O Lambda por si só não existe, e sim expressões lambda, quando atribuídas/inferidas a uma interface funcional. Durante o desenvolvimento do Java 8, o grupo que tocou o Lambda chamava essas interfaces de SAM Types (*Single Abstract Method*).

Se uma interface funcional não está envolvida, o compilador não consegue trabalhar com ela:

```
Object o = () -> {
    System.out.println("O que sou eu? Que lambda?");
};
```

Isso retorna um error: incompatible types: Object is not a functional interface. Faz sentido.

É sempre necessário haver a atribuição (ou alguma forma de inferir) daquele lambda para uma interface funcional. A classe Object certamente não é o caso!

Se atribuirmos o lambda para um tipo que seja uma interface funcional compatível, isso é, com os argumentos e retornos que o método necessita, aí sim temos sucesso na compilação:

```
Runnable o = () -> {
    System.out.println("O que sou eu? Que lambda?");
};
```

Mas o que exatamente é o objeto retornado por esta expressão? Vamos descobrir mais informações:

```
Runnable o = () -> {
    System.out.println("O que sou eu? Que lambda?");
};

System.out.println(o);
System.out.println(o.getClass());
```

A saída não é muito diferente do que você espera, se já conhece bem as classes anônimas:

```
Capitulo3$$Lambda$1@1fc625e
class Capitulo3$$Lambda$1
```

A classe gerada que representa esse nosso Lambda é a `Capitulo3$$Lambda$1`. Na verdade, esse nome vai depender de quantos lambdas você tem na sua classe `Capitulo3`. Repare que não foi gerado um `.class` no seu diretório, essa classe é criada dinamicamente!

A tradução de uma expressão Lambda para o bytecode Java não é trivial. Poderia ser feito de várias maneiras, até mesmo da maneira como as classes anônimas trabalham, mas elas não funcionam assim. Depois de muita discussão, optaram por utilizar `MethodHandles` e `invokedynamic` para isso. Você pode ver detalhes avançados sobre a implementação aqui:

<http://cr.openjdk.java.net/~char126briangoetz/lambda/lambda-translation.html>

Aqui há um comparativo da implementação do Lambda do Scala com a do Java 8:

<http://www.takipiblog.com/2014/01/16/compiling-lambda-expressions-scala-vs-java-8/>

No capítulo [12.2](#), conhceremos mais detalhes sobre a inferência de tipos e outras peculiaridades do lambda.

Captura de variáveis locais

Assim como numa classe anônima local, você também pode acessar as variáveis finais do método a qual você pertence:

```
public static void main(String[] args) {
    final int numero = 5;
    new Thread(() -> {
        System.out.println(numero);
    }).start();
}
```

Um lambda do Java *enclausura* as variáveis que estavam naquele contexto, assim com as classes anônimas.

Quer mais? Você pode até mesmo acessar a variável local que não é final:

```
public static void main(String[] args) {
    int numero = 5;
    new Thread(() -> {
        System.out.println(numero);
    }).start();
}
```

Porém, ela deve ser *efetivamente final*. Isso é, apesar de não precisar declarar as variáveis locais como `final`, você não pode alterá-las se estiver utilizando-as dentro do lambda. O seguinte código não compila:

```
public static void main(String[] args) {  
    int numero = 5;  
    new Thread(() -> {  
        System.out.println(numero); // não compila  
    }).start();  
  
    numero = 10; // por causa dessa linha!  
}
```

E isso também vale para as classes anônimas a partir do Java 8. Você não precisa mais declarar as variáveis locais como `final`, basta não alterá-las que o Java vai permitir acessá-las.

Claro que problemas de concorrência ainda podem acontecer no caso de você invocar métodos que alterem estado dos objetos envolvidos. Veremos alguns detalhes disso mais à frente.

CAPÍTULO 4

Default Methods

4.1 O MÉTODO FOREACH NA INTERFACE ITERABLE

Até agora engolimos o método `forEach` invocado em nossa `List`. De onde ele vem, se sabemos que até o Java 7 isso não existia?

Uma possibilidade seria a equipe do Java tê-lo declarado em uma interface, como na própria `List`. Qual seria o problema? Todo mundo que criou uma classe que implementa `List` precisaria implementá-lo. O trabalho seria enorme, mas esse não é o principal ponto. Ao atualizar o seu Java, bibliotecas que têm sua própria `List`, como o Hibernate, teriam suas implementações quebradas, faltando métodos, podendo gerar os assustadores `NoSuchMethodErrors`.

Como adicionar um método em uma interface e garantir que todas as implementações o possuam implementado? Com um novo recurso, declarando código dentro de um método de uma interface!

Por exemplo, o método `forEach` que utilizamos está declarado dentro de `java.lang.Iterable`, que é mãe de `Collection`, por sua vez mãe de `List`. Abrindo seu código-fonte, podemos ver:

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

Pois é. Método com código dentro de interfaces! Algo que nunca esperaríamos ser possível no Java. Esses métodos são chamados de *default methods* (na documentação do beta pode ser também encontrado *defender methods* ou ainda *extension methods*).

Como `ArrayList` implementa `List`, que é filha (indireta) de `Iterable`, a `ArrayList` possui esse método, quer ela queira ou não. É por esse motivo que podemos fazer:

```
usuarios.forEach(u -> System.out.println(u.getNome()));
```

Esse *default method*, o `forEach`, espera receber uma implementação da interface funcional `Consumer<T>` que já vimos ser uma das interfaces presentes no novo pacote `java.util.function` do Java 8. Ela possui um único método abstrato, o `accept`, que recebe o tipo parametrizado `T` e executa o que você achar interessante.

4.2 A INTERFACE CONSUMER NÃO TEM SÓ UM MÉTODO!

Tomamos o cuidado, nos capítulos anteriores, de deixar claro que uma interface funcional é aquela que possui apenas um método **abstrato**! Ela pode ter sim mais métodos, desde que sejam métodos *default*.

Observe que a interface `Consumer<T>`, além do método `accept(T t)`, ainda possui o *default method* `andThen`:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Repare que, por ser um método *default*, a classe pode ser explicitamente anotada com `@FunctionalInterface`.

O método `andThen` pode ser usado para compor instâncias da interface `Consumer` para que possam ser executadas sequencialmente, por exemplo:

```
public class Capitulo4 {  
    public static void main(String ... args) {  
  
        Usuario user1 = new Usuario("Paulo Silveira", 150);  
        Usuario user2 = new Usuario("Rodrigo Turini", 120);  
        Usuario user3 = new Usuario("Guilherme Silveira", 190);  
  
        List<Usuario> usuarios = Arrays.asList(user1, user2, user3);  
  
        Consumer<Usuario> mostraMensagem = u ->  
            System.out.println("antes de imprimir os nomes");  
  
        Consumer<Usuario> imprimeNome = u ->  
            System.out.println(u.getNome());  
  
        usuarios.forEach(mostraMensagem.andThen(imprimeNome));  
    }  
}
```

Rode o código. Você vai ver que ambos os consumidores serão executados:

```
antes de imprimir os nomes  
Paulo Silveira  
antes de imprimir os nomes  
Rodrigo Turini  
antes de imprimir os nomes  
Guilherme Silveira
```

Podemos então ter implementações comuns de `Consumer`, e utilizá-las em diferentes momentos do nosso código, passando-os como argumentos e depois compondo-os de maneira a reutilizá-los. Por exemplo, se você tem um `Consumer<Usuario>` `auditor` que guarda no log que aquele usuário realizou algo no sistema, você pode reutilizá-lo, com ou sem o `andThen`. Um bom exemplo do pattern *decorator*.

4.3 MAIS UM NOVO MÉTODO EM COLLECTION: REMOVEIF

A interface `Collection` ganhou novos métodos com implementação *default*. Vimos o `forEach`. Um outro é o `removeIf`, que recebe um `Predicate`. O `Predicate` é uma interface funcional que permite testar objetos de um determinado tipo. Dado um `Predicate`, o `removeIf` vai remover todos os elementos que devolverem `true` para esse *predicado*.

Se desejarmos remover todos os usuários com mais de 100 pontos da nossa coleção, podemos fazer:

```
Predicate<Usuario> predicado = new Predicate<Usuario>() {  
    public boolean test(Usuario u) {  
        return u.getPontos() > 160;  
    }  
};  
  
List<Usuario> usuarios = new ArrayList<>();  
usuarios.add(user1);  
usuarios.add(user2);  
usuarios.add(user3);  
  
usuarios.removeIf(predicado);  
usuarios.forEach(u -> System.out.println(u));
```

Há um detalhe aqui: o `removeIf` invoca o `remove` de uma coleção, então ela não pode ser imutável, ou um `UnsupportedOperationException` será lançado. Esse é o caso da lista devolvida por `Arrays.asList`. Por isso, foi necessário utilizar uma coleção mutável como `ArrayList`.

Podemos também usar aqui o lambda. Não precisamos nem mesmo declarar a variável `predicado`! Podemos passar o `u.getPontos() > 160` direto para o `removeIf`:

```
usuarios.removeIf(u -> u.getPontos() > 160);
```

Quase sempre é vantajoso utilizar um lambda em vez de criar uma classe anônima. Uma das restrições do lambda é que não podemos declarar atributos, isto é, manter estado dentro dela. Essa situação não aparece com frequência, mas veremos sim alguns casos em que o uso do lambda não será possível ou teria de envolver atributos da classe externa.

Veremos outros métodos que entraram nas coleções. Por exemplo, `Map` ganhou `computeIfPresent`, `computeIfAbsent`, `getOrDefault` e outros métodos úteis que costumavam nos forçar a escrever um código bem comum. Não poderemos ver todos, mas utilizaremos diversas novas funcionalidades, interfaces e métodos no decorrer do livro.

4.4 HERANÇA MÚLTIPLA?

Métodos defaults foram adicionados para permitir que interfaces evoluam sem quebrar código existente. Essa é uma das frases mais repetidas na lista de discussão da especificação.

Eles não foram criados para permitir alguma variação de herança múltipla ou de mixins. Vale lembrar que há uma série de restrições para esses métodos. Em especial, eles não podem acessar atributos de instância, até porque isso não existe em interfaces! Em outras palavras, não há herança múltipla ou compartilhamento de estado.

Brian Goetz e os outros líderes da especificação possuem argumentos fortes na seção 10 deste documento:

<http://cr.openjdk.java.net/\char126briangoetz/lambda/lambda-state-final.html>

É certo que há um aumento no acoplamento mas, dadas as restrições desse recurso no Java 8, não há a mesma possibilidade de usos nocivos como em outras linguagens. É sempre um trade-off.

CAPÍTULO 5

Ordenando no Java 8

5.1 COMPARATORS COMO LAMBDA

Se uma classe implementa `Comparable`, podemos passar uma lista de instâncias dessa classe para o `Collections.sort`. Esse não é o caso de nossa classe `Usuario`. Como é bem sabido, precisamos de um `Comparator<Usuario>`.

Vamos criar um que ordena pelo nome do usuário, e depois pedir para que a lista seja ordenada:

```
Comparator<Usuario> comparator = new Comparator<Usuario>() {
    public int compare(Usuario u1, Usuario u2) {
        return u1.getNome().compareTo(u2.getNome());
    }
};

Collections.sort(usuarios, comparator);
```

Adivinhe! `Comparator` é uma interface funcional, com apenas um método abstrato. Podemos tirar proveito do lambda e reescrever a instanciação daquela classe

anônima de maneira mais simples:

```
Comparator<Usuario> comparator =  
    (u1, u2) -> u1.getNome().compareTo(u2.getNome());  
  
Collections.sort(usuarios, comparator);
```

Ou ainda, colocando tudo em uma mesma linha, sem a declaração da variável local:

```
Collections.sort(usuarios,  
    (u1, u2) -> u1.getNome().compareTo(u2.getNome()));
```

Muito melhor que o antigo código, não?

OS CUIDADOS COM COMPARATORS

Atenção! Para deixar o código mais sucinto, não nos precavemos aqui de possíveis usuários com atributo `nome` igual a `null`. Mesmo sendo uma invariante do seu sistema, é importante sempre checar esses casos particulares e definir se um usuário com nome nulo iria para o começo ou fim nesse critério de comparação.

Há também a boa prática de utilizar comparators que já existem, como o `String.CASE_INSENSITIVE_ORDER`. Seu código ficaria `return String.CASE_INSENSITIVE_ORDER.compare(u1.getNome(), u2.getNome())` ou ainda algum dos `java.text.Collators`, junto com as verificações de valores nulo.

5.2 O MÉTODO LIST.SORT

Podemos ordenar uma lista de usuários de forma ainda mais sucinta:

```
usuarios.sort((u1, u2) -> u1.getNome().compareTo(u2.getNome()));
```

Isso finalmente é possível pois existe um novo método default `sort` declarado na interface `List`, que simplesmente delega a invocação para o já conhecido `Collections.sort`:

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

Esse é um excelente exemplo de uso de métodos default, pois permitiu a evolução da interface `List` sem quebrar o código existente.

5.3 MÉTODOS ESTÁTICOS NA INTERFACE COMPARATOR

É isso mesmo: além de poder ter métodos default dentro de uma interface, agora podemos ter métodos estáticos. A interface `Comparator` possui alguns deles.

O nosso `usuarios.sort(...)` pode ficar ainda mais curto, com o auxílio do `Comparator.comparing(...)`, que é uma fábrica (*factory*) de `Comparators`. Repare:

```
Comparator<Usuario> comparator =  
    Comparator.comparing(u -> u.getNome());  
  
usuarios.sort(comparator);
```

Ele retorna um `Comparator` que usará a `String` do nome do usuário como critério de comparação.

Ainda podemos colocar tudo em uma linha, sem a declaração da variável local:

```
usuarios.sort(Comparator.comparing(u -> u.getNome()));
```

Com o uso do `static import`, poderemos fazer apenas `usuarios.sort(comparing(u -> u.getNome()));` e ainda veremos como melhorar ainda mais no próximo capítulo!

O uso do `comparing` só funciona passando um lambda que, dado um tipo `T` (no nosso caso um usuário), devolve algo que seja `Comparable<U>`. No nosso exemplo, devolvemos uma `String`, o nome do usuário, que é `Comparable<String>`. Veremos um pouco melhor seu funcionamento mais à frente.

Indexando pela ordem natural

E se tivermos uma lista de objetos que implementam `Comparable`, como por exemplo `String`? Antes faríamos:

```
List<String> palavras =  
    Arrays.asList("Casa do Código", "Alura", "Caelum");  
  
Collections.sort(palavras);
```

Tentar fazer, no Java 8, `palavras.sort()` não compila. Não há esse método `sort` em `List` que não recebe parâmetro. Nem teria como haver, pois o compilador não teria como saber se o objeto invocado é uma `List` de `Comparable` ou de suas filhas.

Como resolver? Criar um lambda de `Comparator` que simplesmente delega o trabalho para o `compareTo`? Não é necessário, pois isso já existe:

```
List<String> palavras =  
    Arrays.asList("Casa do Código", "Alura", "Caelum");  
  
palavras.sort(Comparator.naturalOrder());
```

`Comparator.naturalOrder()` retorna um `Comparator` que delega para o próprio objeto. Há também o `Comparator.reverseOrder()`.

5.4 CONHECENDO MELHOR O COMPARATOR.COMPARING

Apesar de não ser o nosso foco, é interessante ver como um método desses é implementado. Não se preocupe se você não domina todos os artifícios do generics e seus lower e upper bounds. Vamos ao método:

```
public static <T, U extends Comparable<? super U>> Comparator<T>  
    comparing(Function<? super T, ? extends U> keyExtractor) {  
    Objects.requireNonNull(keyExtractor);  
    return (Comparator<T> & Serializable)  
        (c1, c2) -> keyExtractor.apply(c1).  
            compareTo(keyExtractor.apply(c2));  
}
```

Dado um tipo `T`, o `comparing` recebe um lambda que devolve um tipo `U`. Isso é definido pela nova interface do Java 8, a `Function`:

Depois disso, ele usa a sintaxe do lambda para criar um `Comparator`, que já utilizamos no começo deste capítulo. A chamada do `apply` faz com que o nosso `u.getNome()` seja invocado. Repare que `apply` é o único método abstrato da

interface Function e que o comparing gerou um Comparator que também não previne o caso de a chave de comparação ser nula.

Você poderia usar o comparing passo a passo, para isso ficar mais claro:

```
Function<Usuario, String> extraiNome = u -> u.getNome();
Comparator<Usuario> comparator =
    Comparator.comparing(extraiNome);

usuarios.sort(comparator);
```

5.5 ORDENANDO POR PONTOS E O AUTOBOXING

É bem simples realizarmos a ordenação dos nossos usuários pela quantidade de pontos que cada um tem, em vez do nome. Basta alterarmos o lambda passado como argumento:

```
usuarios.sort(Comparator.comparing(u -> u.getPontos()));
```

Para enxergar melhor o que acontece, podemos quebrar esse código em mais linhas e variáveis locais, como já vimos:

```
Function<Usuario, Integer> extraiPontos = u -> u.getPontos();
Comparator<Usuario> comparator =
    Comparator.comparing(extraiPontos);

usuarios.sort(comparator);
```

Autoboxing nos lambdas

Há um problema aqui. O extraiPontos gerado terá um método apply que recebe um Usuario e devolve um Integer, em vez de um int. Isso gerará um autoboxing toda vez que esse método for invocado. E ele poderá ser invocado muitíssimas vezes pelo sort, através do compare do Comparator devolvido pelo Comparator.comparing.

Esse tipo de problema vai aparecer diversas vezes nas novas APIs do Java 8. Para evitar esse autoboxing (e às vezes o unboxing) desnecessário, há interfaces equivalentes que trabalham diretamente com long, double e int.

Em vez de utilizar a interface Function, devemos nesse caso utilizar oToIntFunction e, consequentemente, o método comparingInt, que recebe ToIntFunction em vez de Function:

```
ToIntFunction<Usuario> extraiPontos = u -> u.getPontos();  
Comparator<Usuario> comparator =  
    Comparator.comparingInt(extraiPontos);  
  
usuarios.sort(comparator);
```

Claro que você pode (e deve) usar a versão mais enxuta, passando diretamente o lambda para a fábrica de comparators, e até mesmo invocar o `sort` na mesma linha:

```
usuarios.sort(Comparator.comparingInt(u -> u.getPontos()));
```

CAPÍTULO 6

Method References

Utilizamos as expressões lambda para simplificar o uso daquelas funções de *callback*, interfaces comumente instanciadas como classes anônimas, como vimos com o `ActionListener` pra dar uma ação ao `button`.

Com o uso das expressões lambda, escrevemos um código muito mais conciso, então evitamos o que é conhecido nas listas de discussões do Java 8 como **problema vertical**, aquelas várias linhas de código para executar uma ação simples.

6.1 TORNANDO TODOS OS USUÁRIOS MODERADORES

Note que muitas vezes vamos escrever uma expressão lambda que apenas delega a invocação para um método existente. Por exemplo, como faríamos para tornar moderadores todos os elementos de uma lista de usuários? Poderíamos fazer um `forEach` simples chamando o método `tornaModerador` de cada elemento iterado:

```
usuarios.forEach(u -> u.tornaModerador());
```

Mas é possível executar essa mesma lógica de uma forma muito mais simples, apenas informando ao nosso `forEach` qual método deverá ser chamado. Podemos fazer isso usando um novo recurso da linguagem, o *method reference*.

```
usuarios.forEach(Usuario::tornaModerador);
```

Repare que a sintaxe é bem simples: eu tenho inicialmente o tipo (nesse caso a classe `Usuario`), seguido do delimitador `:` e o nome do método, sem parênteses.

Já vimos que o método `forEach` espera receber um `Consumer<Usuario>` como parâmetro, mas por que eu posso passar uma referência de método no lugar dessa interface funcional? Você já deve imaginar a resposta: da mesma forma como uma expressão lambda, o *method reference* é traduzido para uma interface funcional! Portanto, o seguinte código compila:

```
Consumer<Usuario> tornaModerador = Usuario::tornaModerador;  
usuarios.forEach(tornaModerador);
```

Esse poderoso recurso é tratado pelo compilador praticamente da mesma forma que uma expressão lambda! Em outras palavras, `Consumer<Usuario> tornaModerador = Usuario::tornaModerador` gera o mesmo consumidor que `Consumer<Usuario> tornaModerador = u -> u.tornaModerador()`. Vale frisar que aqui não há reflection sendo utilizada, tudo é resolvido em tempo de compilação, sem custos de overhead para a performance.

6.2 COMPARANDO DE UMA FORMA AINDA MAIS ENXUTA

Podemos usar *method reference* para deixar a implementação de uma comparação ainda mais simples e enxuta.

Vimos que podemos facilmente ordenar nossa lista de usuários usando o método `comparing` da classe `Comparator`, passando uma expressão lambda como parâmetro:

```
usuarios.sort(Comparator.comparing(u -> u.getNome()));
```

Vamos ver como esse código ficaria usando uma referência ao método `getNome`, no lugar da expressão `u -> u.getNome()`:

```
usuarios.sort(Comparator.comparing(Usuario::getNome));
```

Tiramos os parênteses do método, e o operador de seta do lambda. Agora nossa expressão parece estar ainda mais simples e um pouco mais legível, não acha?

Pensando na fluência do nosso código, poderíamos também fazer um import estático do método `comparing`, e extrair nossa expressão para uma variável com nome bem significativo, algo como:

```
Function<Usuario, String> byName = Usuario::getNome;  
usuarios.sort(comparing(byName));
```

6.3 COMPONDÓ COMPARATORS

Assim como ordenamos pelo nome, vimos que podemos ordenar os usuários pelos pontos:

```
usuarios.sort(Comparator.comparingInt(u -> u.getPontos()));
```

Utilizamos o `comparingInt` em vez do `comparing` para evitar o boxing desnecessário.

Usando a nova sintaxe, podemos fazer:

```
usuarios.sort(Comparator.comparingInt(Usuario::getPontos));
```

E se quisermos um critério de comparação mais elaborado? Por exemplo: ordenar pelos pontos e, no caso de empate, ordenar pelo nome.

Isso é possível graças a alguns métodos default existentes em `Comparator`, como o `thenComparing`. Vamos criar um `Comparator` que funciona dessa forma:

```
Comparator<Usuario> c = Comparator.comparingInt(Usuario::getPontos)  
    .thenComparing(Usuario::getNome);
```

Existem também variações desse método para evitar o boxing de primitivos, como `thenComparingInt`.

Podemos passar esse comparator direto para o `sort`:

```
usuarios.sort(Comparator.comparingInt(Usuario::getPontos)  
    .thenComparing(Usuario::getNome));
```

Há outro método que você pode aproveitar para compor comparators, porém passando o composto como argumento: o `nullsLast`.

```
usuarios.sort(Comparator.nullsLast(  
    Comparator.comparing(Usuario::getNome)));
```

Com isso, todos os usuários nulos da nossa lista estarão posicionados no fim, e o restante ordenado pelo nome! Há também o método estático `nullsFirst`.

comparator.reversed()

E se desejar ordenar por pontos, porém na ordem decrescente? Utilizamos o método default `reversed()` no `Comparator`:

```
usuarios.sort(Comparator.comparing(Usuario::getPontos).reversed());
```

Apesar do livro não ter como objetivo cobrir extensamente as novas APIs, é importante que você esteja inclinado a pesquisar bastante e experimentar um pouco dos novos métodos. Dessa forma, você não acaba escrevendo o que já existe ou invocando métodos estáticos em outras classes!

6.4 REFERENCIANDO MÉTODOS DE INSTÂNCIA

Dada uma referência a um objeto, podemos criar um *method reference* que invoque um de seus métodos:

```
Usuario rodrigo = new Usuario("Rodrigo Turini", 50);  
Runnable bloco = rodrigo::tornaModerador;  
bloco.run();
```

A invocação `bloco.run()` vai fazer com que `rodrigo.tornaModerador()` seja invocado. Para ficar mais nítido, os dois blocos a seguir são equivalentes:

```
Runnable bloco1 = rodrigo::tornaModerador;  
Runnable bloco2 = () -> rodrigo.tornaModerador();
```

Repare que isso é **bastante** diferente de quando fazemos `Usuario::tornaModerador`, pois estamos referenciando o método do meu usuário existente, e não de qualquer objeto do tipo `Usuario`.

Como enxergar melhor a diferença? O lambda `rodrigo::tornaModerador` pode ser inferido para um `Runnable`, pois esse método não recebe argumentos e deixamos claro de qual usuário é para ser invocado.

No caso do `Usuario::tornaModerador`, o compilador pode inferir o lambda para `Consumer<Usuario>` pois, apesar de `tornaModerador` não receber um argumento, é necessário saber de qual usuário estamos falando. Veja:

```
Usuario rodrigo = new Usuario("Rodrigo Turini", 50);
```

```
Consumer<Usuario> consumer = Usuario::tornaModerador;  
consumer.accept(rodrigo);
```

A chamada `consumer.accept(rodrigo)` acaba invocando `rodrigo.tornaModerador()`. O efeito é o mesmo, porém aqui precisamos passar o argumento. Novamente para reforçar, veja que os dois consumidores a seguir são equivalentes, um usando *method reference* e o outro usando lambda:

```
Consumer<Usuario> consumer1 = Usuario::tornaModerador;  
Consumer<Usuario> consumer2 = u -> u.tornaModerador();
```

O que não pode é misturar expectativas de número e tipos diferentes dos argumentos e retorno! Por exemplo, o Java não vai aceitar fazer `Runnable consumer = Usuario::tornaModerador`, pois esse *method reference* só pode ser atribuído a uma interface funcional que necessariamente receba um parâmetro em seu método abstrado, que seria utilizado na invocação de `tornaModerador`.

6.5 REFERENCIANDO MÉTODOS QUE RECEBEM ARGUMENTOS

Também podemos referenciar métodos que recebem argumentos, como por exemplo o `println` da instância `PrintStream out` da classe `java.lang.System`.

Para imprimir todos os itens de uma lista de usuários, podemos fazer um `forEach` que recebe como parâmetro uma referência ao método `println`:

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

Esse método compila, mas o que ele vai imprimir? Não estamos passando nenhum parâmetro para o método `println`!

Na verdade, implicitamente estamos. Quando escrevemos `System.out::println` temos um código equivalente a essa expressão lambda: `u -> System.out.println(u)`. Ou seja, o compilador sabe que ao iterar um uma lista de usuários, a cada iteração do método `forEach` teremos um objeto do tipo `Usuario`, e infere que esse é o parâmetro que deverá ser passado ao *method reference*.

Nosso código traduzido para Java 7, por exemplo, seria equivalente ao código a seguir:

```
for(Usuario u : usuarios) {  
    System.out.println(u);  
}
```

Para testar esse exemplo, sobrescreva o método `toString` da nossa classe `Usuario`:

```
public String toString() {  
    return "Usuario " + nome;  
}
```

Agora escreva a seguinte classe, que cria uma lista de usuários, e imprime o `toString` de todos os usuários.

```
public class Capitulo6 {  
    public static void main(String ... args) {  
        Usuario user1 = new Usuario("Paulo Silveira", 150);  
        Usuario user2 = new Usuario("Rodrigo Turini", 120);  
        Usuario user3 = new Usuario("Guilherme Silveira", 190);  
  
        List<Usuario> usuarios = Arrays.asList(user1, user2, user3);  
  
        usuarios.forEach(System.out::println);  
    }  
}
```

6.6 REFERENCIANDO CONSTRUTORES

Assim como métodos estáticos, podemos usar *method reference* com construtores. Por sinal, é comum ouvir esse tipo de referência ser chamada de *constructor reference*.

Neste caso, usamos o `new` para indicar que queremos referenciar um construtor, desta forma:

```
Usuario rodrigo = Usuario::new;
```

Ao tentar compilar esse código, recebemos a seguinte exception:

```
Capitulo6.java:11: error: incompatible types: Usuario is not a  
functional interface  
    Usuario rodrigo = Usuario::new;  
                           ^  
1 error
```

Claro, afinal, assim como as expressões lambda, precisamos guardar o resultado dessa referência em uma interface funcional! Vamos utilizar a interface `Supplier`, também presente no pacote `java.util.function`. Note a assinatura de seu método:

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

Ela tem a forma de uma *factory*. Podemos guardar a expressão `Usuario::new` em um `Supplier<Usuario>` e chamar seu método `get` sempre que for desejada a instanciação de um novo objeto do tipo `Usuario`:

```
Supplier<Usuario> criadorDeUsuarios = Usuario::new;  
Usuario novo = criadorDeUsuarios.get();
```

Utilizamos um `Supplier` sempre para criar um novo objeto a partir de seu construtor default. Caso queiramos criar a partir de algum construtor com argumento da classe, precisaremos utilizar alguma outra interface funcional. Considere o seguinte construtor que recebe uma `String` para atribuir ao nome do `Usuario`:

```
public Usuario(String nome) {  
    this.nome = nome;  
}
```

Precisamos de uma interface funcional que receba tanto o que será criado, que neste caso é o tipo `Usuario`, como também qual argumento será passado para o construtor! Podemos usar a interface que vimos no capítulo anterior, a `Function`!

Nosso código deve ficar assim:

```
Function<String, Usuario> criadorDeUsuarios = Usuario::new;
```

Pronto! Agora conseguimos criar novos usuários invocando seu único método abstrato, o `apply`:

```
Function<String, Usuario> criadorDeUsuarios = Usuario::new;  
Usuario rodrigo = criadorDeUsuarios.apply("Rodrigo Turini");  
Usuario paulo = criadorDeUsuarios.apply("Paulo Silveira");
```

Neste contexto, criar um usuário da forma antiga, dando um `new Usuario("Nome")`, é verdadeiramente mais simples, mas teremos mais para frente situações em que o *constructor reference* pode ajudar bastante.

Implementar o `criadorDeUsuarios` foi bem simples, não? Afinal, criamos um construtor que recebe apenas um parâmetro. Mas e se quisermos criar um usuário usando o construtor de dois parâmetros?

```
public Usuario(String nome, int pontos) {  
    this.pontos = pontos;  
    this.nome = nome;  
}
```

A API do Java já possui uma interface funcional que se encaixa perfeitamente nesse caso, a `BiFunction`.

Conseguimos utilizá-la para criar um usuário da seguinte forma:

```
BiFunction<String, Integer, Usuario> criadorDeUsuarios =  
    Usuario::new;  
Usuario rodrigo = criadorDeUsuarios.apply("Rodrigo Turini", 50);  
Usuario paulo = criadorDeUsuarios.apply("Paulo Silveira", 300);
```

Igualmente simples, não acha? Mas se eu quiser criar um usuário a partir de um construtor com 3 argumentos a API padrão não vai me ajudar, não existe nenhuma interface para esse caso! Não se preocupe, se fosse necessário poderíamos facilmente implementar uma `TriFunction` que resolveria o problema. Você verá que a API não possui todas as combinações, pois não seria possível com essa abordagem de lambda do Java 8.

REFERENCIANDO O CONSTRUTOR DE UM ARRAY

Também consigo usar *constructor reference* com um array, mas neste caso a sintaxe vai mudar um pouco. Basta adicionar os colchetes do array antes do delimitador `::`, por exemplo: `int []::new`.

6.7 OUTROS TIPOS DE REFERÊNCIAS

Não para por aí. Podemos referenciar um método sobreescrito da classe mãe. Neste caso, usamos a palavra reservada `super`, como em `super::toString`.

Também será útil poder se referenciar a métodos estáticos. Por exemplo, podemos atribuir `Math::max` a uma `BiFunction<Integer, Integer, Integer>`, pois ela recebe dois inteiros e devolve um inteiro.

Novamente o cuidado com o boxing

A atenção ao auto boxing desnecessário é constante. Assim como vimos no caso da `Function`, a `BiFunction` possui suas interfaces análogas para tipos primitivos.

Em vez de usar a `BiFunction`, poderíamos usar aqui uma `ToIntBiFunction<Integer, Integer>`, evitando o unboxing do retorno. Sim, é possível evitar todo boxing, usando uma `IntBinaryOperator`, que recebe dois `ints` e devolve um `int`.

Logo, as seguintes atribuições são todas possíveis:

```
BiFunction<Integer, Integer, Integer> max = Math::max;  
ToIntBiFunction<Integer, Integer> max2 = Math::max;  
IntBinaryOperator max3 = Math::max;
```

Qual usar? `IntBinaryOperator` certamente é mais interessante, mas vai depender do método que receberá nosso lambda como argumento. Normalmente nem paramos para pensar muito nisso, pois passaremos `Math::max` diretamente como argumento para algum método.

CAPÍTULO 7

Streams e Collectors

Ordenar uma coleção, que é um processo bem comum em nosso dia a dia, foi bastante melhorado com os novos recursos da linguagem. Mas como podemos tirar maior proveito desses recursos em outras situações comuns quando estamos trabalhando com uma coleção? Neste capítulo vamos mostrar os avanços da API de Collections de uma maneira bem prática!

7.1 TORNANDO MODERADORES OS 10 USUÁRIOS COM MAIS PONTOS

Para filtrar os 10 usuários com mais pontos e torná-los moderadores, podemos agora fazer o seguinte código:

```
usuarios.sort(Comparator.comparing(Usuario::getPontos).reversed());
usuarios
    .subList(0,10)
    .forEach(Usuario::tornaModerador);
```

Repare que isso já adianta bastante código! Antes do Java 8 seria necessário fazer algo como:

```
Collections.sort(usuarios, new Comparator<Usuario>() {  
    @Override  
    public int compare(Usuario u1, Usuario u2) {  
        return u1.getPontos() - u2.getPontos();  
    }  
});  
  
Collections.reverse(usuarios);  
List<Usuario> top10 = usuarios.subList(0, 10);  
for(Usuario usuario : top10) {  
    usuario.tornaModerador();  
}
```

Agora quero filtrar todos os usuários que têm mais de 100 pontos. Como eu faço?

7.2 STREAMS: TORNANDO MODERADORES OS USUÁRIOS COM MAIS DE 100 PONTOS

Como filtro uma coleção? Posso fazer um laço e, para cada elemento, usar um `if` para saber se devemos ou não executar uma tarefa.

Para tornar moderadores os usuários com mais de 100 pontos podemos fazer:

```
for(Usuario usuario : usuarios) {  
    if(usuario.getPontos() > 100) {  
        usuario.tornaModerador();  
    }  
}
```

Esse código está muito grande e imperativo. A qualquer mudança na filtragem ou na ação a ser executada teremos de encadear mais blocos de código. Em muitas linguagens, há um método `filter` nas diversas estruturas de dados.

Diferente do método `sort`, que existe em `List`, e `forEach`, que existe em `Iterable`, não há `filter` em `Iterable`! Nem em `Collection` ou `List`. Então onde acabou ficando definido esse método?

Ele poderia perfeitamente ter sido adicionado na API de `Collection`, mas sabemos o quanto essa API já é grande, e que seria cada vez mais difícil evoluí-la

sem haver uma quebra de compatibilidade. Estamos falando de uma API com mais de 15 anos!

Um outro ponto é que, segundo as listas de discussões, analisando o uso das coleções do Java foi possível identificar um padrão muito comum: as listas eram criadas, transformadas por diversas operações como o próprio filtro que pretendemos usar e, após essas transformações, tudo era resumido em uma única operação, como uma média ou soma dos valores tratados. O problema desse padrão identificado é que, na maior parte das vezes, ele exige a criação de variáveis pra guardar os valores intermediários desse processo todo.

Tendo essa análise como motivação e conhecendo as limitações de evoluir a API de `Collection`, o Java 8 introduziu o `Stream`. O `Stream` traz para o Java uma forma mais funcional de trabalhar com as nossas coleções, usando uma interface fluente! Separando as funcionalidades do `Stream` da `Collection`, também ficou mais fácil de deixar claro que métodos são mutáveis, evitar problema de conflito de nome de métodos, entre outros.

Mas como criar um `Stream` que represente os elementos da minha lista de usuários?

Um novo *default method* foi adicionado na interface `Collection`, o `stream()`:

```
Stream<Usuario> stream = usuarios.stream();
```

A partir desse nosso `Stream<Usuario>`, conseguimos utilizar o método `filter`. Ele recebe um parâmetro do tipo `Predicate<Usuario>`, com um único método de teste. É a mesma interface que usamos no `Collection.removeIf`.

Podemos passar uma expressão lambda para determinar o critério de filtro, já que `Predicate` é uma interface funcional:

```
Stream<Usuario> stream = usuarios.stream();
stream.filter(u -> {return u.getPontos() > 100});
```

Podemos remover o `return` e com isso as chaves não são mais necessárias, como já havíamos visto:

```
Stream<Usuario> stream = usuarios.stream();
stream.filter(u -> u.getPontos() > 100);
```

Claro, podemos ainda simplificar essa operação removendo a variável temporária:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100);
```

Você verá que a grande maioria dos métodos novos se encontra na família de interfaces do `Stream`. Dessa forma, as interfaces antigas não ficam cheias de métodos *default*. Além disso, há uma vantagem grande de se utilizar streams: evitar efeitos colaterais na coleção com que estamos trabalhando. Repare:

```
class Capitulo7 {
    public static void main (String... args) {
        Usuario user1 = new Usuario("Paulo Silveira", 150);
        Usuario user2 = new Usuario("Rodrigo Turini", 120);
        Usuario user3 = new Usuario("Guilherme Silveira", 90);

        List<Usuario> usuarios = Arrays.asList(user1, user2, user3);

        usuarios.stream()
            .filter(u -> u.getPontos() > 100);

        usuarios.forEach(System.out::println);
    }
}
```

E a saída desse código será:

```
Usuario Paulo Silveira
Usuario Rodrigo Turini
Usuario Guilherme Silveira
```

Por que na saída apareceu o `Guilherme Silveira`, sendo que ele não tem mais de 100 pontos? Ele não aplicou o filtro na lista de usuários! Isso porque o método `filter`, assim como os demais métodos da interface `Stream`, não alteram os elementos do `stream` original! **É muito importante saber que o `Stream` não tem efeito colateral sobre a coleção que o originou.**

Por isso, sempre que aplicamos uma transformação em um `Stream`, como fizemos com o `filter`, ele nos retorna um novo `Stream` com o resultado:

```
Stream<Usuario> stream = usuarios.stream()
    .filter(u -> u.getPontos() > 100);
```

Podemos alterar nossa classe `Capitulo7` para fazer o `forEach` desse novo `stream`:

```
class Capitulo7 {  
    public static void main (String... args) {  
        Usuario user1 = new Usuario("Paulo Silveira", 150);  
        Usuario user2 = new Usuario("Rodrigo Turini", 120);  
        Usuario user3 = new Usuario("Guilherme Silveira", 90);  
  
        List<Usuario> usuarios = Arrays.asList(user1, user2, user3);  
  
        Stream<Usuario> stream = usuarios  
            .stream()  
            .filter(u -> u.getPontos() > 100);  
  
        stream.forEach(System.out::println);  
    }  
}
```

Agora sim, ao executar esse código recebemos a saída:

```
Usuario Paulo Silveira  
Usuario Rodrigo Turini
```

O `Stream` então é uma outra coleção? Certamente não. A mais clara diferença é que um `Stream` nunca guarda dados. Ele não tem uma estrutura de dados interna para armazenar cada um dos elementos: ele na verdade usa uma coleção ou algum outro tipo de fonte para trabalhar com os objetos e executar uma série de operações (um *pipeline* de operações). Ele está mais próximo a um `Iterator`. O `Stream` é uma sequência de elementos que pode ser trabalhada de diversas formas.

Podemos encadear as invocações ao `Stream` de maneira fluente. Em vez de fazer:

```
Stream<Usuario> stream = usuarios  
    .stream()  
    .filter(u -> u.getPontos() > 100);  
  
stream.forEach(System.out::println);
```

Vamos utilizar o retorno do `filter` para encaixar diretamente o `forEach`:

```
usuarios.stream()  
    .filter(u -> u.getPontos() > 100)  
    .forEach(System.out::println);
```

A diferença é que `forEach` devolve `void`, então nesse ponto não temos mais a referência ao `Stream` resultante do `filter`. De qualquer maneira, o `Stream` é desenhado para que você utilize-o apenas uma vez. Caso queira realizar novas operações, você deve invocar `stream()` na coleção mais uma vez, obtendo um novo.

Voltemos à questão original. Para filtrar esses usuários e torná-los moderadores, fazemos assim:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .forEach(Usuario::tornaModerador);
```

Outro exemplo? Para filtrar os usuários que são moderadores, podemos fazer:

```
usuarios.stream()
    .filter(u -> u.isModerador());
```

Nesse caso, o `filter`, assim como qualquer um que recebe uma interface funcional e invoca apenas um método, pode também tirar proveito de *method references*.

```
usuarios.stream()
    .filter(Usuario::isModerador);
```

7.3 COMO OBTER DE VOLTA UMA LISTA?

O `forEach` é um método `void`. Se o `forEach` é `void`, e o `filter` devolve `Stream<T>`, quem devolve uma `List` caso eu precise?

Poderíamos fazer isso manualmente: criar uma lista que vai guardar os usuários que têm mais de 100 pontos e, depois de filtrar quem tem mais de 100 pontos, no `forEach` eu adiciono esses usuários a nova coleção:

```
List<Usuario> maisQue100 = new ArrayList<>();
```

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .forEach(u -> maisQue100.add(u));
```

Podemos simplificar, tirando proveito da sintaxe do *method reference* aqui:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .forEach(maisQue100::add);
```

Ainda assim é meio esquisito. Como isso é um trabalho que vai acontecer com frequência, há uma maneira mais simples de **coletar** os elementos de um Stream.

7.4 COLLECTORS

Podemos usar o método `collect` para resgatar esses elementos do nosso Stream<Usuario> para uma List. Porém, repare sua assinatura:

```
<R> R collect(Supplier<R> supplier,
                 BiConsumer<R, ? super T> accumulator,
                 BiConsumer<R, R> combiner);
```

Elá recebe três argumentos. Os três são interfaces funcionais. O primeiro é uma factory que vai criar o objeto que será devolvido no final da coleta. O segundo é o método que será invocado para adicionar cada elemento. O terceiro pode ser invocado se precisarmos adicionar mais de um elemento ao mesmo tempo (por exemplo, se formos usar uma estratégia de coletar elementos paralelamente, como veremos no futuro).

Para fazer essa transformação simples, eu teria que escrever um código como esse:

```
Supplier<ArrayList<Usuario>> supplier = ArrayList::new;
BiConsumer<ArrayList<Usuario>, Usuario> accumulator =
    ArrayList::add;
BiConsumer<ArrayList<Usuario>, ArrayList<Usuario>> combiner =
    ArrayList::addAll;

List<Usuario> maisQue100 = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(supplier, accumulator, combiner);
```

Bastante complicado, não acha? Poderíamos tentar simplificar deixando o código em um único *statement*, mas ainda sim teríamos uma operação complicada e perderíamos um pouco na legibilidade:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

A API simplificou esse trabalho, claro. Temos uma outra opção de uso do método `collect`, que recebe um `Collector` como parâmetro:

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

A interface `Collector` nada mais é que alguém que tem um `supplier`, um `accumulator` e um `combiner`. A vantagem é que existem vários `Collectors` prontos. Podemos simplificar bastante nosso código, passando como parâmetro em nosso método `collect` o `Collectors.toList()`, uma das implementações dessa nova interface.

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(Collectors.toList());
```

`Collectors.toList` devolve um coletor bem parecido com o qual criamos na mão, com a diferença de que ele devolve uma lista que não sabemos se é mutável, se é thread-safe ou qual a sua implementação.

Fazendo um import estático podemos deixar o código um pouco mais enxuto, em um único *statement*.

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(toList());
```

Pronto! Agora conseguimos coletar o resultado das transformações no nosso `Stream<Usuario>` em um `List<Usuario>`:

```
List<Usuario> maisQue100 = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(toList());
```

Apesar do import estático deixar nosso código mais enxuto, evitaremos abusar. O motivo é simples: para ficar mais claro durante a leitura do livro qual método estamos invocando.

7.5 AVANÇADO: POR QUE NÃO HÁ UM `toList` EM STREAM?

Esse tema foi diversas vezes discutido na lista do Java 8. Brian Goetz e outros líderes dão uma resposta parecida: há muitos, muitos métodos que cada pessoa gostaria que tivesse como *default method* em diversas interfaces. Se fossem aceitar toda sugestão interessante, a API ficaria ainda mais gigante. Você pode ler mais sobre essa decisão aqui:

<http://openjdk.5641.n7.nabble.com/easier-work-with-collections-default-Stream-toList-tt132.html>

Também podemos utilizar o método `toSet` para coletar as informações desse Stream em um `Set<Usuario>`:

```
Set<Usuario> maisQue100 = usuarios
    .stream()
    .filter(u -> u.getPontos() > 100)
    .collect(toSet());
```

Há ainda o método `toCollection`, que permite que você escolha a implementação que será devolvida no final da coleta:

```
Set<Usuario> set = stream.collect(
    Collectors.toCollection(HashSet::new));
```

O `toCollection` recebe um `Supplier<T>`, que já vimos. A interface `Supplier` é como uma factory, possuindo um único método (`get`) que não recebe argumento e devolve `T`. Relembrando method references com construtores, fazer `toCollection(HashSet::new)` é o mesmo que `toCollection(() -> new HashSet<Usuario>())` nesse caso.

Você também pode invocar `toArray` que devolve um array de `Object` em um `Stream`, ou invocar o `toArray` passando a forma de construir uma array (através de um `IntSupplier`). Um exemplo seria `Usuario[] array = stream.toArray(Usuario[]::new)`.

7.6 LISTE APENAS OS PONTOS DE TODOS OS USUÁRIOS COM O MAP

Com o seguinte código, não muito diferente do que faríamos com as versões pré-Java 8, conseguimos extrair uma lista com a pontuação de todos os usuários:

```
List<Integer> pontos = new ArrayList<>();
usuarios.forEach(u -> pontos.add(u.getPontos()));
```

Mas repare que é preciso criar uma variável intermediária com a lista, e adicionar os pontos manualmente. Ainda não estamos tirando muito proveito da nova API do Java 8! Além disso, o nosso lambda está causando efeitos colaterais: alterando

o estado de variáveis e objetos fora de seu escopo. Há algumas desvantagens nessa abordagem que entenderemos posteriormente.

Há uma forma bem mais interessante e conhecida de quem já tem algum conhecimento de programação funcional: o *map*.

Utilizando o método `map` da API de `Stream`, conseguimos obter o mesmo resultado aplicando uma transformação em minha lista sem a necessidade de variáveis intermediárias! Repare:

```
List<Integer> pontos = usuarios.stream()
    .map(u -> u.getPontos())
    .collect(toList());
```

E utilizando o `map`, podemos tirar proveito da sintaxe do *method reference*, simplificando ainda mais nossa operação:

```
List<Integer> pontos = usuarios.stream()
    .map(Usuario::getPontos)
    .collect(toList());
```

Repare que nem entramos em detalhes para dizer o que `map` recebe como argumento. Ele trabalha com `Function`, que é uma interface funcional. No nosso caso, dado um `Usuario` ele precisa devolver um `Integer`, definido pelo seu único método abstrato, o `apply`. É uma `Function<Usuario, Integer>`. Perceba que não foi necessário saber exatamente os argumentos, retorno e nome do método para leremos o código que usa o `map`, já que o lambda ajuda na legibilidade.

7.7 INTSTREAM E A FAMÍLIA DE STREAMS

Repare que, quando escrevemos o seguinte código, temos como retorno um `Stream<Integer>`:

```
Stream<Integer> stream = usuarios.stream()
    .map(Usuario::getPontos);
```

Isso gera o boxing dos nossos inteiros. Se formos operar sobre eles, teremos um overhead indesejado, que pode ser fatal para listas grandes ou caso haja muita repetição dessa instrução.

O pacote `java.util.stream` possui implementações equivalentes ao `Stream` para os principais tipos primitivos: `IntStream`, `LongStream`, e `DoubleStream`.

Podemos usar o `IntStream` aqui para evitar o autoboxing! Basta utilizarmos o método `mapToInt`:

```
IntStream stream = usuarios.stream()
    .mapToInt(Usuario::getPontos);
```

O `mapToInt` recebe uma função mais específica. Enquanto o `map` recebe uma `Function`, o `mapToInt` recebe `ToIntFunction`, interface que o método `apply` sempre retorna `int` e se chama `applyAsInt`.

No `IntStream`, existem métodos que simplificam bastante nosso trabalho quando estamos trabalhando com inteiros, como `max`, `sorted` e `average`.

Observe como ficou mais simples obter a média de pontos dos usuários:

```
double pontuacaoMedia = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average()
    .getAsDouble();
```

Veremos depois que o `average` é um caso especial da chamada redução.

Nas versões anteriores da linguagem, teríamos que iterar pelos `usuarios` somando suas pontuações em uma variável temporária, para somente no final dividir essa soma pela quantidade de elementos da lista. Algo como:

```
double soma = 0;
for (Usuario u : usuarios) {
    soma += u.getPontos();
}
double pontuacaoMedia = soma / usuarios.size();
```

Você verá que a operação de `map` é utilizada com bastante frequência, direta ou indiretamente.

7.8 O OPTIONAL EM JAVA.UTIL

Utilizamos `average().getAsDouble()` para receber o `double` da média. O que será que esse `average()` devolve? E por que não um `double?`

É para tentar evitar aqueles padrões de verificação de casos extremos. Por exemplo:

```

double soma = 0;
for (Usuario u : usuarios) {
    soma += u.getPontos();
}
double pontuacaoMedia = soma / usuarios.size();

```

O que acontece se o número de usuários for zero? Teremos como resultado o positivo infinito! Era isso que gostaríamos? Caso não, vamos adicionar um `if`, para retornar zero:

```

double soma = 0;
for (Usuario u : usuarios) {
    soma += u.getPontos();
}

double pontuacaoMedia;
if (usuarios.isEmpty()) {
    pontuacaoMedia = 0;
}
else {
    pontuacaoMedia = soma / usuarios.size();
}

```

O Java 8 introduz a classe `java.util.Optional`, que era muito esperada e já existia em outras linguagens. Ela permite que possamos cuidar desses casos de uma maneira mais simples, utilizando uma interface fluente. Há também algumas versões primitivas, como `OptionalDouble` e `OptionalInt` que ajudam nesses casos. O `average`, que só existe em streams primitivos, devolve um `OptionalDouble`. Repare:

```

OptionalDouble media = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average();

double pontuacaoMedia = media.orElse(0.0);

```

Pronto. Se a lista for vazia, o valor de `pontuacaoMedia` será `0.0`. Sem o uso do `orElse`, ao invocar o `get` você receberia um `NoSuchElementException`, indicando que o `Optional` não possui valor definido.

Podemos escrever em uma única linha:

```
double pontuacaoMedia = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average()
    .orElse(0.0);
```

Ou ir ainda além, como lançar uma exception utilizando o método `orElseThrow`. Ele recebe um `Supplier` de exceptions, aquela interface funcional que parece bastante uma factory. Podemos então fazer:

```
double pontuacaoMedia = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average()
    .orElseThrow(IllegalStateException::new);
```

Podemos verificar o contrário: se realmente existe um valor dentro do opcional. E, no caso de existir, passamos um `Consumer` como argumento:

```
usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average()
    .ifPresent(valor -> janela.atualiza(valor));
```

Há diversos outros métodos nos *optionals* e voltaremos a utilizá-los, pois os streams costumam trabalhar bastante com eles.

O caso mais frequente é quando um valor pode ser `null`. Com o `Optional`, você será sempre obrigado a trabalhar com a condição de aquele elemento não existir, evitando um `NullPointerException` descuidado.

Por exemplo: queremos o usuário com maior quantidade de pontos. Podemos usar o método `max` para tal, que recebe um `Comparator`:

```
Optional<Usuario> max = usuarios
    .stream()
    .max(Comparator.comparingInt(Usuario::getPontos));
```

Se a lista for vazia, não haverá usuário para ser retornado. Por isso, o resultado é um `Optional`. Você deve verificar se há ou não um usuário presente nesse resultado, usando os métodos que vimos. Pode ser um simples `get`, podendo receber um `null`, ou algo mais rebuscado com o `orElse` ou `ifPresent`.

Você pode até mesmo continuar trabalhando com `Optional` de maneira *lazy*. Se você quiser o nome do usuário com maior número de pontos, pode *mapear* (transformar) esse resultado:

```
Optional<String> maxNome = usuarios
    .stream()
    .max(Comparator.comparingInt(Usuario::getPontos))
    .map(u -> u.getNome());
```

Sim, você poderia usar o method reference `Usuario::getNome` aqui.

CAPÍTULO 8

Mais operações com Streams

Trabalhar com streams vai fazer parte do seu dia a dia. Conhecer bem sua API através de exemplos práticos é uma excelente forma de aprender mais conceitos que ela envolve.

8.1 ORDENANDO UM STREAM

Dada uma `List<Usuario>` `usuarios`, sabemos como podemos ordená-la por nome:

```
usuarios.sort(Comparator.comparing(Usuario::getNome));
```

E um stream? Imagine que queremos filtrar os usuários com mais de 100 pontos e aí ordená-los:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .sorted(Comparator.comparing(Usuario::getNome));
```

No stream, o método de ordenação é o `sorted`. A diferença entre ordenar uma lista com `sort` e um stream com `sorted` você já deve imaginar: um método invocado em `Stream` não altera quem o gerou. No caso, ele não altera a `List<Usuario>` `usuarios`. Se quisermos o resultado em uma `List`, precisamos usar um coletor, como visto:

```
List<Usuario> filtradosOrdenados = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .sorted(Comparator.comparing(Usuario::getNome))
    .collect(Collectors.toList());
```

Um código fácil de ler e bastante enxuto. Talvez você não esteja se lembrando de como precisaria escrever muito mais linhas no Java 7, perdendo um pouco da clareza e simplicidade:

```
List<Usuario> usuariosFiltrados = new ArrayList<>();
for(Usuario usuario : usuarios) {
    if(usuario.getPontos() > 100) {
        usuariosFiltrados.add(usuario);
    }
}

Collections.sort(usuariosFiltrados, new Comparator<Usuario>() {
    public int compare(Usuario u1, Usuario u2) {
        return u1.getNome().compareTo(u2.getNome());
    }
});
```

É necessária uma lista temporária para a filtragem, um laço para esse mesmo filtro, uma classe anônima para o `Comparator` e finalmente a invocação para a ordenação.

8.2 MUITAS OPERAÇÕES NO STREAM SÃO LAZY!

Quando manipulamos um `Stream`, normalmente encadeamos diversas operações computacionais. Esse conjunto de operações realizado em um `Stream` é conhecido como *pipeline*. O Java pode tirar proveito dessa estrutura para otimizar as operações. Como ele faz isso? Evitando executar as operações o máximo possível: grande parte delas são *lazy* e executam realmente só quando necessário para obter o resultado final.

Um exemplo? Pense no código anterior, antes do `collect`:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .sorted(Comparator.comparing(Usuario::getNome));
```

Os métodos `filter` e `sorted` devolvem um `Stream`. No momento dessas invocações, esses métodos nem filtram, nem ordenam: eles apenas devolvem novos streams em que essa informação é marcada. Esses métodos são chamados de **operações intermediárias**. Os novos streams retornados sabem que devem ser filtrados e ordenados (ou o equivalente) no momento em que uma **operação terminal** for invocada.

O `collect` é um exemplo de operação terminal e só nesse momento o stream realmente vai começar a executar o *pipeline* de operações pedido.

8.3 QUAL É A VANTAGEM DOS MÉTODOS SEREM LAZY?

Imagine que queremos encontrar um usuário com mais de 100 pontos. Basta um e serve qualquer um, desde que cumpra o predicado de ter mais de 100 pontos.

Podemos filtrar o stream, coletá-lo em uma lista e pegar o primeiro elemento:

```
Usuario maisDe100 = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .collect(Collectors.toList())
    .get(0);
```

É muito trabalho para algo simples: aqui filtramos todos os usuários e criamos uma nova coleção com todos eles apenas para pegar o primeiro elemento. Além disso, no caso de não haver nenhum usuário com mais de 100 pontos, receberemos uma exception.

O `Stream` possui o método `findAny` que devolve qualquer um dos elementos:

```
Optional<Usuario> usuarioOptional = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .findAny();
```

Há duas vantagens aqui em trabalhar com `Stream`. A primeira é que o `findAny` devolve um `Optional<Usuario>` e com isso somos obrigados a fazer um `get` para receber o `Usuario`, ou usar os métodos de teste (como `orElse` e

`isPresent`) para saber se realmente há um usuário filtrado aí. A segunda vantagem é que, como todo o trabalho foi lazy, o stream **não foi inteiramente filtrado**.

É isso mesmo. O `findAny` é uma operação terminal e forçou a execução do *pipeline* de operações. Ao mesmo tempo, esse método é escrito de maneira inteligente, analisando as operações invocadas anteriormente e percebendo que não precisa filtrar todos os elementos da lista para pegar apenas um deles que cumpra o predicado dado. Ele começa a executar o filtro e, assim que encontrar um usuário com mais de 100 pontos, retorna-o e termina a filtragem ali mesmo. Outro método que pode ser útil é o `findFirst`, similar ao `findAny`, mas utilizando os elementos na ordem percorrida pelo stream.

Essa técnica de otimização nem sempre é possível de ser aplicada, dependendo do *pipeline* de operações. Portanto, você não deve ter isso como garantia para escrever código possivelmente ineficiente.

Como ver que a otimização realmente acontece?

Você poderia colocar um breakpoint no método `getPontos` ou imprimir uma informação de log quando ele for invocado. Você veria que ele será invocado apenas a quantidade de vezes necessária até encontrar algum `Usuario` com mais de 100 pontos! Mas há uma forma mais interessante: utilizando o método intermediário `peek`.

8.4 ENXERGANDO A EXECUÇÃO DO PIPELINE COM PEEK

Podemos pedir para que o stream execute um tarefa toda vez que processar um elemento. Fazemos isso através do `peek`:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .peek(System.out::println)
    .findAny();
```

Podemos ver que só serão mostrados os elementos até que seja encontrado algum elemento que cumpra o predicado `u.getPontos() > 100`. Assim, fica claro o poder que o lazyness tem na API de `Stream`.

Bem diferente de um `forEach`, que devolve `void` e é uma operação terminal, o `peek` devolve um novo `Stream` e é uma operação intermediária. Ele não forçará a execução do *pipeline*. O seguinte código simplesmente não imprime nada:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .peek(System.out::println);
```

Por quê? Pois o `peek` devolve um novo `Stream`, onde está marcado para imprimir todos os elementos processados. Ele só vai processar elementos quando encontrar uma operação terminal, como o `findAny`, o `collect` ou o `forEach`.

Com o `peek`, podemos ver se outras operações conseguem tirar vantagem do lazyness. Experimente fazer o mesmo truque com o `sorted`:

```
usuarios.stream()
    .sorted(Comparator.comparing(Usuario::getNome))
    .peek(System.out::println)
    .findAny();
```

Aqui o `peek` imprime todos os usuários, mesmo se só queremos fazer `findAny`. Dizemos que o `sorted` é um método intermediário *stateful*. Operações *stateful* podem precisar processar todo o stream mesmo que sua operação terminal não demande isso.

8.5 OPERAÇÕES DE REDUÇÃO

Operações que utilizam os elementos da stream para retornar um valor final são frequentemente chamadas de operações de redução (*reduction*). Um exemplo é o `average`, que já havíamos visto:

```
double pontuacaoMedia = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .average()
    .getAsDouble();
```

Há outros métodos úteis como o `average`: o `count`, o `min`, o `max` e o `sum`. Esse último, como o `average`, encontra-se apenas nos streams primitivos. O `min` e o `max` pedem um `Comparator` como argumento. Todos, com exceção do `sum` e `count`, trabalham com `Optional`. Por exemplo:

```
Optional<Usuario> max = usuarios.stream()
    .max(Comparator.comparing(Usuario::getPontos));
Usuario maximaPontuacao = max.get();
```

Se desejarmos somar todos os pontos dos usuários, fazemos:

```
int total = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .sum();
```

Essa soma é executada através de uma operação de redução que podemos deixar bem explícita. Como ela funciona? Ela pega o primeiro elemento, que é a pontuação do primeiro usuário do stream, e guarda o valor acumulado até então, com uma operação de soma. Também precisamos ter um valor inicial que, para o caso da soma, é zero.

Podemos quebrar essa operação de soma para enxergar melhor o que é uma operação de redução. Repare nas definições:

```
int valorInicial = 0;
IntBinaryOperator operacao = (a, b) -> a + b;
```

`IntBinaryOperator` é uma interface funcional que define o método `applyAsInt`, que recebe dois inteiros e devolve um inteiro. A soma é assim, não é mesmo?

Com essas definições, podemos pedir para que o stream processe a redução, passo a passo:

```
int valorInicial = 0;
IntBinaryOperator operacao = (a, b) -> a + b;

int total = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .reduce(valorInicial, operacao);
```

Pronto. Temos um código equivalente ao `sum`. Poderíamos ter escrito tudo mais sucintamente, sem a declaração de variáveis locais:

```
int total = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .reduce(0, (a, b) -> a + b);
```

Podemos ir além. Na classe `Integer`, há agora o método estático `Integer.sum`, que soma dois inteiros. Em vez do lambda, podemos usar um *method reference*:

```
int total = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .reduce(0, Integer::sum);
```

Qual é a vantagem de usarmos a redução em vez do `sum`? Nenhuma. O importante é conhecê-lo para poder realizar operações que não se encontram no `Stream`. Por exemplo? Multiplicar todos os pontos:

```
int multiplicacao = usuarios.stream()
    .mapToInt(Usuario::getPontos)
    .reduce(1, (a,b) -> a * b);
```

Há também alguns casos especiais em que invocar o `map` pode ser custoso, e o melhor seria fazer a operação de soma diretamente. Esse não é o nosso caso, mas só para enxergarmos o exemplo, a soma sem o `map` ficaria assim:

```
int total = usuarios.stream()
    .reduce(0, (atual, u) -> atual + u.getPontos(), Integer::sum);
```

Esse overload do `reduce` recebe mais um lambda que, no nosso caso, é o `Integer::sum`. Esse lambda a mais serve para combinar os valores de reduções parciais, no caso de streams paralelos.

Não há necessidade de querer sempre evitar o número de operações em um `stream`. São casos isolados em que tiraremos proveito dessa abordagem.

Em outras linguagens, essa operação também é conhecida como *fold*. Veremos, no próximo capítulo, que essas operações são paralelizáveis e tiram grande proveito dos streams paralelos. Também repararemos que o `collect` é um caso de redução.

8.6 CONHECENDO MAIS MÉTODOS DO STREAM

São muitas as funcionalidades do `Stream`. O livro não tem intenção de cobrir todos os métodos da API, mas podemos passar rapidamente por alguns dos mais interessantes, além dos que já vimos por aqui.

Além disso, conheceremos mais alguns detalhes sobre o bom uso dos streams.

Trabalhando com iterators

Vimos que podemos coletar o resultado de um pipeline de operações de um `Stream` em uma coleção com o `collect`. Algumas vezes nem precisamos de uma coleção: bastaria iterarmos pelos elementos de um `Stream`.

O que será que acontece se tentarmos percorrer o `Stream` da maneira antiga?

```
for (Usuario u : usuarios.stream()){
    //...
}
```

Ocorre um erro de compilação. O *enhanced for* espera ou uma array ou um `Iterable`.

Mas por que decidiram fazer com que `Stream` não seja um `Iterable`? Pois as diversas operações terminais de um `Stream` o marcam como já utilizado. Sua segunda invocação lança um `IllegalStateException`. Vale lembrar que se você invocar duas vezes `usuarios.stream()` não haverá problema, pois abrimos dois streams diferentes a cada invocação!

Porém, podemos percorrer os elementos de um `Stream` através de um `Iterator`. Para isso, podemos invocar o método `iterator`:

```
Iterator<Usuario> i = usuarios.stream().iterator();
```

A interface `Iterator` já existe há bastante tempo no Java e define os métodos `hasNext`, `next` e `remove`. Com o Java8, também podemos percorrer um iterator utilizando o método `forEachRemaining` que recebe um `Consumer` como parâmetro:

```
usuarios.stream().iterator()
    .forEachRemaining(System.out::println);
```

Mas quando devemos utilizar um iterator de um `Stream`, se há um `forEach` tanto em `Collection` quanto no próprio `Stream`?

Um motivo para usar um `Iterator` é quando queremos modificar os objetos de um `Stream`. Quando utilizarmos streams paralelos, veremos que não devemos mudar o estado dos objetos que estão nele, correndo o risco de ter resultados não determinísticos. Outro motivo é a compatibilidade de APIs. Pode ser que você precise invocar um método que recebe `Iterator`.

Testando Predicates

Vimos bastante o uso do `filter`. Ele recebe um lambda como argumento, que é da interface `Predicate`. Há outras situações em que queremos testar predicados mas não precisamos da lista filtrada. Por exemplo, se quisermos saber se há algum elemento daquela lista de usuários que é moderador:

```
boolean hasModerator = usuarios.stream()
    .anyMatch(Usuario::isModerador);
```

Aqui o `Usuario::isModerador` tem o mesmo efeito que `u -> u.isModerador()`, gerando um predicado que testa se um usuário é moderador e devolve um booleano. O processamento dessa operação vai parar assim que o stream encontrar algum usuário que é moderador.

Assim como o `anyMatch`, podemos descobrir se todos os usuários são moderadores com `allMatch` ou se nenhum deles é, com o `noneMatch`.

Há muitos outros métodos e detalhes!

Você pode utilizar o `count` para saber quantos elementos há no `Stream`, `skip` para pular os `n` próximos elementos e `limit` para cortar o número de elementos.

Também há formas de você criar um `Stream` sem a existência de uma coleção. Na própria interface `Stream` há os métodos estáticos `empty` e `of`. O primeiro claramente cria um `Stream` vazio, e o `of` depende do que você passar como argumento, como por exemplo `Stream.of(user1, user2, user3)` retorna um `Stream<Usuario>`. Você também pode concatená-los com `Stream.concat`.

Fora da própria interface também podemos produzir `Streams`. Por exemplo, você pode usar a API de regex para devolver um `Stream<String>` através do `Pattern.splitAsStream`, ou ainda pegar um `Stream` das linhas de um arquivo com o `Files.lines`. Se estiver trabalhando diretamente com um array, pode usar o `Arrays.stream`.

Não deixe de conhecer a API e investigar bastante essa importante interface sempre que tiver novas ideias e sentir falta de algum método: ele provavelmente existe.

Outro ponto relevante: como alguns `Streams` podem ser originados de recursos de IO, ele implementa a interface `AutoCloseable` e possui o `close`. Um exemplo é usar os novos métodos da classe `java.nio.file.Files`, incluída no Java 7 e com novidades no Java 8, para gerar um `Stream`. Nesse caso, é fundamental tratar as exceções e o `finally`, ou então usar o *try with resources*. Em outras situações, quando sabemos que estamos lidando com streams gerados por coleções, essa preocupação não é necessária.

8.7 STREAMS PRIMITIVOS E INFINITOS

Assim como vimos com o `comparingInt`, que devolve um `Comparator` que não tem a necessidade de fazer o unboxing, vimos também o caso do `mapToInt`, que devolve um `IntStream`, também para evitar operações desnecessárias.

O `IntStream`, o `LongStream` e o `DoubleStream` possuem operações especiais e que são importantes. Até mesmo o iterator deles devolvem `Iterators` diferentes. No caso do `IntStream`, é o `PrimitiveIterator.OfInt`, que implementa `Iterator<Integer>` mas que, além de um `next` que devolve um `Integer` fazendo o boxing, também possui o `nextInt`.

Tem ainda métodos de factory, como `IntStream.range(inicio, fim)`.

Streams infinitos

Um outro recurso poderoso do Java 8: através da interface de factory `Supplier`, podemos definir um `Stream` infinito, bastando dizer qual é a regra para a criação de objetos pertencentes a esse `Stream`.

Por exemplo, se quisermos gerar uma lista “infinita” de números aleatórios, podemos fazer assim:

```
Random random = new Random(0);
Supplier<Integer> supplier = () -> random.nextInt();
Stream<Integer> stream = Stream.generate(supplier);
```

O `Stream` gerado por `generate` é *lazy*. Certamente ele não vai gerar infinitos números aleatórios. Eles só serão gerados à medida que forem necessários.

Aqui estamos gerando o boxing o tempo todo. Podemos usar o `IntSupplier` e o `IntStream`. Além disso, removeremos a variável temporária `supplier`:

```
Random random = new Random(0);
IntStream stream = IntStream.generate(() -> random.nextInt());
```

Agora precisamos de cuidado. Qualquer operação que necessite passar por todos os elementos do `Stream` nunca terminará de executar. Por exemplo:

```
int valor = stream.sum();
```

Você pode apenas utilizar operações de curto-circuito em `Streams` infinitos.

Operações de curto circuito

São operações que não precisam processar todos os elementos. Um exemplo seria pegar apenas os 100 primeiros elementos com `limit`:

```
Random random = new Random(0);
IntStream stream = IntStream.generate(() -> random.nextInt());
```

```
List<Integer> list = stream
    .limit(100)
    .boxed()
    .collect(Collectors.toList());
```

Repare a invocação de `boxed`. Ele retorna um `Stream<Integer>` em vez do `IntStream`, possibilitando a invocação a `collect` da forma que já vimos. Sem isso, teríamos apenas a opção de fazer `IntStream.toArray`, ou então de chamar o `collect` que recebe três argumentos, mas não teríamos onde guardar os números. Não foi criado no Java um `IntList` que seria o análogo primitivo a `List<Integer>`, e também não entraram no Java 8 os tais dos *value objects* ou *value types*, que possibilitariam algo como `List<int>`.

Vamos rever o mesmo código com a interface fluente:

```
Random random = new Random(0);
List<Integer> list = IntStream
    .generate(() -> random.nextInt())
    .limit(100)
    .boxed()
    .collect(Collectors.toList());
```

O `Supplier` passado ao `generate` pode servir para gerar um `Stream` infinito de constantes, por exemplo `IntStream.generate(() -> 1)` e `Stream.generate(() -> new Object())`.

Pode ser útil para um `Supplier` manter estado. Nesse caso, precisamos usar uma classe ou classe anônima, pois dentro de um lambda não podemos declarar atributos. Vamos gerar a sequência infinita de números de Fibonacci de maneira *lazy* e imprimir seus 10 primeiros elementos:

```
class Fibonacci implements IntSupplier {
    private int anterior = 0;
    private int proximo = 1;

    public int getAsInt() {
        proximo = proximo + anterior;
        anterior = proximo - anterior;
        return anterior;
    }
}
```

```
IntStream.generate(new Fibonacci())
    .limit(10)
    .forEach(System.out::println);
```

Veremos que manter o estado em uma interface funcional pode limitar os recursos de paralelização que um `Stream` fornece.

Além do `limit`, há outras operações que são de curto-circuito. O `findFirst` é uma delas. Mas não queremos pegar o primeiro elemento Fibonacci. Quero pegar o primeiro elemento maior que 100! Como fazer? Podemos filtrar antes de invocar o `findFirst`:

```
int maiorQue100 = IntStream
    .generate(new Fibonacci())
    .filter(f -> f > 100)
    .findFirst()
    .getAsInt();
```

```
System.out.println(maiorQue100);
```

O `filter` não é de curto-circuito: ele não produz um `Stream` finito dado um `Stream` infinito. Basta apenas que você tenha uma operação de curto-circuito no pipeline (seja a operação intermediária ou final), que você terá chances de que a execução do pipeline não tome tempo infinito. Por que digo chances? Pois, por exemplo, se não houvesse um elemento de Fibonacci maior que 100, isso também rodaria indefinidamente.

Os matchers também são de curto-circuito. Podemos tentar descobrir se todos os elementos de Fibonacci são pares com `allMatch(f -> f % 2 == 0)`. Se houver algum ímpar, ele retornará falso. Mas se houvesse apenas pares, ele rodaria indefinidamente! Lembre-se: trabalhar com `Streams` infinitos pode ser perigoso, mesmo que você utilize operações de curto-circuito.

Quando for necessário manter o estado de apenas uma variável, podemos usar o `iterate` em vez do `generate`, que recebe um `UnaryOperator`. Para gerar os números naturais:

```
IntStream.iterate(0, x -> x + 1)
    .limit(10)
    .forEach(System.out::println);
```

Há modificações também na API antiga para trabalhar com `Streams` infinitos. Você pode ver que a classe `java.util.Random` já devolve `Streams` infinitos, através de métodos como `Random.ints()`.

8.8 PRATICANDO O QUE APRENDEMOS COM JAVA.NIO.FILES

A classe `java.nio.file.Files` entrou no Java 7 para facilitar a manipulação de arquivos e diretórios, trabalhando com a interface `Path`. É uma das classes que agora possuem métodos para trabalhar com `Stream`. Excelente oportunidade para praticarmos boa parte do que aprendemos.

Se quisermos listar todos os arquivos de um diretório, basta pegar o `Stream<Path>` e depois um `forEach`:

```
Files.list(Paths.get("./br/com/casadocodigo/java8"))
    .forEach(System.out::println);
```

Quer apenas os arquivos `java`? Pode usar um `filter`:

```
Files.list(Paths.get("./br/com/casadocodigo/java8"))
    .filter(p -> p.toString().endsWith(".java"))
    .forEach(System.out::println);
```

E se quisermos todo o conteúdo dos arquivos? Vamos tentar usar o `Files.lines` para ler todas as linhas de cada arquivo.

```
Files.list(Paths.get("./br/com/casadocodigo/java8"))
    .filter(p -> p.toString().endsWith(".java"))
    .map(p -> Files.lines(p))
    .forEach(System.out::println);
```

Infelizmente esse código não compila. O problema é que `Files.lines` lança `IOException`. Mesmo que o método que invoca o `map` lance essa exception, não compilará, pois nesse caso é a implementação do lambda que estará lançando `IOException`. O `map` recebe uma `Function`, que tem o método `apply` e que não lança exception alguma na assinatura.

Uma solução seria escrever uma classe anônima ou um lambda definido com as chaves e com `try/catch` por dentro. Outra seria fazer um método estático simples, que faz o `wrap` da chamada para evitar a checked exception:

```
static Stream<String> lines(Path p) {
    try {
        return Files.lines(p);
    } catch(IOException e) {
```

```
        throw new UncheckedIOException(e);
    }
}
```

Em vez de invocarmos `map(p -> Files.lines(p))`, invocamos o nosso próprio `lines`, que não lança checked exception:

```
Files.list(Paths.get("./br/com/casadocodigo/java8"))
    .filter(p -> p.toString().endsWith(".java"))
    .map(p -> lines(p))
    .forEach(System.out::println);
```

Agora o código compila. A saída é algo como:

```
java.util.stream.ReferencePipeline$Head@5b6f7412
java.util.stream.ReferencePipeline$Head@27973e9b
java.util.stream.ReferencePipeline$Head@312b1dae
java.util.stream.ReferencePipeline$Head@7530d0a
...
...
```

O problema é que, com esse `map`, teremos um `Stream<Stream<String>>`, pois a invocação de `lines(p)` devolve um `Stream<String>` para cada `Path` do nosso `Stream<Path>` original! Isso fica mais claro de observar se não usarmos o `forEach` e atribuirmos o resultado a uma variável:

```
Stream<Stream<String>> strings =
    Files.list(Paths.get("./br/com/casadocodigo/java8"))
        .filter(p -> p.toString().endsWith(".java"))
        .map(p -> lines(p));
```

8.9 FLATMAP

Podemos *achatar* um `Stream` de `Streams` com o `flatMap`. Basta trocar a invocação, que teremos no final um `Stream<String>`:

```
Stream<String> strings =
    Files.list(Paths.get("./br/com/casadocodigo/java8"))
        .filter(p -> p.toString().endsWith(".java"))
        .flatMap(p -> lines(p));
```

Isso pode ser encadeado em vários níveis. Para cada `String` podemos invocar `String.chars()` e obter um `IntStream` (definiram assim para evitar o boxing para `Stream<Character>`). Se fizermos `map(s -> s.chars())`, obtémos um indesejado `Stream<IntStream>`. Precisamos passar esse lambda para o `flatMapToInt`:

```
IntStream chars =
    Files.list(Paths.get("./br/com/casadocodigo/java8"))
        .filter(p -> p.toString().endsWith(".java"))
        .flatMap(p -> lines(p))
        .flatMapToInt((String s) -> s.chars());
```

O `IntStream` resultante possui todos os caracteres de todos os arquivos `java` do nosso diretório.

Mais um exemplo de flatMap

Quando trabalhamos com coleções de coleções, usamos o `flatMap` quando queremos que o resultado da nossa transformação seja reduzido a um `Stream` ‘simples’, sem composição.

Imagine que temos grupos de usuários:

```
class Grupo {
    private Set<Usuario> usuarios = new HashSet<>();

    public void add(Usuario u) {
        usuarios.add(u);
    }

    public Set<Usuario> getUsuarios() {
        return Collections.unmodifiableSet(this.usuarios);
    }
}
```

E que tenhamos alguns grupos, separando quem fala inglês e quem fala espanhol:

```
Grupo englishSpeakers = new Grupo();
englishSpeakers.add(user1);
englishSpeakers.add(user2);

Grupo spanishSpeakers = new Grupo();
```

```
spanishSpeakers.add(user2);
spanishSpeakers.add(user3);
```

Se temos esses grupos dentro de uma coleção:

```
List<Grupo> groups = Arrays.asList(englishSpeakers, spanishSpeakers);
```

Pode ser que queiramos todos os usuários desses grupos. Se fizermos um simples `groups.stream().map(g -> g.getUsuarios().stream())`, teremos um `Stream<Stream<Usuario>>`, que não desejamos. O `flatMap` vai desembuchar esses Streams, achatando-os.

```
groups.stream()
    .flatMap(g -> g.getUsuarios().stream())
    .distinct()
    .forEach(System.out::println);
```

Temos como resultado todos os usuários de ambos os grupos, sem repetição. Se tivéssemos coletado o resultado do *pipeline* em um `Set`, não precisaríamos do `distinct`.

Um outro exemplo de uso de `flatMap`? Se nossos `Usuarios` possuísem `List<Pedido> pedidos`, chamar o `usuarios.map(u -> u.getPedidos())` geraria um `Stream<List<Pedido>>`. Se você tentar fazer `usuarios.map(u -> u.getPedidos().stream())`, vai cair no `Stream<Stream<Pedido>>`. A resposta para obter um `Stream<Pedido>` com os pedidos de todos os usuários da nossa lista é fazer `usuarios.flatMap(u -> u.getPedidos().stream())`.

CAPÍTULO 9

Mapeando, particionando, agrupando e paralelizando

Podemos realizar tarefas mais complexas e interessantes com o uso de streams e coletores. Vamos ver códigos mais elaborados e necessidades que aparecem com frequência.

9.1 COLETORES GERANDO MAPAS

Vimos como gerar um `Stream` com todas as linhas dos arquivos de determinado diretório:

```
Stream<String> strings =
    Files.list(Paths.get("./br/com/casadocodigo/java8"))
        .filter(p -> p.toString().endsWith(".java"))
        .flatMap(p -> lines(p));
```

Poderíamos ter um `Stream` com a quantidade de linhas de cada arquivo. Para isso, em vez de fazer um `flatMap` para as linhas, fazemos um `map` para a quantidade de linhas, usando o `count` do `Stream`:

```
LongStream lines =  
    Files.list(Paths.get("./br/com/casadocodigo/java8"))  
        .filter(p -> p.toString().endsWith(".java"))  
        .mapToLong(p -> lines(p).count());
```

Se quisermos uma `List<Long>` com os valores desse `LongStream`, fazemos um `collect` como já conhecemos.

```
List<Long> lines =  
    Files.list(Paths.get("./br/com/casadocodigo/java8"))  
        .filter(p -> p.toString().endsWith(".java"))  
        .map(p -> lines(p).count())  
        .collect(Collectors.toList());
```

Um detalhe: repare que poderíamos usar o `mapToLong` e depois invocar o `boxed` para fazer o `collect`, mas usamos o `map`, que já retorna `Stream<Long>` em vez de `LongStream`.

De qualquer maneira, o resultado não parece muito útil: um monte de `longs`. O que precisamos com mais frequência é saber quantas linhas tem cada arquivo, por exemplo. Podemos fazer um `forEach` e popular um `Map<Path, Long>`, no qual a chave é o arquivo e o valor é a quantidade de linhas daquele arquivo:

```
Map<Path, Long> linesPerFile = new HashMap<>();  
  
Files.list(Paths.get("./br/com/casadocodigo/java8"))  
    .filter(p -> p.toString().endsWith(".java"))  
    .forEach(p ->  
        linesPerFile.put(p, lines(p).count()));  
  
System.out.println(linesPerFile);
```

O resultado será algo como:

```
{  
    ./br/com/casadocodigo/java8/Capitulo3.java=45,  
    ./br/com/casadocodigo/java8/Usuario.java=44,  
    ./br/com/casadocodigo/java8/Capitulo8.java=156,  
    ...  
}
```

Essa abordagem está correta e já é muito mais concisa e expressiva do que se tivéssemos usado `BufferedReader`s e loops para criar esse `Map`. Ao mesmo tempo, essa solução não é muito funcional: o lambda passado para o `forEach` utiliza uma variável declarada fora do seu escopo, mudando seu estado, o que chamamos de efeito colateral. Isso diminui a possibilidade de otimizações, em especial para a execução em paralelo.

Podemos criar esse mesmo mapa com um outro coletor mais específico para esse tipo de tarefa, o `toMap`:

```
Map<Path, Long> lines =  
    Files.list(Paths.get("./br/com/casadocodigo/java8"))  
        .filter(p -> p.toString().endsWith(".java"))  
        .collect(Collectors.toMap(  
            p -> p,  
            p -> lines(p).count()));  
  
System.out.println(lines);
```

O `toMap` recebe duas `Functions`. A primeira produzirá a chave (no nosso caso o próprio `Path`) e a segunda produzirá o valor (a quantidade de linhas). Como é comum precisarmos de um lambda que retorna o próprio argumento (o nosso `p -> p`), podemos utilizar `Function.identity()` para deixar mais claro.

Se quisermos gerar um mapa de cada arquivo para toda a lista de linhas contidas nos arquivos, podemos utilizar um outro coletor e gerar um `Map<Path, List<String>>`:

```
Map<Path, List<String>> content =  
    Files.list(Paths.get("./br/com/casadocodigo/java8"))  
        .filter(p -> p.toString().endsWith(".java"))  
        .collect(Collectors.toMap(  
            Function.identity(),  
            p -> lines(p).collect(Collectors.toList())));
```

Certamente, o `toMap` vai aparecer bastante no seu código. São muitos os casos em que queremos gerar mapas temporários para processar dados e gerar estatísticas e relatórios.

Mapear todos os usuários utilizando seu nome como chave fica fácil:

```
Map<String, Usuario> nameToUser = usuarios  
    .stream()
```

```
.collect(Collectors.toMap(
    Usuario::getNome,
    Function.identity()));
```

Se o `Usuario` fosse uma entidade JPA, poderíamos utilizar `toMap(Usuario::getId, Function.identity)` para gerar um `Map<Long, Usuario>`, no qual a chave é o `id` da entidade.

9.2 GROUPINGBY E PARTITIONINGBY

Há muitos coletores já prontos que sabem gerar mapas importantes. Para os nossos exemplos ficarem mais interessantes, vamos popular nossa lista de usuários com mais objetos:

```
Usuario user1 = new Usuario("Paulo Silveira", 150, true);
Usuario user2 = new Usuario("Rodrigo Turini", 120, true);
Usuario user3 = new Usuario("Guilherme Silveira", 90);
Usuario user4 = new Usuario("Sergio Lopes", 120);
Usuario user5 = new Usuario("Adriano Almeida", 100);

List<Usuario> usuarios =
    Arrays.asList(user1, user2, user3, user4, user5);
```

Considere que o `boolean` passado para a sobrecarga do construtor de `Usuario` é para definir se ele é um moderador ou não.

Queremos um mapa em que a chave seja a pontuação do usuário e o valor seja uma lista de usuários que possuem aquela pontuação. Isto é, um `Map<Integer, List<Usuario>>`.

Para fazer isso de maneira tradicional, precisamos passar por todos os usuários e ver se já existe uma lista para aquela pontuação. Caso não exista, criamos uma `ArrayList`. Se existe, adicionamos o usuário a lista. O código fica da seguinte forma:

```
Map<Integer, List<Usuario>> pontuacao = new HashMap<>();

for(Usuario u: usuarios) {
    if(!pontuacao.containsKey(u.getPontos())) {
        pontuacao.put(u.getPontos(), new ArrayList<>());
    }
    pontuacao.get(u.getPontos()).add(u);
```

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

A saída é a seguinte, aqui formatada para facilitar a leitura:

```
{  
    100=[Usuario Adriano Almeida],  
    150=[Usuario Paulo Silveira],  
    120=[Usuario Rodrigo Turini, Usuario Sergio Lopes],  
    90=[Usuario Guilherme Silveira]  
}
```

No Java 8 poderíamos diminuir um pouco esse código com a ajuda de novos métodos default do Map:

```
Map<Integer, List<Usuario>> pontuacao = new HashMap<>();  
  
for(Usuario u: usuarios) {  
    pontuacao  
        .computeIfAbsent(u.getPontos(), user -> new ArrayList<>())  
        .add(u);  
}  
  
System.out.println(pontuacao);
```

O método `computeIfAbsent` vai chamar a `Function` do lambda no caso de não encontrar um valor para a chave `u.getPontos()` e associar o resultado (a nova `ArrayList`) a essa mesma chave. Isto é, essa invocação do `computeIfAbsent` faz o papel do `if` que fizemos no código anterior.

Mas o que realmente queremos é trabalhar com `Streams`. Poderíamos escrever um `Collector` ou trabalhar manualmente com o `reduce`, mas há um `Collector` que faz exatamente esse trabalho:

```
Map<Integer, List<Usuario>> pontuacao = usuarios  
    .stream()  
    .collect(Collectors.groupingBy(Usuario::getPontos));
```

A saída é a mesma! O segredo é o `Collectors.groupingBy`, que é uma factory de `Collectors` que fazem agrupamentos.

Podemos fazer mais. Podemos particionar todos os usuários entre moderadores e não moderadores, usando o `partitionBy`:

```
Map<Boolean, List<Usuario>> moderadores = usuarios
    .stream()
    .collect(Collectors.partitioningBy(Usuario::isModerador));

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

O resultado é um mapa para listas com quem é moderador e quem não é:

```
{
false=
[Usuario Guilherme Silveira, Usuario Sergio Lopes,
 Usuario Adriano Almeida],
true=
[Usuario Paulo Silveira, Usuario Rodrigo Turini]
}
```

O `partitioningBy` nada mais é do que uma versão mais eficiente para ser usada ao agrupar booleans.

O `partitioningBy(Usuario::isModerador)` nos devolve um `Map<Boolean, List<Usuario>>`. E se quiséssemos uma lista com os nomes dos usuários? Se fizermos `stream().map(Usuario::getNome)` não poderemos particionar por `Usuario::isModerador`, pois o `map` nos retornaria um `Stream<String>`.

Tanto o `partitioningBy` quanto o `groupingBy` possuem uma sobrecarga que permite passar um `Collector` como argumento. Há um `Collector` que sabe coletar os objetos ao mesmo tempo que realiza uma transformação de `map`.

Em vez de guardar os objetos dos usuários, poderíamos guardar uma lista com apenas o nome de cada usuário, usando o `mapping` para coletar esses nomes em uma lista:

```
Map<Boolean, List<String>> nomesPorTipo = usuarios
    .stream()
    .collect(
        Collectors.partitioningBy(
            Usuario::isModerador,
            Collectors.mapping(Usuario::getNome,
                Collectors.toList())));

```

E o resultado:

```
{  
    false=  
        [Guilherme Silveira, Sergio Lopes, Adriano Almeida],  
    true=  
        [Paulo Silveira, Rodrigo Turini]  
}
```

Vamos a mais um desafio. Queremos particionar por moderação, mas ter como valor não os usuários, mas sim a soma de seus pontos. Também existe um coletor para realizar essas somatórias, que pode ser usado em conjunto com o `partitioningBy` e `groupingBy`:

```
Map<Boolean, Integer> pontuacaoPorTipo = usuarios  
    .stream()  
    .collect(  
        Collectors.partitioningBy(  
            Usuario::isModerador,  
            Collectors.summingInt(Usuario::getPontos)));  
  
System.out.println(pontuacaoPorTipo);
```

E o resultado indica quantos pontos totalizam os moderadores e os usuários normais:

```
{false=310, true=270}
```

Conhecer bem toda a factory `Collectors` certamente vai ajudar suas manipulações de coleções. Perceba que não usamos mais loops para processar os elementos. Até mesmo para concatenar todos os nomes dos usuários há um coletor:

```
String nomes = usuarios  
    .stream()  
    .map(Usuario::getNome)  
    .collect(Collectors.joining(", "));
```

Com os streams e coletores, conseguimos os mesmos resultados de antigamente, porém em um estilo funcional e consequentemente mais enxuto e expressivo. Além disso, mesmo que você não tenha percebido, acabamos trabalhando com **menos efeitos colaterais e favorecemos a imutabilidade** das coleções originais.

E qual é a vantagem de evitar efeitos colaterais e mutabilidade? Facilitar a paralelização.

9.3 EXECUTANDO O PIPELINE EM PARALELO

Vamos voltar a um exemplo simples de uso dos streams. Filtrar os usuários com mais de 100 pontos, ordená-los e coletar o resultado em uma lista:

```
List<Usuario> filtradosOrdenados = usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .sorted(Comparator.comparing(Usuario::getNome))
    .collect(Collectors.toList());
```

Tudo acontece na própria thread, como é esperado. Se tivermos uma lista com milhões de usuários, o processo poderá levar mais que alguns segundos.

E se precisarmos parallelizar esse processo? Até seu smartphone possui 4 processadores. Escrever um código que use `Thread` para filtrar, ordenar e coletar dá bastante trabalho. Uma opção seria tirar proveito da API de Fork/Join. Apesar de já ser um pouco mais simples, ainda assim não é uma tarefa fácil.

As collections oferecem uma implementação de `Stream` diferente, o stream paralelo. Ao usar um stream paralelo, ele vai decidir quantas threads deve utilizar, como deve quebrar o processamento dos dados e qual será a forma de unir o resultado final em um só. Tudo isso sem você ter de configurar nada. Basta apenas invocar `parallelStream` em vez de `Stream`:

```
List<Usuario> filtradosOrdenados = usuarios.parallelStream()
    .filter(u -> u.getPontos() > 100)
    .sorted(Comparator.comparing(Usuario::getNome))
    .collect(Collectors.toList());
```

Pronto! O resultado será exatamente o mesmo da versão sequencial.

Caso você não tenha acesso ao produtor de dados original, o `Stream` tem um método `parallel` que devolve sua versão de execução em paralelo. Há também o `sequential` que retorna sua versão clássica.

Performance final

Seu código vai rodar mais rápido? Não sabemos. Se a coleção for pequena, o overhead de utilizar essa abordagem certamente tornará a execução bem mais lenta. É necessário tomar cuidado com o uso dos streams paralelos. Eles são uma forma simples de realizar operações com a API de Fork/Join: o tamanho do input precisa ser grande.

Com uma coleção pequena, não podemos enxergar as perdas e ganhos com facilidade. Vamos gerar uma quantidade grande de números, filtrá-los e ordená-los, para poder ter uma base de comparação.

Para gerar os números de 1 a um bilhão, utilizaremos o `LongStream.range`. Usaremos o `parallel` e o `filter` para filtrar:

```
long sum =  
    LongStream.range(0, 1_000_000_000)  
        .parallel()  
        .filter(x -> x % 2 == 0)  
        .sum();  
System.out.println(sum);
```

Em um computador com 2 cores, executamos o código em `1.276s` de tempo realmente gasto. As configurações da máquina não importam muito, o que queremos é fazer a comparação.

Removendo a invocação do `parallel()`, o tempo é significativamente maior: `1.842s`. Não é o dobro, pois paralelizar a execução de um pipeline sempre tem seu preço.

Essa diferença favorece o `parallel` pois temos um input grande de dados. Se diminuirmos a sequência gerada de 1 bilhão para 1 milhão, fica claro o problema. O paralelo roda em `0.239s` e o sequencial em `0.201`. Isso mesmo: a versão paralela é mais lenta por causa do overhead: quebrar o problema em várias partes, juntar os dados resultantes etc.

São dois fatores que podem ajudar sua decisão: o tamanho do input dos dados e a velocidade de execução da operação. Soma de números é uma operação muito barata, por isso é necessário um input grande. Se a operação fosse mais lenta, envolvesse operações blocantes, o tamanho necessário para valer a pena seria menor.

Micro benchmarks são sempre perigosos e podem enganar com facilidade, ainda mais sem saber quais serão as operações realizadas. Seja cético e teste seus próprios casos.

9.4 OPERAÇÕES NÃO DETERMINÍSTICAS E ORDERED STREAMS

Há diversas outras considerações que devem ser feitas. Algumas operações no `pipeline` são chamadas de não-determinísticas. Elas podem devolver diferentes resulta-

dos quando utilizadas em streams paralelos. Os principais exemplos são o `forEach` e o `findAny`.

Ao invocar esses dois métodos em um stream paralelo, você não tem garantia da ordem de execução. Isso melhora sua performance em paralelo. Caso necessite garantir a ordem da execução, você deve utilizar o `forEachOrdered` e o `findFirst` respectivamente. Na maioria das vezes eles não são necessários.

Mesmo métodos como `map` podem ter sua execução em paralelo melhoradas. Quando criamos um `stream` de uma `List` (ou do `LongStream.range`), ele é criado de maneira `ordered`. Isto é, ao coletarmos a lista resultante de um `map` da função `x -> x + 1` nos elementos `1, 2, 3`, no final teremos `2, 3, 4`. Se gerarmos um `stream` com o mesmo conteúdo, só que vindo de um `HashSet`, o resultado final pode ser `3, 4, 2`. Ao executar em paralelo o `map`, a fase de fazer o `join` dos resultados será bem mais simples por não necessitar garantir a mesma ordem de entrada!

Você pode relaxar a restrição de ordem de um `Stream ordered` invocando seu método `unordered`.

Um outro obstáculo para a performance do `Stream` paralelo é o coletor de agrupamento. O `Collectors.groupingBy` garante a ordem de aparição dos elementos ao agrupá-los, o que pode ser custoso na fase de fazer `join`. Utilizar `Collectors.groupingByConcurrent` não garante essa ordem, utilizando um único mapa concorrente como `ConcurrentHashMap`, mas a performance final será melhor.

E se tivéssemos efeitos colaterais?

Os streams paralelos são incrivelmente poderosos, mas não há mágica: continuamos a ter problemas se houver operações com efeitos colaterais em estado compartilhado.

Imagine uma outra forma de somar de 1 a 1 bilhão. Em vez de usar o `sum` do `LongStream`, vamos criar atributo `total` para armazenar o resultado. Pelo `forEach` realizamos a soma nesse atributo em comum. Repare:

```
class UnsafeParallelStreamUsage {  
    private static long total = 0;  
  
    public static void main (String... args) {  
        LongStream.range(0, 1_000_000_000)  
            .parallel()  
    }  
}
```

```
    .filter(x -> x % 2 == 0)
    .forEach(n -> total += n);

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

Se você possuir mais de um core, cada vez que você rodar esse código obterá provavelmente um resultado diferente! O uso concorrente de uma variável compartilhada possibilita o interleaving de operações de forma indesejada. Claro, você pode utilizar o `synchronized` dentro do bloco do lambda para resolver isso, mas perdendo muita performance. Estado compartilhado entre threads continua sendo um problema.

Para saber mais: Spliterators

A base do trabalho de todo Stream paralelo é o `Spliterator`. Ele é como um `Iterator`, só que muitas vezes pode ser facilmente quebrado em spliterators menores, para que cada thread disponível consuma um pedaço do seu stream.

A interface `Iterable` agora também define um método default `spliterator()`. Tudo que vimos de paralelização são abstrações que utilizam `Spliterators` por debaixo dos panos, junto com a API de Fork/Join. Caso você vá criar uma operação complexa paralela, é esse o caminho que deve seguir.

CAPÍTULO 10

Chega de Calendar! Nova API de datas

Trabalhar com datas em Java sempre foi um tanto complicado. Existem muitas críticas às classes `Date` e `Calendar`: são mutáveis, possuem várias limitações, decisões de design estranhas, alguns bugs conhecidos e dificuldade de se trabalhar com elas.

Uma grande introdução do Java 8 foi o pacote `java.time`, que nos trouxe uma nova API de datas!

10.1 A JAVA.TIME VEM DO JODA TIME

Há muito tempo existe o desejo de uma API de datas melhor, baseada no projeto Joda-Time. Essa afirmação foi uma das motivações usadas na proposta dessa nova feature.

O Joda-Time é uma poderosa biblioteca open source bastante conhecida e utilizada no mercado, que trabalha com tempo, datas e cronologia:

<http://www.joda.org/joda-time/>

Baseada nesse projeto, a nova API de datas possui métodos mais intuitivos, seu código ficou muito mais interessante e fluente. Você pode conhecer mais da JSR 310, que criou essa API, aqui:

<https://jcp.org/en/jsr/detail?id=310>.

Ela foi especificada por Steven Colebourne e Michael Nascimento, dois nomes bem conhecidos na comunidade, sendo este último um brasileiro.

10.2 TRABALHANDO COM DATAS DE FORMA FLUENTE

Aplicar uma transformação em um `Calendar` é um processo muito verboso, como por exemplo para criar uma data com um mês a partir da data atual. Repare:

```
Calendar mesQueVem = Calendar.getInstance();
mesQueVem.add(Calendar.MONTH, 1);
```

Com a nova API de datas podemos fazer essa mesma operação de uma forma mais moderna, utilizando sua interface fluente:

```
LocalDate mesQueVem = LocalDate.now().plusMonths(1);
```

Como a maior parte de seus métodos não possibilita o retorno de parâmetros nulos, podemos encadear suas chamadas de forma fluente sem a preocupação de receber um `NullPointerException`!

Da mesma forma que fizemos para adicionar o mês, podemos utilizar os métodos `plusDays`, `plusYears` em diante de acordo com nossa necessidade. E de forma igualmente simples, conseguimos decrementar os valores utilizando os métodos `minus` presentes nesses novos modelos. Para subtrair um ano, por exemplo, faríamos:

```
LocalDate anoPassado = LocalDate.now().minusYears(1);
```

Um ponto importante é que a classe `LocalDate` representa uma data sem horário nem timezone, por exemplo `25-01-2014`. Se as informações de horário forem importantes, usamos a classe `LocalDateTime` de forma bem parecida:

```
LocalDateTime agora = LocalDateTime.now();
System.out.println(agora);
```

Um exemplo de saída desse código seria 2014-02-19 T17:49:19.587.

Há ainda o `LocalTime` para representar apenas as horas:

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

Neste caso, a saída seria algo como 17:49:19.587.

Uma outra forma de criar uma `LocalDateTime` com horário específico é utilizando o método `atTime` da classe `LocalDate`, repare:

```
LocalDateTime hojeAoMeioDia = LocalDate.now().atTime(12,0);
```

Assim como fizemos com esse método `atTime`, sempre podemos utilizar os método `at` para combinar os diferentes modelos. Observe como fica simples unir uma classe `LocalDate` com um `LocalTime`:

```
LocalTime agora = LocalTime.now();
LocalDate hoje = LocalDate.now();
LocalDateTime dataEhora = hoje.atTime(agora);
```

Também podemos, a partir desse `LocalDateTime`, chamar o método `atZone` para construir um `ZonedDateTime`, que é o modelo utilizado para representar uma data com hora e timezone.

```
ZonedDateTime dataComHoraETimezone =
    dataEhora.atZone(ZoneId.of("America/Sao_Paulo"));
```

Em alguns momentos é importante trabalhar com o timezone, mas o ideal é utilizá-lo apenas quando realmente for necessário. A própria documentação pede cuidado com o uso dessa informação, pois muitas vezes não será necessário e usá-la pode causar problemas, como para guardar o aniversário de um usuário.

Para converter esses objetos para outras medidas de tempo podemos utilizar os métodos `to`, como é o caso do `toLocalDateTime` presente na classe `ZonedDateTime`:

```
LocalDateTime semTimeZone = dataComHoraETimezone.toLocalDateTime();
```

O mesmo pode ser feito com o método `toLocalDate` da classe `LocalDateTime`, entre diversos outros métodos para conversão.

Além disso, as classes dessa nova API contam com o método estático `of`, que é um *factory method* para construção de suas novas instâncias:

```
LocalDate date = LocalDate.of(2014, 12, 25);
LocalDateTime dateTime = LocalDateTime.of(2014, 12, 25, 10, 30);
```

E claro, é muito comum precisarmos instanciar uma data a partir de uma `String`. Para isso, podemos utilizar o método `parse` que será melhor detalhado adiante.

Repare que todas essas invocações não só podem como devem ser encadeadas por um bom motivo: o modelo do `java.time` é imutável! Cada operação devolve um novo valor, nunca alterando o valor interno dos horários, datas e intervalos utilizados na operação. Isso simplifica muita coisa, não apenas para trabalhar concorrentemente.

De forma similar aos `setters`, os novos modelos imutáveis possuem os métodos `withs` para facilitar a inserção de suas informações. Para modificar o ano de um `LocalDate`, por exemplo, poderíamos utilizar o método `withYear`. Repare:

```
LocalDate dataDoPassado = LocalDate.now().withYear(1988);
```

Mas e para recuperar essas informações? Podemos utilizar seus métodos `gets`, de acordo com o valor que estamos procurando. Por exemplo, o `getYear` para saber o ano, ou `getMonth` para o mês, assim por diante.

```
LocalDate dataDoPassado = LocalDate.now().withYear(1988);
System.out.println(dataDoPassado.getYear());
```

Neste caso, a saída será `1988`, que foi o valor adicionado pelo método `withYear`.

Isso simplifica um pouco nosso código. Em um `Calendar`, por exemplo, teríamos que utilizar o método `get` passando a constante `Calendar.YEAR` como argumento.

Existem também outros comportamentos essenciais, como saber se alguma medida de tempo acontece antes, depois ou ao mesmo tempo que outra. Para esses casos, utilizamos os métodos `is`:

```
LocalDate hoje = LocalDate.now();
LocalDate amanha = LocalDate.now().plusDays(1);

System.out.println(hoje.isBefore(amanha));
System.out.println(hoje.isAfter(amanha));
System.out.println(hoje.isEqual(amanha));
```

Neste exemplo, apenas o `isBefore` vai retornar `true`.

Há ainda os casos em que queremos comparar datas iguais, porém em `timezones` diferentes. Utilizar o método `equals`, nesse caso, não causaria o efeito esperado — claro, afinal a sobrescrita desse método na classe `ZonedDateTime` espera que o `offset` entre as datas seja o mesmo:

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj instanceof ZonedDateTime) {  
        ZonedDateTime other = (ZonedDateTime) obj;  
        return dateTime.equals(other.dateTime) &&  
            offset.equals(other.offset) &&  
            zone.equals(other.zone);  
    }  
    return false;  
}
```

Para estes casos podemos e devemos utilizar o método `isEqual` exatamente como fizemos com o `LocalDate`. Repare:

```
ZonedDateTime tokyo = ZonedDateTime  
.of(2011, 5, 2, 10, 30, 0, 0, ZoneId.of("Asia/Tokyo"));  
  
ZonedDateTime saoPaulo = ZonedDateTime  
.of(2011, 5, 2, 10, 30, 0, 0, ZoneId.of("America/Sao_Paulo"));  
  
System.out.println(tokyo isEqual(saoPaulo));
```

Não muito diferente do que podemos esperar, o resultado ainda será `false`. Apesar da semelhança entre as datas, elas estão em `timezones` diferentes, portanto não são iguais. Para que o resultado do método `isEqual` de um `ZonedDateTime` seja `true`, precisamos acertar a diferença de tempo entre as duas datas. Uma forma de fazer isso seria adicionar as 12 horas de diferença na instância de `tokyo`, repare:

```
ZonedDateTime tokyo = ZonedDateTime  
.of(2011, 5, 2, 10, 30, 0, 0, ZoneId.of("Asia/Tokyo"));  
  
ZonedDateTime saoPaulo = ZonedDateTime
```

```
.of(2011, 5, 2, 10, 30, 0, 0, ZoneId.of("America/Sao_Paulo"));

tokyo = tokyo.plusHours(12);

System.out.println(tokyo.isEqual(saoPaulo));
```

Agora sim, ao executar o método a saída será `true`.

A nova API possui diversos outros modelos que facilitam bastante nosso trabalho com data e tempo, como por exemplo as classes `MonthDay`, `Year` e `YearMonth`.

Para obter o dia do mês atual, por exemplo, poderíamos utilizar o método `getDayOfMonth` de uma instância da classe `MonthDay`. Repare:

```
System.out.println("hoje é dia: " + MonthDay.now().getDayOfMonth());
```

Você pode pegar o `YearMonth` de uma determinada data.

```
YearMonth ym = YearMonth.from(data);
System.out.println(ym.getMonth() + " " + ym.getYear());
```

A vantagem de trabalhar apenas com ano e mês, por exemplo, é poder agrupar dados de uma forma mais direta. Com o `Calendar`, precisaríamos utilizar uma data completa e ignorar dia e hora, por exemplo. Soluções incompletas.

10.3 ENUMS NO LUGAR DE CONSTANTES

Essa nova API de datas favorece o uso de Enums no lugar das famosas constantes do `Calendar`. Para representar um mês, por exemplo, podemos utilizar o enum `Month`. Cada valor desse enum tem um valor inteiro que o representa seguindo o intervalo de 1 (Janeiro) a 12 (Dezembro). Você não precisa, mas trabalhar com esses enums é uma boa prática, afinal seu código fica muito mais legível:

```
System.out.println(LocalDate.of(2014, 12, 25));
System.out.println(LocalDate.of(2014, Month.DECEMBER, 25));
```

Nos dois casos, o valor de saída é `25/12/2014`.

Outra vantagem de utilizar os enums são seus diversos métodos auxiliares. Note como é simples consultar o primeiro dia do trimestre de determinado mês, ou então incrementar/decrementar meses:

```
System.out.println(Month.DECEMBER.firstMonthOfQuarter());  
System.out.println(Month.DECEMBER.plus(2));  
System.out.println(Month.DECEMBER.minus(1));
```

O resultado desse código será:

```
OCTOBER  
FEBRUARY  
NOVEMBER
```

Para imprimir o nome de um mês formatado, podemos utilizar o método `getDisplayName` fornecendo o estilo de formatação (completo, resumido, entre outros) e também o `Locale`:

```
Locale pt = new Locale("pt");  
  
System.out.println(Month.DECEMBER  
    .getDisplayName(TextStyle.FULL, pt));  
  
System.out.println(Month.DECEMBER  
    .getDisplayName(TextStyle.SHORT, pt));
```

Repare que, como estamos utilizando `TextStyle.FULL` no primeiro exemplo e `TextStyle.SHORT` no seguinte, teremos como resultado:

```
Dezembro  
Dez
```

Outro enum introduzido no `java.time` foi o `DayOfWeek`. Com ele, podemos representar facilmente um dia da semana, sem utilizar constantes ou números mágicos!

10.4 FORMATANDO COM A NOVA API DE DATAS

A formatação de datas também recebeu melhorias. Formatar um `LocalDateTime`, por exemplo, é um processo bem simples! Tudo que você precisa fazer é chamar o método `format` passando um `DateTimeFormatter` como parâmetro.

Para formatar em horas, por exemplo, podemos fazer algo como:

```
LocalDateTime agora = LocalDateTime.now();  
String resultado = agora.format(DateTimeFormatter.ISO_LOCAL_TIME);
```

Um exemplo de resultado seria 01:15:45, ou seja, o pattern é hh:mm:ss.

Note que usamos um `DateTimeFormatter` predefinido, o `ISO_LOCAL_TIME`. Assim como ele existem diversos outros que você pode ver no javadoc do `DateTimeFormatter`.

Mas como criar um `DateTimeFormatter` com um novo padrão? Uma das formas é usando o método `ofPattern`, que recebe uma `String` como parâmetro:

```
LocalDateTime agora = LocalDateTime.now();
agora.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"));
```

O resultado do método `format` seria, por exemplo, 06/02/2014. Também existe uma sobrecarga desse método que, além do *pattern*, recebe um `java.util.Locale`.

De forma parecida, podemos transformar uma `String` em alguma representação de data ou tempo válida, e para isso utilizamos o método `parse!` Podemos, por exemplo, retornar o nosso resultado anterior em um `LocalDate` utilizando o mesmo formatador:

```
LocalDateTime agora = LocalDateTime.now();
DateTimeFormatter formatador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String resultado = agora.format(formatador);
LocalDate agoraEmData = LocalDate.parse(resultado, formatador);
```

Repare que fizemos o `parse` desse resultado em um `LocalDate`, e não em um `LocalDateTime`, que é o tipo da data inicial. Não poderíamos retornar um `LocalDateTime`, afinal quando formatamos em data (com o padrão `dd/MM/yyyy`), perdemos as informações de tempo! Essa tentativa resultaria em uma `DateTimeParseException`!

10.5 DATAS INVÁLIDAS

Ao trabalhar com `Calendar`, alguma vez você já pode ter se surpreendido ao executar um código como este:

```
Calendar instante = Calendar.getInstance();
instante.set(2014, Calendar.FEBRUARY, 30);
SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yy");
System.out.println(dateFormat.format(instante.getTime()));
```

Levando em consideração que fevereiro deste ano só vai até o dia 28, qual será o resultado? Uma exception? Esse pode ser o resultado esperado, afinal estamos passando um argumento inválido!

Mas, na verdade, o resultado será: 02/03/14. Isso mesmo, o `Calendar` ajustou o mês e dia, sem dar nenhum feedback desse erro que provavelmente passaria despercebido.

Nem sempre isso é o que esperamos ao tentar criar uma data com valor inválido.

Muito diferente desse comportamento, a nova API de datas vai lançar uma `DateTimeException` em casos como esse. Repare:

```
LocalDate.of(2014, Month.FEBRUARY, 30);
```

O resultado será: `java.time.DateTimeException: Invalid date 'FEBRUARY 30'.`

O mesmo acontecerá se eu tentar criar um `LocalDateTime` com um horário inválido:

```
LocalDateTime horaInvalida = LocalDate.now().atTime(25, 0);
```

Neste caso, a exception será: `java.time.DateTimeException: Invalid value for HourOfDay (valid values 0 - 23): 25.`

A própria API do Joda-time deixou passar um problema parecido ao criar a classe `DateMidnight`. Na documentação do método `toDateMidnight` da classe `LocalDate`, você vai encontrar a seguinte nota:

“Desde a versão 1.5, recomendamos que você evite o `DateMidnight` e passe a utilizar o método `toDateTimeAtStartOfDay` por causa da exceção que será detalhada a seguir. Esse método vai lançar uma exceção se o horário de verão padrão começar à meia noite e esse `LocalDate` represente esse horário da troca. O problema é que não existe meia noite nessa data e, assim, uma exceção é lançada”.

Ou seja, esse método é um tanto perigoso, pois sempre lança uma exceção no dia em que começa o horário de verão. O próprio Stephen Colebourne disse em seu blog que não existe uma boa razão para algum dia utilizar a classe `DateMidnight`. E claro, essa classe não está presente na nova API de datas.

10.6 DURAÇÃO E PERÍODO

Sempre foi problemático trabalhar com a diferença de alguma medida de tempo em Java. Por exemplo, como calcular a diferença de dias entre dois `Calendars`? Você possivelmente já passou por isso:

```
Calendar agora = Calendar.getInstance();

Calendar outraData = Calendar.getInstance();
outraData.set(1988, Calendar.JANUARY, 25);

long diferenca = agora.getTimeInMillis() - outraData.getTimeInMillis();

long milissegundosDeUmDia = 1000 * 60 * 60 * 24;

long dias = diferenca / milissegundosDeUmDia;
```

Resolvemos o problema, claro, mas sabemos que trabalhar com a diferença de milissegundos pode não ser o ideal. Há também alguns casos particulares que não são cobertos nesse caso. E a cada nova necessidade, precisaríamos criar um código igualmente verboso e difícil de manter.

Agora podemos fazer essa mesma operação de forma muito mais simples, utilizando o enum `ChronoUnit` da nova api:

```
LocalDate agora = LocalDate.now();
LocalDate outraData = LocalDate.of(1989, Month.JANUARY, 25);
long dias = ChronoUnit.DAYS.between(outraData, agora);
```

Esse enum está presente no pacote `java.time.temporal` e possui uma representação para cada uma das diferentes medidas de tempo e data. Além disso, possui vários métodos para facilitar o cálculo de diferença entre seus valores e que nos auxiliam a extrair informações úteis, como é o caso do `between`.

Mas e se também quisermos saber a diferença de anos e meses entre essas duas datas? Poderíamos utilizar o `ChronoUnit.YEARS` e `ChronoUnit.MONTHS` para obter essas informações, mas ele vai calcular cada uma das medidas de forma separada. Repare:

```
long dias = ChronoUnit.DAYS.between(outraData, agora);
long meses = ChronoUnit.MONTHS.between(outraData, agora);
long anos = ChronoUnit.YEARS.between(outraData, agora);
System.out.printf("%s dias, %s meses e %s anos", dias, meses, anos);
```

Neste caso, o resultado será algo como: 9147 dias, 300 meses e 25 anos.

Uma forma de conseguir o resultado que esperamos: os dias, meses e anos entre duas datas, é utilizando o modelo `Period`. Essa classe da API também possui o método `between`, que recebe duas instâncias de `LocalDate`:

```
LocalDate agora = LocalDate.now();
LocalDate outraData = LocalDate.of(1989, Month.JANUARY, 25);
Period periodo = Period.between(outraData, agora);
System.out.printf("%s dias, %s meses e %s anos",
    periodo.getDays(), periodo.getMonths(), periodo.getYears());
```

Note que agora o resultado será: 16 dias, 0 meses e 25 anos.

A classe `Period` tem uma série de métodos que auxiliam nas diversas situações que enfrentamos ao trabalhar com datas. Por exemplo, ao calcular uma diferença entre datas, é comum a necessidade de lidarmos com valores negativos. Observe o que acontece se alterarmos o ano da `outraData` para 2015:

```
LocalDate agora = LocalDate.now();
LocalDate outraData = LocalDate.of(2015, Month.JANUARY, 25);
Period periodo = Period.between(outraData, agora);
System.out.printf("%s dias, %s meses e %s anos",
    periodo.getDays(), periodo.getMonths(), periodo.getYears());
```

A saída será algo como: -15 dias, -11 meses e 0 anos.

Essa pode ser a saída esperada, mas caso não seja, podemos facilmente perguntar ao `Period` se ele é um período de valores negativos invocando o método `isNegative`. Caso seja, poderíamos negar seus valores com o método `negated`, repare:

```
Period periodo = Period.between(outraData, agora);

if (periodo.isNegative()) {
    periodo = periodo.negated();
}

System.out.printf("%s dias, %s meses e %s anos",
    periodo.getDays(), periodo.getMonths(), periodo.getYears());
```

Agora a saída terá seus valores invertidos: 15 dias, 11 meses e 0 anos.

Existem diversas outras formas de se criar um `Period`, além do método `between`. Uma delas é utilizando o método `of(years, months, days)` de forma fluente:

```
Period periodo = Period.of(2, 10, 5);
```

Também podemos criar um período com apenas dias, meses ou anos utilizando os métodos: `ofDays`, `ofMonths` ou `ofYears`.

Mas como criar um período de horas, minutos ou segundos? A resposta é: não criamos. Neste caso, estamos interessados em outra medida de tempo, que é a `Duration`. Enquanto um `Period` considera as medidas de data (dias, meses e anos), a `Duration` considera as medidas de tempo (horas, minutos, segundos etc.). Sua API é muito parecida, observe:

```
LocalDateTime agora = LocalDateTime.now();
LocalDateTime daquiAUMaHora = LocalDateTime.now().plusHours(1);
Duration duration = Duration.between(agora, daquiAUMaHora);

if (duration.isNegative()) {
    duration = duration.negated();
}

System.out.printf("%s horas, %s minutos e %s segundos",
    duration.toHours(), duration.toMinutes(), duration.getSeconds());
```

Repare que, como agora estamos trabalhando com tempo, utilizamos a classe `LocalDateTime`.

10.7 DIFERENÇAS PARA O JODA TIME

É importante lembrar que a nova API de datas (JSR-310) é baseada no Joda-Time, mas que não é uma cópia. Existem algumas diferenças de design que foram cuidadosamente apontadas pelo Stephen Colebourne em seu blog, no artigo **Why JSR-310 isn't Joda-Time**:

http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html.

As principais mudanças foram inspiradas pelas falhas de design do Joda-Time, como é o caso das referências nulas. No Joda-Time não só era possível fornecer valores nulos para grande maioria de seus métodos, como isso tinha um significado pra cada modelo. Por exemplo, para as representações de data e tempo o valor `null` significava `1970-01-01T00:00Z`. Para as classes `Duration` e `Period`, `null` significava zero.

Essa é uma estratégia de design um tanto inesperada, e que pode facilmente causar bugs em nossas aplicações.

Há ainda o caso da classe `DateTime` que implementa `ReadableInstant`, sendo que como uma interpretação humana de tempo não deveria. No fim, ela acaba

sendo uma projeção de uma linha do tempo de máquina em uma linha do tempo humana, ficando limitada à precisão de milissegundos.

CAPÍTULO 11

Um modelo de pagamentos com Java 8

Vamos a mais um exemplo sucinto para praticar os conceitos e principais pontos da API vistos até aqui. Relembrando que você pode acessar todo o código-fonte do livro aqui:

<https://github.com/peas/java8>

Para fixar a nova API e sintaxe, por mais que você já domine o Java, é importante praticar. Se possível, recrie o código você mesmo, sem copiar e colar.

11.1 UMA LOJA DE DIGITAL GOODIES

Vender bens digitais é o dia a dia da nossa loja.

Vendemos download de músicas, vídeos e imagens para serem utilizados em campanhas publicitárias. Cada um desses produtos possui um nome, um preço e o caminho do arquivo:

```
class Product {  
    private String name;  
    private Path file;  
    private BigDecimal price;  
  
    public Product(String name, Path file, BigDecimal price) {  
        this.name = name;  
        this.file = file;  
        this.price = price;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public Path getFile() {  
        return this.file;  
    }  
  
    public BigDecimal getPrice() {  
        return this.price;  
    }  
  
    public String toString() {  
        return this.name;  
    }  
}
```

Nosso e-commerce conta com diversos clientes, dos quais para nós importa apenas o nome:

```
class Customer {  
    private String name;  
  
    public Customer(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

```
    public String toString() {
        return this.name;
    }
}
```

E quando o usuário realiza uma compra no sistema, um `Payment` é utilizado para representá-la, armazenando uma lista de produtos comprados, o momento da compra e quem a realizou:

```
class Payment {
    private List<Product> products;
    private LocalDateTime date;
    private Customer customer;

    public Payment(List<Product> products,
                  LocalDateTime date,
                  Customer customer) {
        this.products =
            Collections.unmodifiableList(products);
        this.date = date;
        this.customer = customer;
    }

    public List<Product> getProducts() {
        return this.products;
    }

    public LocalDateTime getDate() {
        return this.date;
    }

    public Customer getCustomer() {
        return this.customer;
    }

    public String toString() {
        return "[Payment: " +
            date.format(DateTimeFormatter.ofPattern("dd/MM/yyyy")) +
            " " + customer + " " + products + "]";
    }
}
```

Utilizamos algumas das novidades do Java 8, como o `java.time` para armazenar a data e formatá-la.

Sim, poderíamos ter modelado as classes de mil outras maneiras. Em vez de `Path` poderíamos ter `URL` se considerarmos que os arquivos estão no cloud. Poderíamos ter deixado o relacionamento bidirecional, com o `Customer` possuindo também um `List<Payment>`. A escolha aqui é o suficiente para exercitarmos o nosso aprendizado e uma modelagem melhor fica sem dúvida a cargo do uso do TDD e de conhecer bem o domínio.

Vamos popular um pouco a nossa base de testes, criando quatro clientes:

```
Customer paulo = new Customer("Paulo Silveira");
Customer rodrigo = new Customer("Rodrigo Turini");
Customer guilherme = new Customer("Guilherme Silveira");
Customer adriano = new Customer("Adriano Almeida");
```

E fazemos o mesmo com os nossos produtos, criando seis deles:

```
Product bach = new Product("Bach Completo",
    Paths.get("/music/bach.mp3"), new BigDecimal(100));
Product poderosas = new Product("Poderosas Anita",
    Paths.get("/music/poderosas.mp3"), new BigDecimal(90));
Product bandeira = new Product("Bandeira Brasil",
    Paths.get("/images/brasil.jpg"), new BigDecimal(50));
Product beauty = new Product("Beleza Americana",
    Paths.get("beauty.mov"), new BigDecimal(150));
Product vingadores = new Product("Os Vingadores",
    Paths.get("/movies/vingadores.mov"), new BigDecimal(200));
Product amelie = new Product("Amelie Poulain",
    Paths.get("/movies/amelie.mov"), new BigDecimal(100));
```

Agora precisamos de alguns pagamentos, relacionando clientes com produtos. Repare que Paulo fez pagamentos duas vezes. Por uma questão de legibilidade, utilizamos o import estático de `Arrays.asList`:

```
LocalDateTime today = LocalDateTime.now();
LocalDateTime yesterday = today.minusDays(1);
LocalDateTime lastMonth = today.minusMonths(1);

Payment payment1 =
    new Payment(asList(bach, poderosas), today, paulo);
Payment payment2 =
```

```
new Payment(asList(bach, bandeira, amelie), yesterday, rodrigo);
Payment payment3 =
    new Payment(asList(beauty, vingadores, bach), today, adriano);
Payment payment4 =
    new Payment(asList(bach, poderosas, amelie), lastMonth, guilherme);
Payment payment5 =
    new Payment(asList(beauty, amelie), yesterday, paulo);

List<Payment> payments = asList(payment1, payment2,
    payment3, payment4, payment5);
```

11.2 ORDENANDO NOSSOS PAGAMENTOS

O primeiro desafio é fácil. Ordenar os pagamentos por data e imprimi-los, para que fique clara a nossa base de dados. Para isso, podemos encadear o `sorted` e o `forEach` do `stream` dessa coleção:

```
payments.stream()
    .sorted(Comparator.comparing(Payment::getDate))
    .forEach(System.out::println);
```

O resultado nos ajuda a enxergar as respostas dos próximos desafios:

```
[Payment: 16/02/2014 Guilherme Silveira
 [Bach Completo, Poderosas Anita, Amelie Poulain]]
[Payment: 15/03/2014 Rodrigo Turini
 [Bach Completo, Bandeira Brasil, Amelie Poulain]]
[Payment: 15/03/2014 Paulo Silveira
 [Beleza Americana, Amelie Poulain]]
[Payment: 16/03/2014 Paulo Silveira
 [Bach Completo, Poderosas Anita]]
[Payment: 16/03/2014 Adriano Almeida
 [Beleza Americana, Os Vingadores, Bach Completo]]
```

11.3 REDUZINDO BIGDECIMAL EM SOMAS

Vamos calcular o valor total do pagamento `payment1` utilizando a API de Stream e lambdas. Há um problema. Se preço fosse um `int`, poderíamos usar o `mapToDouble` e invocar o `sum` do `DoubleStream` resultante. Não é o caso. Teremos um `Stream<BigDecimal>` e ele não possui um `sum`.

Nesse caso precisaremos fazer a redução na mão, realizando a soma de `BigDecimal`. Podemos usar o `(total, price) -> total.add(price)`, mas fica ainda mais fácil usando um method reference:

```
payment1.getProducts().stream()
    .map(Product::getPrice)
    .reduce(BigDecimal::add)
    .ifPresent(System.out::println);
```

A resposta é 190.

O `ifPresent` é do `Optional<BigDecimal>` retornado pelo `reduce`. Se invocarmos o `reduce` que recebe o argumento de inicialização, teríamos como retorno um `BigDecimal` diretamente:

```
BigDecimal total =
    payment1.getProducts().stream()
        .map(Product::getPrice)
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Uma simples operação como a soma de `BigDecimal` apresenta várias soluções. Na maioria das vezes, essas soluções são equivalentes, pois a estratégia de execução do *pipeline* consegue ser bastante eficiente para evitar qualquer operação repetida.

Certamente poderíamos ter um método `getTotalAmount` em `Payment`, caso julgássemos necessário.

E se precisarmos somar todos os valores de todos os pagamentos da lista `payments`? Novamente nos deparamos com diversas opções. Podemos usar o mesmo código anterior, usando o `map` de `payments`:

```
Stream<BigDecimal> pricesStream =
    payments.stream()
        .map(p -> p.getProducts().stream()
            .map(Product::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add));
```

Repare que o código dentro do primeiro `map` é o mesmo que o código que usamos para calcular a soma dos valores do `payment1`. Com esse `map`, temos como resultado um `Stream<BigDecimal>`. Precisamos repetir a operação de `reduce` para somar esses valores intermediários. Isto é, realizamos a soma de preços dos produtos de cada pagamento, agora vamos somar cada um desses subtotais:

```
BigDecimal total =  
    payments.stream()  
        .map(p -> p.getProducts().stream()  
            .map(Product::getPrice)  
            .reduce(BigDecimal.ZERO, BigDecimal::add))  
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

O código está um pouco repetitivo. Em vez de realizarmos operações de soma em momentos distintos, podemos criar um único `Stream<BigDecimal>` com os valores de todos os produtos de todos pagamentos. Conseguimos esse `Stream` usando o `flatMap`:

```
Stream<BigDecimal> priceOfEachProduct =  
    payments.stream()  
        .flatMap(p -> p.getProducts().stream().map(Product::getPrice));
```

Se está difícil ler este código, leia-o passo a passo. O importante é enxergar essa função:

```
Function<Payment, Stream<BigDecimal>> mapper =  
    p -> p.getProducts().stream().map(Product::getPrice);
```

Essa função mapeia um `Payment` para o `Stream` que passeia por todos os seus produtos. E é por esse exato motivo que precisamos invocar depois o `flatMap` e não o `map`, caso contrário obteríamos um `Stream<Stream<BigDecimal>>`.

Para somar todo esse `Stream<BigDecimal>`, basta realizarmos a operação de `reduce` que conhecemos:

```
BigDecimal totalFlat =  
    payments.stream()  
        .flatMap(p -> p.getProducts().stream().map(Product::getPrice))  
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

11.4 PRODUTOS MAIS VENDIDOS

Queremos saber nossos campeões de vendas. Há, mais uma vez, diversas maneiras de realizar tal tarefa. Mapearemos nossos produtos para a quantidade que eles aparecem. Para tal, criamos um `Stream` com todos os `Product` vendidos. Mais uma vez entra o `flatMap`:

```
Stream<Product> products = payments.stream()
    .map(Payment::getProducts)
    .flatMap(p -> p.stream());
```

Em vez de `p -> p.stream()`, há a possibilidade de passar o lambda como method reference: `List::stream`:

```
Stream<Product> products = payments.stream()
    .map(Payment::getProducts)
    .flatMap(List::stream);
```

Sempre podemos juntar dois `maps` (independente de um deles ser `flat`) em um único `map`:

```
Stream<Product> products = payments.stream()
    .flatMap(p -> p.getProducts().stream());
```

E também não há diferença de performance significativa. Fica a seu cargo utilizar o que considerar mais legível. Como a API e o Java 8 são muito recentes, boas práticas ainda não surgiram para dizer qual das duas abordagens é mais adequada para facilitar a manutenção. Pessoalmente acreditamos que as duas são suficientemente claras.

Precisamos gerar um `Map` de `Product` para `Long`. Esse `Long` indica quantas vezes o produto foi vendido. Usaremos o `groupingBy`, agrupando todos esses produtos pelo próprio produto, mapeando-o pela sua contagem:

```
Map<Product, Long> topProducts = payments.stream()
    .flatMap(p -> p.getProducts().stream())
    .collect(Collectors.groupingBy(Function.identity(),
        Collectors.counting()));

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

Pode ser ruim ler o resultado do `System.out.println`, pois o `toString` do nosso `Map` gerará uma linha bem comprida. Podemos pegar o `entrySet` desse mapa e imprimir linha a linha:

```
topProducts.entrySet().stream()
    .forEach(System.out::println);
```

Certamente poderíamos ter encadeado essa chamada ao `entrySet` logo após nosso `collect`, porém não teríamos mais acesso ao `Map`.

Nosso resultado é parecido com o seguinte:

```
Beleza Americana=2
Amelie Poulain=3
Bandeira Brasil=1
Bach Completo=4
Poderosas Anita=2
Os Vingadores=1
```

Pelo visto, *Bach* ainda é o mais popular. Mas como pegar apenas essa entrada do mapa? Basta pedirmos a maior entrada do mapa considerando um `Comparator` que compare o `value` de cada entrada. Vale lembrar que ela é representada pela interface interna `Map.Entry`:

```
topProducts.entrySet().stream()
    .max(Comparator.comparing(Map.Entry::getValue))
    .ifPresent(System.out::println);
```

O `max` devolve um `Optional`, por isso o `ifPresent`.

11.5 VALORES GERADOS POR PRODUTO

Calculamos a quantidade de vendas por produtos. E a soma do valor por produto?

O processo é muito parecido. Em vez de agruparmos com o valor de `Collectors.counting`, queremos fazer algo como `Collectors.summing`. Há diversos métodos como esse em `Collectors`, porém todos trabalham com tipos primitivos. Para realizar a soma em `BigDecimal` teremos de deixar o `reduce` explícito:

```
Map<Product, BigDecimal> totalValuePerProduct = payments.stream()
    .flatMap(p -> p.getProducts().stream())
    .collect(Collectors.groupingBy(Function.identity(),
        Collectors.reducing(BigDecimal.ZERO, Product::getPrice,
            BigDecimal::add)));
```

Podemos usar a mesma estratégia do `stream().forEach(System.out::println)` para mostrar o resultado, mas vamos aproveitar e ordenar a saída por valor:

```
totalValuePerProduct.entrySet().stream()
    .sorted(Comparator.comparing(Map.Entry::getValue))
    .forEach(System.out::println);
```

Assim obtemos o resultado desejado:

```
Bandeira Brasil=50
Poderosas Anita=180
Os Vingadores=200
Beleza Americana=300
Amelie Poulain=300
Bach Completo=400
```

11.6 QUAIS SÃO OS PRODUTOS DE CADA CLIENTE?

Em um primeiro momento, podemos ter, para cada `Customer`, sua `List<Payment>`, bastando agrupar os `payments` com `groupingBy(Payment::getCustomer)`:

```
Map<Customer, List<Payment>> customerToPayments =
    payments.stream()
        .collect(Collectors.groupingBy(Payment::getCustomer));
```

Não estamos interessados nos `payments` de um `Customer`, e sim nas listas de `Product` dentro de cada um desses `Payments`.

Uma implementação inocente vai gerar uma `List<List<Product>>` dentro do valor do `Map`:

```
Map<Customer, List<List<Product>>> customerToProductsList =
    payments.stream()
        .collect(Collectors.groupingBy(Payment::getCustomer,
            Collectors.mapping(Payment::getProducts, Collectors.toList())));
    customerToProductsList.entrySet().stream()
        .sorted(Comparator.comparing(e -> e.getKey().getName()))
        .forEach(System.out::println);
```

A saída mostra o problema das listas aninhadas:

```
Adriano Almeida=[[Beleza Americana, Os Vingadores, Bach Completo]]
Guilherme Silveira=[[Bach Completo, Poderosas Anita, Amelie Poulain]]
Paulo Silveira=[[Bach Completo, Poderosas Anita], [Beleza Americana,
Amelie Poulain]]
Rodrigo Turini=[[Bach Completo, Bandeira Brasil, Amelie Poulain]]
```

Queremos esse mesmo resultado, porém com as listas achatadas em uma só.

Há duas formas. Sim, uma envolve o `flatMap` do mapa resultante. Dado o `customerToProductsList`, queremos que o `value` de cada `entry` seja achado. Fazemos:

```
Map<Customer, List<Product>> customerToProducts2steps =  
    customerToProductsList.entrySet().stream()  
    .collect(Collectors.toMap(Map.Entry::getKey,  
        e -> e.getValue().stream()  
            .flatMap(List::stream)  
            .collect(Collectors.toList())));  
  
customerToProducts2steps.entrySet().stream()  
    .sorted(Comparator.comparing(e -> e.getKey().getName()))  
    .forEach(System.out::println);
```

Usamos o `Collectors.toMap` para criar um novo mapa no qual a chave continua a mesma (`Map.Entry.getKey`) mas o valor é o resultado do `flatMap` dos `ListStream` de todas as listas.

Podemos verificar o conteúdo desse mapa, obtendo o mesmo resultado anterior, porém sem as listas aninhadas:

```
Adriano Almeida=[Beleza Americana, Os Vingadores, Bach Completo]  
Guilherme Silveira=[Bach Completo, Poderosas Anita, Amelie Poulain]  
Paulo Silveira=[Bach Completo, Poderosas Anita, Beleza Americana,  
Amelie Poulain]  
Rodrigo Turini=[Bach Completo, Bandeira Brasil, Amelie Poulain]
```

Poderíamos ter feito tudo com uma única chamada. Creio que nesse caso estouramos o limite da legibilidade do uso dessa API. Apenas para efeitos didáticos, veja como ficaria:

```
Map<Customer, List<Product>> customerToProducts1step = payments.stream()  
    .collect(Collectors.groupingBy(Payment::getCustomer,  
        Collectors.mapping(Payment::getProducts, Collectors.toList())))  
    .entrySet().stream()  
    .collect(Collectors.toMap(Map.Entry::getKey,  
        e -> e.getValue().stream()  
            .flatMap(List::stream)  
            .collect(Collectors.toList())));
```

Difícil de seguir a sequência. Certamente quebrar em vários passos é o mais indicado.

Como sempre, há outras formas de resolver o mesmo problema. Podemos usar o `reducing` mais uma vez, pois queremos acumular as listas de cada cliente agrupado.

```
Map<Customer, List<Product>> customerToProducts = payments.stream()
    .collect(Collectors.groupingBy(Payment::getCustomer,
        Collectors.reducing(Collections.emptyList(),
            Payment::getProducts,
            (l1, l2) -> { List<Product> l = new ArrayList<>();
                l.addAll(l1);
                l.addAll(l2);
                return l;} ));
```

Tivemos de escrever algo muito parecido com o que o `Collectors.toList` devolve. Infelizmente não há um método auxiliar que une duas `Collections`, impedindo a simplificação do terceiro argumento, que é um `BinaryOperator`. Criamos um coletor que pega todos os `Payment::getProducts` e vai acumulando todo o resultado em uma nova `ArrayList`.

O resultado é exatamente o mesmo que com o `flatMap`.

11.7 QUAL É NOSSO CLIENTE MAIS ESPECIAL?

Qual seria a estratégia para obter o desejado `Map<Customer, BigDecimal>`? Será a mesma que a da redução anterior, apenas mudando a operação. Começaremos com `BigDecimal.ZERO` e, para cada `Payment`, faremos `BigDecimal::add` da soma dos preços de seus produtos. Por esse motivo uma redução ainda aparece dentro do `reducing`!

```
Map<Customer, BigDecimal> totalValuePerCustomer = payments.stream()
    .collect(Collectors.groupingBy(Payment::getCustomer,
        Collectors.reducing(BigDecimal.ZERO,
            p -> p.getProducts().stream().map(Product::getPrice).reduce(
                BigDecimal.ZERO, BigDecimal::add),
            BigDecimal::add)));
```

O código está no mínimo muito acumulado. Cremos já termos passado do limite da legibilidade. Vamos quebrar essa redução, criando uma variável temporária responsável por mapear um `Payment` para a soma de todos os preços de seus produtos:

```
Function<Payment, BigDecimal> paymentToTotal =
    p -> p.getProducts().stream()
```

```
.map(Product::getPrice)
.reduce(BigDecimal.ZERO, BigDecimal::add);
```

Com isso, podemos utilizar essa Function no reducing:

```
Map<Customer, BigDecimal> totalValuePerCustomer = payments.stream()
.collect(Collectors.groupingBy(Payment::getCustomer,
    Collectors.reducing(BigDecimal.ZERO,
        paymentToTotal,
        BigDecimal::add)));
```

Novamente surge um forte indício de que deveria haver um método `getTotalAmount` em `Payment`, que calculasse todo esse valor. Nesse caso, poderíamos fazer um simples `reducing(BigDecimal.ZERO, Payment getTotalAmount, BigDecimal add)`. Ao mesmo tempo, este está sendo um excelente exercício de manipulação de coleções e relacionamentos.

Já podemos exibir o conteúdo desse mapa:

```
totalValuePerCustomer.entrySet().stream()
.sorted(Comparator.comparing(Map.Entry::getValue))
.forEach(System.out::println);
```

Assim obtemos o resultado esperado.

```
Rodrigo Turini=250
Guilherme Silveira=290
Paulo Silveira=440
Adriano Almeida=450
```

11.8 RELATÓRIOS COM DATAS

É muito simples separarmos os pagamentos por data, usando um `groupingBy(Payment::getDate)`. Há um perigo: o `LocalDateTime` vai agrupar os pagamentos até pelos milissegundos. Não é o que queremos.

Podemos agrupar por `LocalDate`, usando um `groupingBy(p -> p.getDate().toLocalDate())`, ou em um intervalo ainda maior, como por ano e mês. Para isso usamos o `YearMonth`:

```
Map<YearMonth, List<Payment>> paymentsPerMonth = payments.stream()
.collect(Collectors.groupingBy(p -> YearMonth.from(p.getDate())));
```

```
paymentsPerMonth.entrySet().stream()
    .forEach(System.out::println);
```

E se quisermos saber, também por mês, quanto foi faturado na loja? Basta agrupar com o mesmo critério e usar a redução que conhecemos: somando todos os preços de todos os produtos de todos pagamentos.

```
Map<YearMonth, BigDecimal> paymentsValuePerMonth = payments.stream()
    .collect(Collectors.groupingBy(p -> YearMonth.from(p.getDate())),
        Collectors.reducing(BigDecimal.ZERO,
            p -> p.getProducts().stream()
                .map(Product::getPrice)
                .reduce(BigDecimal.ZERO,
                    BigDecimal::add),
            BigDecimal::add)));
```

11.9 SISTEMA DE ASSINATURAS

Imagine que também oferecemos o sistema de assinatura. Pagando um valor mensal, o cliente tem acesso a todo o conteúdo de nosso e-commerce. Além do valor e do cliente, uma `Subscription` precisa ter a data de início e *talvez* uma data de término de assinatura:

```
class Subscription {
    private BigDecimal monthlyFee;
    private LocalDateTime begin;
    private Optional<LocalDateTime> end;
    private Customer customer;
}
```

Repare no uso do `Optional<LocalDateTime>`. Fica interessante para ex-
pormos o `getEnd`, tornando obrigatório o cuidado com o valor nulo por parte da
classe que utilizará essa informação. É claro que poderíamos ter usado um simples
`LocalDateTime` como atributo, e o `getter` envelopar seu valor em um `Optional`.

Teremos dois construtores. Um para assinaturas que ainda estão ativas, outro
para as que se encerraram e precisam ter uma data específica para tal:

```
public Subscription(BigDecimal monthlyFee, LocalDateTime begin,
    Customer customer) {
```

```
this.monthlyFee = monthlyFee;
this.begin = begin;
this.end = Optional.empty();
this.customer = customer;
}

public Subscription(BigDecimal monthlyFee, LocalDateTime begin,
    LocalDateTime end, Customer customer) {
    this.monthlyFee = monthlyFee;
    this.begin = begin;
    this.end = Optional.of(end);
    this.customer = customer;
}
```

Considere também os respectivos *getters* de cada atributo.

Teremos três usuários com assinaturas de 99.90. Dois deles encerraram suas assinaturas:

```
BigDecimal monthlyFee = new BigDecimal("99.90");

Subscription s1 = new Subscription(monthlyFee,
    yesterday.minusMonths(5), paulo);

Subscription s2 = new Subscription(monthlyFee,
    yesterday.minusMonths(8), today.minusMonths(1), rodrigo);

Subscription s3 = new Subscription(monthlyFee,
    yesterday.minusMonths(5), today.minusMonths(2), adriano);

List<Subscription> subscriptions = Arrays.asList(s1, s2, s3);
```

Como calcular quantos meses foram pagos através daquela assinatura? Basta usar o que conhecemos da API de `java.time`. Mas depende do caso. Se a assinatura ainda estiver ativa, calculamos o intervalo de tempo entre `begin` e a data de hoje:

```
long meses = ChronoUnit.MONTHS
.between(s1.getBegin(), LocalDateTime.now());
```

E se a assinatura terminou? Em vez de enchermos nosso código com `ifs`, tiramos proveito do `Optional`:

```
long meses = ChronoUnit.MONTHS
    .between(s1.getBegin(), s1.getEnd().orElse(LocalDateTime.now()));
```

Para calcular o valor gerado por aquela assinatura, basta multiplicar esse número de meses pelo custo mensal:

```
BigDecimal total = s1.getMonthlyFee()
    .multiply(new BigDecimal(ChronoUnit.MONTHS
        .between(s1.getBegin(),
            s1.getEnd().orElse(LocalDateTime.now()))));
```

Desta vez vamos facilitar nosso trabalho e adicionar um novo método na própria Subscription:

```
public BigDecimal getTotalPaid() {
    return getMonthlyFee()
        .multiply(new BigDecimal(ChronoUnit.MONTHS
            .between(getBegin(),
                getEnd().orElse(LocalDateTime.now()))));
}
```

Dada uma lista de subscriptions, fica fácil somar todo o total pago:

```
BigDecimal totalPaid = subscriptions.stream()
    .map(Subscription::getTotalPaid)
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

CAPÍTULO 12

Apêndice: mais Java 8 com reflection, JVM, APIs e limitações

12.1 NOVOS DETALHES NA LINGUAGEM

Operador diamante melhorado

O operador diamante (*diamond operator*) é aquele que evita digitarmos desnecessariamente em algumas situações óbvias. Por exemplo, em vez de fazer:

```
List<Usuario> lista = new ArrayList<Usuario>();
```

Podemos fazer, a partir do agora antigo Java 7:

```
List<Usuario> lista = new ArrayList<>();
```

Porém o recurso era bastante limitado: basicamente apenas podia ser usado junto com a declaração da variável referência. Considerando a existência de um re-

positório que tem o método `adiciona(List<Usuario>)`, o código a seguir falha no Java 7:

```
repositorio.adiciona(new ArrayList<>());
```

Para compilar no Java 7, você precisaria fazer `repositorio.adiciona(new ArrayList<Usuario>())`, pois o compilador não conseguia inferir o tipo declarado na assinatura do método, que neste caso é `Usuario`.

Se você conhece a API de Collections, sabe que ela possui o método `Collections.emptyList()`. Podemos utilizá-lo neste exemplo para resolver o problema sem ter que explicitamente inferir o tipo `Usuario`, visto que o compilador não fará essa inferência quando utilizamos o operador diamante.

Nosso código ficará assim:

```
repositorio.adiciona(Collections.emptyList());
```

Porém o seguinte erro será exibido ao tentar compilar esse código:

```
The method adiciona(List<Usuario>) in the type Capitulo12 is not applicable for the arguments (List<Object>)
```

Isso acontece pois o compilador inferiu o tipo genérico como sendo um `Object`. Para que esse código compile precisaríamos fazer:

```
repositorio.adiciona(Collections.<Usuario>emptyList());
```

Já no Java 8, os dois casos funcionam perfeitamente, dadas as melhorias na inferência.

Agora eu posso escrever e compilar o seguinte código sem nenhuma preocupação, afinal o compilador vai extrair essa informação genérica pelo parâmetro esperado no método.

```
repositorio.adiciona(new ArrayList<>());  
repositorio.adiciona(Collections.emptyList());
```

Você pode ler mais a respeito da proposta de melhorias na inferência dos tipos aqui:

<http://openjdk.java.net/jeps/101>

Situações de ambiguidade

Ainda que a inferência dos tipos tenha sido bastante melhorada, existem situações de ambiguidade em que precisaremos ajudar o compilador. Para entender esse problema de forma prática considere a seguinte expressão lambda:

```
Supplier<String> supplier = () -> "retorna uma String";
```

Como a interface funcional `Supplier<T>` tem apenas o método `get` que não recebe nenhum parâmetro e retorna o tipo genérico, a expressão `() -> "retorna uma String"` pode ter seu tipo inferido sem nenhum problema.

Mas essa não é a única interface funcional que atende bem a essa expressão, afinal qualquer interface pode ter um método que retorna um tipo genérico. Nós mesmos podemos criar interfaces com uma estrutura parecida.

Para esse caso nem será preciso criar um caso de ambiguidade, pois dentro da própria API do Java temos a interface `PrivilegedAction<T>` que possui essa estrutura, repare:

```
public interface PrivilegedAction<T> {
    T run();
}
```

Sabendo disso, podemos declarar a mesma expressão `() -> "retorna uma String"` com esses dois tipos!

```
Supplier<String> supplier = () -> "retorna uma String";
PrivilegedAction<String> p = () -> "retorna uma String";
```

Mas isso não será um problema para o compilador, claro, afinal essa expressão tem seu tipo bem determinado em sua declaração.

Isso nem mesmo será um problema quando passarmos a mesma expressão como parâmetro para um método como esse:

```
private void metodo(Supplier<String> supplier) {
    // faz alguma lógica e invoca supplier.get()
}
```

Já vimos que, ao executar o seguinte código, o compilador agora vai inferir o tipo da lambda de acordo com o valor esperado pelo método, que neste caso é um `Supplier`.

```
metodo(() -> "retorna uma String");
```

Mas o que acontece se esse método tiver uma sobrecarga, que recebe uma interface funcional com a mesma estrutura? Poderíamos ter uma opção desse método recebendo uma `PrivilegedAction`:

```
private void metodo(Supplier<String> supplier) {  
    // faz alguma lógica e invoca supplier.get()  
}  
  
private void metodo(PrivilegedAction<String> action) {  
    // faz alguma lógica e invoca action.run()  
}
```

Tudo bem, esse pode não ser um caso tão comum, mas é interessante saber que esse tipo de ambiguidade pode acontecer. Ao adicionar esse segundo método, o compilador não vai conseguir mais inferir o tipo dessa expressão:

```
metodo(() -> "retorna uma String");
```

Como esperado a mensagem de erro será:

```
The method metodo(Supplier<String>) is ambiguous for the type Capitulo12
```

Neste caso será necessário recorrer ao *casting* para ajudar nessa inferência. Algo como:

```
metodo((Supplier<String>) () -> "retorna uma String");
```

Conversões entre interfaces funcionais

Outro detalhe interessante sobre a inferência de tipos é que não existe uma conversão automática entre interfaces funcionais equivalentes. Por exemplo, considere que temos o método `execute` que recebe um `Supplier<String>` como parâmetro e apenas exibe o retorno de seu método `get`:

```
private void execute(Supplier<String> supplier) {  
    System.out.println(supplier.get());  
}
```

Um exemplo de seu uso seria algo como:

```
Supplier<String> supplier = () -> "executando um supplier";
execute(supplier);
```

Ao executá-lo, receberemos a saída esperada:

```
executando um supplier
```

Mas e se eu definir o tipo dessa expressão como um `PrivilegedAction<String>?` Para todos os efeitos vimos que o resultado final deverá ser o mesmo, porém, ainda que estas interfaces funcionais sejam equivalentes, não existirá uma conversão automática. O seguinte código não compila:

```
PrivilegedAction<String> action = () -> "executando uma ação";
execute(action);
```

Isso não deve funcionar, afinal o método `execute(Supplier<String>)` não é aplicável para o argumento `PrivilegedAction<String>`. Para que essa conversão seja possível podemos utilizar method reference nessa instância de `PrivilegedAction`, assim estamos explicitamente indicando ao compilador que desejamos essa conversão:

```
PrivilegedAction<String> action = () -> "executando uma ação";
execute(action::run);
```

Executando o código modificado receberemos a saída:

```
executando uma ação
```

12.2 QUAL É O TIPO DE UMA EXPRESSÃO LAMBDA?

Toda expressão lambda tem e **precisa ter** um tipo. Como vimos rapidamente, o seguinte trecho de código não funciona:

```
Object o = () -> {
    System.out.println("eu sou um runnable!");
};
new Thread(o).start();
```

Pois `Object` não é uma interface funcional. Sempre precisamos ter um tipo que seja uma interface funcional, envolvido na atribuição. Por exemplo, quando definimos esse `Runnable`:

```
Runnable r = () -> {
    System.out.println("eu sou um runnable!");
};

new Thread(r).start();
```

Fica explícito que essa expressão lambda representa a interface funcional `Runnable`, pois este é o tipo que demos para sua variável `r`. Mas e quando fazemos sua declaração de forma direta, em um único *statement*?

```
new Thread(() -> {
    System.out.println("eu sou um runnable?");
}).start();
```

Já não fica assim tão claro, não é? Mas tudo bem, sabemos que o método construtor da classe `Thread` espera receber um `Runnable`, e da mesma forma o compilador também sabe que deve traduzir esse código lambda para um `Runnable`.

O compilador é o responsável por inferir qual o tipo de uma expressão lambda, e para conseguir fazer esse trabalho ele agora leva em consideração o **contexto** em que essa expressão foi aplicada, ou seja, infere o tipo de acordo com o tipo que é esperado pelo método ou construtor. Esse “tipo esperado” é conhecido como *Target Type*.

Conhecer o contexto foi um passo muito importante para o compilador poder inferir o tipo de uma expressão lambda, pois como já vimos, uma mesma expressão pode ter tipos diferentes:

```
Callable<String> c = () -> "retorna uma String";
PrivilegedAction<String> p = () -> "retorna uma String";
```

Na primeira linha desse código, a expressão `() -> "retorna uma String"` representa um `Callable<String>`, e na segunda linha essa mesma expressão representa a interface funcional `PrivilegedAction<String>`. Foi o `Target Type` quem ajudou o compilador a decidir qual tipo essa mesma expressão representava em cada momento.

O mesmo vale para method references

Essa situação também acontece com o recurso de method reference. A mesma referência pode representar tipos diferentes, contanto que sejam equivalentes:

```
Callable<String> c = callable::call;
PrivilegedAction<String> action = callable::call;
```

Como vimos, estes dois tipos possuem estrutura bem parecida, portanto a referência `callable::call` pode ser representada como um `PrivilegedAction<String>`.

A grande diferença é que quando utilizamos method reference a inferência de tipo é mais forte, afinal o tipo está explícito em sua declaração, diferente de quando estamos trabalhando com uma expressão lambda. Outro ponto é que, como já vimos, existe a conversão entre interfaces funcionais.

12.3 LIMITAÇÕES DA INFERÊNCIA NO LAMBDA

Talvez você vá se deparar com esses problemas bem mais pra frente, mas é importante ter conhecimento de que eles existem: algumas vezes o compilador não consegue inferir os tipos com total facilidade, em especial quando encadeamos métodos que utilizam lambda.

Veja que com method references já vimos que isso compila:

```
usuarios.sort(Comparator.comparingInt(Usuario::getPontos)
              .thenComparing(Usuario::getNome));
```

Porém, usando os lambdas que parecem equivalentes, não compila:

```
usuarios.sort(Comparator.comparingInt(u -> u.getPontos())
              .thenComparing(u -> u.getNome));
```

Se avisarmos que o `u` é um `Usuario`, o retorno do `comparingInt` fica mais óbvio para o compilador, e com o código a seguir conseguimos uma compilação sem erros:

```
usuarios.sort(Comparator.comparingInt((Usuario u) -> u.getPontos())
              .thenComparing(u -> u.getNome));
```

Outra forma seria quebrar a declaração do `Comparator`, perdendo a interface fluente:

```
Comparator<Usuario> comparator = Comparator.comparing(
    u -> u.getPontos());
usuarios.sort(comparator.thenComparing(u -> u.getNome));
```

Você também poderia forçar o tipo genérico do método estático, fazendo `Comparator.<Usuario>comparingInt`, que é uma sintaxe pouco vista no dia a dia.

Repare que tudo funcionou perfeitamente quando utilizamos o method reference, já que com eles o compilador pode facilmente dizer quais são os tipos envolvidos.

O mesmo acontece para casos com encadeamentos de interfaces fluentes mais simples. Você pode fazer, como visto:

```
usuarios.sort(Comparator.comparingInt(Usuario::getPontos).reversed());
```

Mas se usar o lambda em vez do method reference, não compila:

```
usuarios.sort(Comparator.comparingInt(u -> u.getPontos()).reversed());
```

Precisaria explicitar os tipos genéricos ou o tipo no lambda. Ou ainda declarar o Comparator antes e só depois invocar o reversed. Como utilizamos com mais frequência o method reference nesses encadeamentos, esses serão casos raros.

Essas limitações são muito bem detalhadas na nova versão da Java Language Specification:

<http://cr.openjdk.java.net/\char126dsmith/jsr335-0.8.0/D.html>

Suporte a múltiplas anotações iguais

Um detalhe que pode ser bastante útil em nosso dia a dia é a capacidade de aplicar a mesma anotação repetidas vezes em um mesmo tipo. Essa possibilidade, conhecida como *repeating annotations*, foi adicionada na nova versão da linguagem!

Considere que temos a anotação @Role para determinar o tipo de acesso permitido para uma classe:

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE})  
public @interface Role {  
    String value();  
}
```

Para adicionar duas roles em uma mesma classe nas versões pré-Java 8 teríamos que modificar sua estrutura pra receber um array de Strings, ou qualquer outro tipo que represente uma regra de acesso de seu sistema.

Ao tentar declarar duas vezes a anotação, para talvez melhorar um pouco a legibilidade, receberíamos o erro de compilação: Duplicate annotation @Role.

```
@Role("presidente")
@Role("diretor")
public class RelatorioController {  
}
```

Para tornar isso possível agora podemos marcar a nossa anotação com `@Repeatable`, passando como argumento uma outra anotação que servirá para armazenar as anotações repetidas. Neste caso:

```
@Repeatable(Roles.class)
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE})
public @interface Role {
    String value();
}
```

Onde `@Roles` conterá um array da anotação que ela armazenará:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE})
public @interface Roles {

    Role[] value();
}
```

Tudo pronto! Agora o seguinte código compila:

```
@Role("presidente")
@Role("diretor")
public class RelatorioController {  
}
```

Aplicamos as anotações repetidas em uma classe neste exemplo, porém o mesmo pode ser feito em qualquer lugar passível de utilizar uma anotação.

Diversos métodos foram adicionados na API de reflection para recuperar essas anotações. Um exemplo é o método `getAnnotationsByType` que recebe a classe da anotação procurada como parâmetro:

```
RelatorioController controller = new RelatorioController();

Role[] annotationsByType = controller
    .getClass()
    .getAnnotationsByType(Role.class);

Arrays.asList(annotationsByType)
    .forEach(a -> System.out.println(a.value()));
```

Como saída, teremos os valores inseridos nas roles:

```
presidente
diretor
```

12.4 FIM DA PERMGEN

Um ponto que não pode passar despercebido nessa nova versão é a remoção completa da PermGem (*Permanent Generation*), conhecida implementação de espaço de memória da JVM reservada para os valores permanentes. Isso significa que nunca mais receberemos o famoso `java.lang.OutOfMemoryError: PermGen!`

Você pode ler mais sobre a proposta (*JEP*) dessa remoção aqui:

Remove the Permanent Generation <http://openjdk.java.net/jeps/122>

Como sucessor desta implementação agora temos o Metaspace, que é a memória nativa para representação dos metadados de classe. Você já pode ter visto uma solução semelhante no JRockit da Oracle, ou JVM da IBM.

Para conhecer mais detalhes avançados sobre performance e particularidades dessa migração você pode se interessar pelo seguinte artigo:

<http://java.dzone.com/articles/java-8-permgen-metaspace>

12.5 REFLECTION: PARAMETER NAMES

A API de reflection também recebeu melhorias. Uma delas foi a tão esperada habilidade de recuperar o nome dos parâmetros dos métodos e construtores.

Isso agora pode ser feito de forma muito simples: basta invocarmos o método `getParameters` de um método ou construtor. Podemos fazer isso com o seguinte construtor da classe `Usuario`:

```
public Usuario(String nome, int pontos) {
    this.pontos = pontos;
```

```
    this.nome = nome;  
}
```

Vamos utilizar o método `getConstructor` fornecendo os tipos dos parâmetros para recuperar esse construtor com reflection:

```
Constructor<Usuario> constructor =  
    Usuario.class.getConstructor(String.class, int.class);
```

Agora tudo que precisamos fazer é chamar o método `getParameters` para obter como resposta um array do tipo `java.lang.reflect.Parameter`:

```
Constructor<Usuario> constructor =  
    Usuario.class.getConstructor(String.class, int.class);  
  
Parameter[] parameters = constructor.getParameters();
```

Pronto! Agora de forma bem intuitiva podemos exibir seu nomes com o método `getName`:

```
Constructor<Usuario> constructor =  
    Usuario.class.getConstructor(String.class, int.class);  
  
Parameter[] parameters = constructor.getParameters();  
  
Arrays.asList(parameters)  
.forEach(param -> System.out.println(param.getName()));
```

Mas note que ao executar esse método o resultado ainda será:

```
arg0  
arg1
```

A informação dos nomes não está presente nesse construtor! Para confirmar isso, podemos imprimir também o resultado do método `isNamePresent` em cada parâmetro:

```
Constructor<Usuario> constructor =  
    Usuario.class.getConstructor(String.class, int.class);  
  
Parameter[] parameters = constructor.getParameters();
```

```
Arrays.asList(parameters)
    .forEach(param -> System.out.println(
        param.isNamePresent() + ": " + param.getName())
    );
```

Rpare que o resultado será `false` para cada parâmetro:

```
false: arg0
false: arg1
```

Isso aconteceu pois precisamos passar o argumento `-parameters` para o compilador! Apenas dessa forma ele vai manter os nomes dos parâmetros no *byte code* para recuperarmos via reflection. Agora sim, ao recompilar a classe teremos o seguinte resultado:

```
true: nome
true: pontos
```

Antes do Java 8 isso também era possível ao compilar nosso código com a opção `-g` (debug), mas o problema é que neste caso precisamos manipular o *byte code* das classes e isso pode ser bastante complicado. Existem alternativas para simplificar esse trabalho, como é o caso da biblioteca **Paranamer**. Repare em um exemplo de seu uso:

```
Constructor<Usuario> constructor =
    Usuario.class.getConstructor(String.class, int.class);

Parameter[] parameters = constructor.getParameters();

Paranamer paranamer = new CachingParanamer();

String[] parameterNames = paranamer.lookupParameterNames(constructor);
```

Uma grande vantagem dessa adição na API de reflection é que, além de simplificar bastante nosso código, evita a necessidade de dependências extras em nossos projetos.

Você pode ler mais sobre a proposta dessa feature no link:

<http://openjdk.java.net/jeps/118>

CAPÍTULO 13

Continuando seus estudos

Existem diversas formas para você continuar seus estudos.

Uma delas é exercitando os códigos utilizados nos capítulos do livro, que podem ser encontrados no repositório:

<https://github.com/peas/java8>

Para criar uma intimidade maior com as novidades da API e essa nova forma funcional de programar em Java, recomendamos que você exercente bastante os exemplos desse livro e que vá além, fazendo novos testes além dos sugeridos.

13.1 COMO TIRAR SUAS DÚVIDAS

Encaminhe suas dúvidas ou crie tópicos para discussão na lista que foi criada especialmente para esse livro:

<https://groups.google.com/forum/#!forum/java8-casadocodigo>

Além de perguntar, você também pode contribuir com sugestões, críticas e melhorias para nosso conteúdo. Envie seus experimentos e códigos.

Você sempre pode tirar suas dúvidas também pelo fórum do GUJ:
<http://www.guj.com.br/tag/java>

13.2 BIBLIOTECAS QUE JÁ USAM OU VÃO USAR JAVA 8

Atualmente existem algumas bibliotecas utilizando Java 8! Uma exemplo é a *GS Collections*, do Goldman Sachs, que oferece diversas classes utilitárias e uma forma simples de trabalhar com as coleções do jdk de forma compatível.

Você pode ver mais detalhes dessa biblioteca e consultar seu código-fonte no repositório:

<https://github.com/goldmansachs/gs-collections/>

Há algumas brincadeiras interessantes, como a implementação de sequências lazy, no melhor estilo funcional:

<http://java.dzone.com/articles/lazy-sequences-implementation>

O framework VRaptor também vai utilizar jdk 8 em sua versão 4.5! Por enquanto, alguns plugins estão sendo desenvolvidos para adicionar alternativas elegantes para quem já usa essa versão da linguagem em seus projetos.

Índice Remissivo

- ActionListener, [13](#)
- andThen, [21](#)
- anotações, [120](#)
- autoboxing, [29](#), [33](#), [63](#)
- average, [59](#)
- BiConsumer, [47](#)
- BiFunction, [38](#)
- boxed, [65](#)
- Calendar, [83](#)
- ChronoUnit, [92](#)
- close, [63](#)
- closure, [17](#)
- Collector, [47](#)
- Collectors, [48](#)
- Comparator, [25](#)
- comparing, [27](#)
- computeIfAbsent, [75](#)
- construtor, [36](#)
- Consumer, [7](#), [20](#)
- Date, [83](#)
- decorator, [21](#)
- default methods, [19](#)
- distinct, [70](#)
- Duration, [94](#)
- efeito colateral, [44](#), [80](#)
- especificação, [2](#), [17](#), [23](#), [84](#), [94](#), [114](#)
- exceptions, [67](#)
- factory, [63](#)
- Fibonacci, [65](#)
- Files, [67](#)
- filter, [43](#)
- final, [17](#)
- findAny, [57](#)
- findFirst, [58](#)
- flatMap, [68](#)
- fold, [61](#)
- forEach, [7](#), [19](#)
- fork/join, [78](#)
- formatters, [89](#)
- Function, [37](#), [50](#), [73](#)
- FunctionalInterface, [15](#)
- generate, [64](#)
- groupingBy, [75](#)
- herança múltipla, [23](#)
- imutabilidade, [86](#)
- IntBinaryOperator, [38](#), [60](#)
- interface fluente, [45](#)
- interface funcional, [11](#)
- IntStream, [50](#)
- invokedynamic, [17](#)
- iterate, [66](#)
- iterator, [61](#)

- JEP, 2, 114
Joda time, 83
lambda, 9, 17
lazy, 56–58, 65
limit, 64
limitações, 119
LocalDate, 84
LocalDateTime, 84
LongStream, 72
métodos default, 19
Map, 72, 75
map, 50, 54
mapToInt, 50
mapToLong, 72
max, 53
metaspace, 122
method references, 31
naturalOrder, 27
operações de curto circuito, 64
operações determinísticas, 79
operações intermediárias, 57
operações terminais, 57
operador diamante, 113
paralelização, 78
paralelização performance, 78
parameter names, 122
Paranamer, 124
partitioningBy, 75
Paths, 67
peek, 58
Period, 92
permgen, 122
pipeline, 56
Predicate, 22, 43, 62
random, 64
redução, 59
reduce, 60
reflection, 122
removeIf, 22
reversed, 34
Runnable, 11
skip, 64
sort, 55
Spliterator, 81
static, 27, 38
Stream, 42
stream infinito, 64
sum, 59
summing, 77
super, 38
Supplier, 37, 47, 64
toArray, 49
toCollection, 49
ToIntBiFunction, 38
ToIntFunction, 29, 50
toList, 48
toMap, 73
toSet, 49
UnaryOperator, 66
Validador, 14
ZonedDateTime, 85