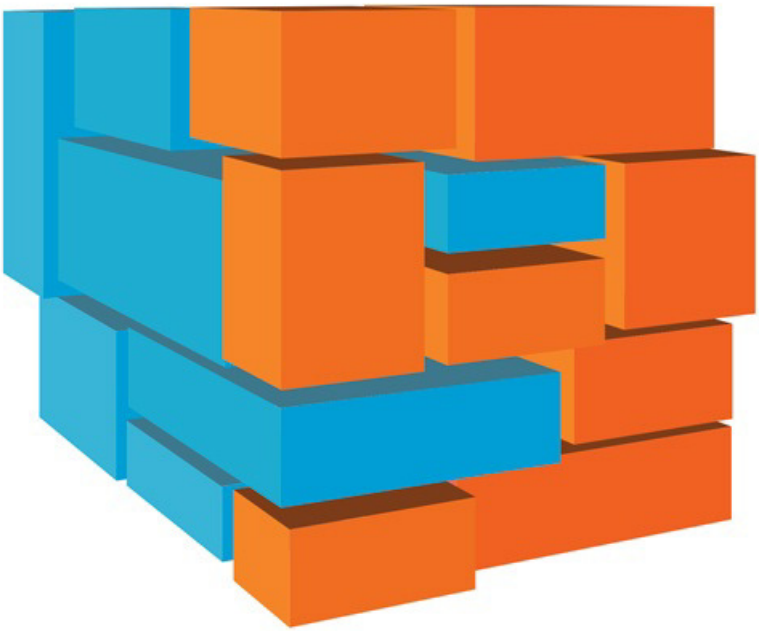
Java 9

Interativo, reativo e modularizado



Casa do Código

SÉRIE CAELUM

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.

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

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

www.casadocodigo.com.br

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

ISBN

Impresso e PDF: 978-85-94188-09-0

EPUB: 978-85-94188-10-6

MOBI: 978-85-94188-11-3

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

DEDICATÓRIA

"Para as famílias Turini, Ferreira e Bertoldo, e ao elo especial que as une: minha filha Clara"

— Rodrigo Turini

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

AGRADECIMENTOS

Escrever um livro sobre uma tecnologia que ainda está em desenvolvimento é um desafio muito grande.

Foi com a ajuda das valiosas sugestões e testes do Phil Ehlert, Lucas Ferreira e Lucas Romão que este projeto se tornou realidade. Um agradecimento especial ao Alexandre Aquiles, que teve envolvimento direto na construção do conteúdo, escolha do nome e demais aspectos do livro. Sem a revisão técnica, sugestões e opinião de todos vocês, este livro certamente não seria o mesmo.

Não posso deixar de agradecer à Vivian Matsui e à Bianca Hubert, que revisaram o português e a didática de todo o texto com a mesma velocidade e carinho com qual ele foi criado. É impressionante o cuidado e atenção que vocês têm pelo conteúdo e processos da editora.

Ao Paulo Silveira, pela mentoria diária e incentivo na escrita de meu primeiro livro, curiosamente sobre Java 8.

Também a você, leitor, que foi para quem escrevi e escolhi cuidadosamente todo o conteúdo, pensando sempre em como ele poderia ser aplicado no dia a dia de sua carreira.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

SOBRE O AUTOR

Meu nome é Rodrigo Turini e sou responsável pela tecnologia por trás da Alura, a plataforma de ensino online do grupo Caelum. Já ocupei diferentes posições nos projetos da empresa, mas foi como líder de desenvolvimento que consegui perceber na prática o impacto que uma boa arquitetura e o encapsulamento de implementações internas — assuntos extensivamente abordados aqui no livro — têm na evolução e manutenção de código.

Apesar de programar em diferentes linguagens, sempre tive um envolvimento muito grande com o Java. Sou integrante ativo do *Java Community Process* (JCP) e tenho participação em diferentes *expert groups* do Java EE. Também sou responsável pelo VRaptor, um framework open source para desenvolvimento web. Foram nesses projetos que conheci o outro lado da moeda, dos *maintainers* de APIs e bibliotecas.

Neste livro, compartilho não só as novidades do Java 9, mas também um pouco da minha experiência e das boas práticas que tenho estudado com afinco desde que entrei no universo da programação e da orientação a objetos.

Meu objetivo é mostrar o quanto os novos recursos e a adoção de abordagens funcionais, reativas e modulares certamente vão mudar sua forma de programar na linguagem.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

COMO O LIVRO FOI ORGANIZADO

Este livro é dividido em fases. Nos primeiros capítulos, você conhecerá o novo ambiente interativo do Java, chamado *JShell*, e com ele explorará algumas das principais evoluções nas APIs existentes, além de implementar funcionalidades utilizando o *HTTP/2 Client* e a nova *Flow API*, que trazem suporte à elegante solução de *Reactive Streams* para a linguagem.

O passo seguinte será importar todo o código construído nos capítulos anteriores para dentro de um novo projeto que será criado, com o nome `bookstore`. Com isso, você terá o ambiente perfeito para conhecer o conceito-chave do JDK 9, que é o sistema de módulos. O projeto tradicional será migrado para o modular, e assim você perceberá de forma prática as principais diferenças e vantagens dessa nova possibilidade.

Em outras palavras, o livro começa no interativo, parte para o reativo e, por fim, ao modular. Dessa forma, você passa por cada uma das marcantes características da nova versão da linguagem.

RESUMO DO CONTEÚDO

A lista a seguir mostra um resumo dos principais recursos e tópicos abordados em cada um dos capítulos do livro.

- **Capítulo 1 — A vida antes e depois do Java 8:** relembra as principais introduções da versão anterior da linguagem, como: *Lambda*, *method reference*, *default methods* e *Streams*. Também são mencionadas

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

as principais novidades do Java 9, além de recursos esperados e que ficaram de fora.

- **Capítulo 2 — JShell: Java 9 interativo:** apresenta a nova ferramenta de REPL (*Read-Eval-Print Loop*), que traz uma forma muito mais eficiente de executar código com feedback imediato. Chega de classes de teste com o método *main*! Com ele, a experiência de executar códigos rápidos na linguagem é completamente renovada, desde o primeiro contato do iniciante até a exploração de novas APIs para usuários avançados.

- **Capítulo 3 — Atualizações nas principais APIs:** mostra o que mudou nas *Collections*, *Streams*, *Optionals* e outros. Entre as principais novidades está o destaque para novos *default methods*, que oferecem formas mais sucintas de se criar mapas e coleções.

- **Capítulo 4 — HTTP/2 Client API:** apresenta o conceito de *incubator modules*, que promete uma evolução mais rápida e estratégica na plataforma! Vamos conhecer seu primeiro candidato, uma API completamente repaginada com suporte para requisições HTTP/2.

- **Capítulo 5 — Reactive Streams:** mostra a evolução arquitetural do pacote `java.util.concurrent` , que introduz uma nova API de fluxos reativos. Nesse ponto, você verá como resolver problemas avançados em execuções assíncronas com *back pressure*.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

- **Capítulo 6 — Juntando as peças:** é o momento em que vamos juntar todas as implementações dos capítulos anteriores em um novo projeto, chamado `bookstore`, ainda com o formato tradicional das antigas versões da linguagem. Aqui os processos de compilação e execução serão revistos, além da, até então, atual abordagem do *classpath*, para que as diferenças da nova versão fiquem bastante evidentes.

- **Capítulo 7 — Java modular:** expõe os principais problemas da antiga estrutura monolítica da plataforma Java, junto com sua solução: um novo sistema de módulos e a modularização do próprio JDK. O projeto criado no capítulo anterior será migrado e você perceberá na prática cada uma das características e vantagens dessa nova possibilidade.

- **Capítulo 8 — Criando imagens de execução customizadas:** explicará quando e como você pode criar ambientes de execução mais leves, com apenas um pedacinho do JRE que a aplicação precisa para executar. Você também verá como analisar as dependências utilizadas em cada ponto de seu código.

- **Capítulo 9 — Evolução dos JARs no JDK 9:** mostrará tudo o que muda na estrutura de um JAR modular, além da nova possibilidade de ter um único JAR que suporta diferentes versões do Java.

- **Capítulo 10 — Logging API:** discute a grande motivação por trás da nova API de *logging* do JDK 9, que nos possibilita criar um provedor padrão de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

mensagens que pode ser usado tanto em seu código como no do próprio JDK.

- **Capítulo 11 — Stack-Walking API:** mostra o essencial sobre a nova API que nos permite passear pela *Stack* das aplicações, de forma extremamente eficiente. Detalhes sobre a retenção de informações da máquina virtual serão comentados, para que você saiba quando e como tirar proveito dessas opções.

- **Capítulo 12 — Mais Java 9, APIs e outras mudanças:** apresenta mais alguns dos detalhes que mudaram na linguagem e plataforma. *Milling Project Coin*, evolução do *deprecated*, melhorias de performance e o uso do *Garbage Collector* G1 como implementação padrão são alguns deles.

- **Capítulo 13 — Continue seus estudos:** é o ponto final de nosso jornada, indicando os possíveis próximos passos para que você aproveite ao máximo o conteúdo deste livro.

PARA QUEM É ESTE LIVRO

O livro é voltado para desenvolvedores que já conhecem a linguagem Java, e querem se atualizar com as novidades dessa versão.

Se ainda não conhece os recursos do JDK 8, certamente vai se interessar em começar pelo livro *Java 8 Prático*, que escrevi junto com o Paulo Silveira. Nele damos um overview sobre *Lambdas*, *Streams* e os novos recursos da versão anterior da linguagem. Para

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

mais, acesse: https://www.casadocodigo.com.br/products/livro-java8.

É importante que você já tenha uma base sólida sobre essa versão do Java antes de prosseguir com o estudo dos novos recursos.

PROJETO E EXEMPLOS DE CÓDIGO

Todos os trechos de código deste livro podem ser encontrados no seguinte repositório, que está organizado em diretórios com o nome de cada um dos capítulos aqui listados.

https://github.com/Turini/livro-java-9

Você pode preferir digitar todos os exemplos ou, desde já, baixar o conteúdo completo e ir acompanhando o livro com o seu editor preferido aberto. Independente da escolha, não deixe de executar todos os exemplos para se adaptar com as mudanças da linguagem e, sempre que possível, fazer novos testes além dos aqui vistos.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Sumário

1 A vida antes e depois do Java 8 **1**

1.1 Que venha o Java 9 6

1.2 Propostas que ficaram de fora 7

1.3 Acesse e exercite todo o código visto 9

1.4 Entre em contato conosco 9

1.5 Instalando o Java 9 9

2 JShell: Java 9 interativo **11**

2.1 JShell, o REPL do Java 12

2.2 Iniciando o JShell 13

2.3 O clássico hello world com REPL 15

2.4 Declarando variáveis explicitamente 16

2.5 Declarando variáveis de forma implícita 17

2.6 Ajustando os níveis de feedback 18

2.7 Métodos, classes e instruções mais complexas 19

2.8 Adicionando imports 23

2.9 Atalhos, truques e auto-completion 25

2.10 Explorando novas APIs 26

2.11 Mais detalhes e configurações 28

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

2.12 Download dos exemplos deste capítulo 29

3 Atualizações nas principais APIs 30

3.1 Melhorias nas Collections 31

3.2 Atualizações nos Streams 38

3.3 Novos Collectors 46

3.4 Atualizações no Optional 54

3.5 Mais novidades 61

3.6 Download dos exemplos deste capítulo 61

4 HTTP/2 Client API 63

4.1 O primeiro dos incubator modules 63

4.2 Preparando seu ambiente 64

4.3 Requests HTTP da forma antiga 64

4.4 O novo HTTP/2 Client API 66

4.5 Lidando com redirects 71

4.6 Configurando o HttpClient 72

4.7 Buscando livros com HTTP/2 client 74

4.8 Requisições assíncronas 80

4.9 Criando um arquivo CSV no retorno 83

4.10 Outras novidades do HTTP/2 Client API 85

4.11 Download dos exemplos deste capítulo 86

5 Reactive Streams 87

5.1 Emissão de notas fiscais 89

5.2 Primeiro contato com fluxo reativo 93

5.3 Criando seu próprio Subscriber 103

5.4 Entendendo o Processor 111

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

5.5 Download dos exemplos deste capítulo 116

6 Juntando as peças 117

6.1 Criando um novo projeto 118

6.2 Listando livros com HTTP/2 client 121

6.3 Integrando a emissão de notas fiscais 127

6.4 Interagindo com a lista de livros 132

6.5 Revendo o projeto e responsabilidades 137

6.6 Download do projeto completo 138

7 Java Modular 139

7.1 Antes do Java 9 139

7.2 Que venha o modular 142

7.3 Modularizando a bookstore 143

7.4 Trabalhando com diferentes módulos 152

7.5 O JDK modular 178

7.6 Para saber mais 186

7.7 Download do projeto completo 186

8 Criando imagens de execução customizadas 187

8.1 Criando uma imagem de execução customizada 190

8.2 Criando a imagem a partir das definições do projeto 195

8.3 Análise de dependências dos módulos 196

8.4 Encontrando detalhes dos módulos e dependências via Reflection 200

8.5 Download das JREs criadas 202

9 Evolução dos JARs no JDK 9 203

9.1 Empacotando projetos modulares 203

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

9.2 Executando projetos modulares 206
9.3 Multi-Release JAR files 208
9.4 Para saber mais 214

10 Logging API 216
10.1 Criando um módulo de logs 218
10.2 Logs da JVM e Garbage Collector 239
10.3 Download do projeto completo 240

11 Stack-Walking API 242
11.1 Novo módulo de Tracking 244
11.2 Stack-Walking API 252
11.3 Criando Stacks com mais informações 257
11.4 Download do projeto completo 258

12 Mais Java 9, APIs e outras mudanças 259
12.1 Quebra de compatibilidade? 266
12.2 Strings mais leves e eficientes 269
12.3 Garbage Collectors e o G1 270
12.4 Download dos exemplos deste capítulo 271

13 Continuando seus estudos 272
13.1 Como tirar suas dúvidas 272
13.2 E agora? 273

Versão: 21.2.21

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 1

A VIDA ANTES E DEPOIS DO JAVA 8

A linguagem Java sempre foi conhecida pela característica de manter a compatibilidade com suas versões anteriores. Isso a torna muito interessante do ponto de vista de manutenção, mas essa retrocompatibilidade sempre veio ao custo de limitar a introdução de novos recursos, criação de novos métodos em interfaces e implementação de outras funcionalidades que implicariam em mudanças de bytecode.

Os *default methods* do Java 8 abriram caminho para a evolução estratégica de suas APIs existentes, sem que houvesse quebra de compatibilidade. Interfaces agora podem ter métodos com uma implementação default, e isso possibilitou que finalmente a interface `List` tivesse métodos como o `sort`, `forEach` e outros, sem que todas as suas implementações nos projetos e bibliotecas existentes fossem quebradas.

A versão também trouxe novas APIs e conceitos-chaves para a evolução do Java, que aderiu um estilo mais funcional e declarativo. Quer um exemplo prático? Considere a classe `Book` a seguir:

```
public class Book {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  private final String name;
  private final String author;

  // outros atributos

  public Book (String name, String author) {
    this.name = name;
    this.author = author;
  }

  // getters e outros métodos

  public String getName() {
    return name;
  }

  public boolean hasAuthor(String name) {
    return author.contains(name);
  }

  @Override
  public String toString() {
  return "\nlivro: " + name
      +  "\nautor: " + author;
  }
}
```

E uma lista com alguns livros de diferentes autores:

```
List<Book> allBooks = new ArrayList<>();

allBooks.add(
  new Book("Desbravando Java", "Rodrigo Turini"));

allBooks.add(
  new Book("APIs Java", "Rodrigo Turini"));

allBooks.add(
  new Book("Java 8 Prático", "Rodrigo Turini, Paulo Silveira"));

allBooks.add(
  new Book("TDD", "Mauricio Aniche"));
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
allBooks.add(
  new Book("Certificação Java", "Guilherme Silveira"));
```

Para filtrar todos os livros cujo autor sou eu, e retorná-los ordenados pelo seus nomes, faríamos assim com o Java 7:

```
List<Book> filteredBooks = new ArrayList<>();

for (Book book: allBooks) {
  if (book.hasAuthor("Rodrigo Turini")) {
    filteredBooks.add(book);
  }
}

Collections.sort(filteredBooks,
new Comparator<Book>() {
  public int compare(Book b1, Book b2) {
    return b1.getName().compareTo(b2.getName());
  }
});

for (Book book: filteredBooks) {
  System.out.println(book);
}
```

Esse é o famoso problema vertical do Java! São várias linhas de código para executar operações simples, como filtros e ordenações. Imagine que eu queira agora mostrar apenas os livros de Java, ou filtrar por outros critérios além do nome do autor: seriam `ifs` e mais `ifs` !

Veja agora o mesmo código escrito em Java 8, com o uso da API de *Streams*, expressões *lambda* e do *method reference*:

```
allBooks.stream()
  .filter(book -> book.hasAuthor("Rodrigo Turini"))
  .sorted(comparing(Book::getName))
  .forEach(System.out::println);
```

Sem `ifs` , sem variáveis intermediárias e sem todo o

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

boilerplate de antes. Nosso código não só ficou mais enxuto, como simples e de fácil legibilidade.

O resultado da execução de ambos os casos será:

```
livro: APIs Java
autor: Rodrigo Turini

livro: Desbravando Java
autor: Rodrigo Turini

livro: Java 8 Prático
autor: Rodrigo Turini, Paulo Silveira
```

E não foram apenas as *collections* que receberam melhorias. A versão também introduziu uma nova API de datas, que resolveu vários dos problemas de design e mutabilidade dos modelos de `Date`, `Calendar` e do próprio *Joda-Time* (biblioteca na qual a API foi inspirada).

Lembra de como fazíamos para criar datas no passado com `Calendar`? Você tinha de adicionar -1 dia para criar a data de ontem, por exemplo, pela ausência de um método para retroceder dias:

```
Calendar yesterday = Calendar.getInstance();
yesterday.add(Calendar.DATE, -1);
```

Com o Java 8 isso passou a ser feito assim, de forma fluente e em um modelo imutável do pacote `java.time`:

```
LocalDateTime.now().minusDays(1);
```

Você também consegue descobrir o intervalo entre uma data e outra de uma forma bem simples, utilizando o *enum* `ChronoUnit` dessa mesma API:

```
ChronoUnit.DAYS.between(yesterday, now());
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Esses são apenas alguns dos muitos benefícios dessa nova API.

No Java 8, também surgiram os `Optional` s apresentando uma forma mais interessante de representar valores opcionais, além de permitir a manipulação de seus dados com uma abordagem um tanto elegante. Um exemplo seria o próprio método `hasAuthor` , que usamos na classe `Book` . Se o atributo `author` do livro for `null` , o resultado seria um evidente `NullPointerException` .

É sempre interessante programar na defensiva, precavendo-nos de possíveis problemas como esse, portanto poderíamos fazer um *early return* para contornar o caso:

```
public boolean hasAuthor(String name) {

    if (author == null) {
        return false;
    }
    return this.author.contains(name);
}
```

Até então, o código está simples, mas conforme outras necessidades de filtros aparecem, novas condições surgem junto com novos `ifs` . O `Optional` troca essa forma imperativa de resolver problemas por uma abordagem mais declarativa. Se existe a possibilidade de o atributo ser opcional, ele poderia ser declarado da seguinte forma:

```
Optional<String> author;
```

E o nosso código ficaria assim:

```
public boolean hasAuthor(String name) {

  return author
    .filter(s -> s.contains(name))
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    .isPresent();
}
```

O `filter` e o `isPresent` deixam a intenção do código bem clara: filtrar o valor e conferir se ele existe. E o grande benefício aqui vai além da sintaxe diferenciada e dos nomes significativos. Quando você trabalha com um `Optional`, você não precisa se lembrar de que aquele valor pode não existir e que você precisa tratar essa condição de alguma forma, já que o compilador faz isso para você.

Essas são só algumas das muitas novidades que fizeram do Java 8 um dos principais releases da linguagem, que já tem mais de 20 anos desde o seu primeiro lançamento. Se alguma parte desse código pareceu novidade para você, certamente vai se interessar em começar pelo livro *Java 8 Prático*, que escrevi junto com o Paulo Silveira. Nele damos um overview sobre *Lambdas*, *Streams* e os novos recursos da versão anterior da linguagem. Para mais, acesse: https://www.casadocodigo.com.br/products/livro-java8.

Ressalto que é importante que você já tenha uma base sólida sobre essa versão do Java antes de prosseguir com o estudo dos novos recursos.

1.1 QUE VENHA O JAVA 9

São muitas as novidades dessa nova versão. Um destaque especial vai para o *Jigsaw*, o tão esperado sistema de módulos, e a modularização do próprio JDK. Esse é um plano que começou desde muito antes do Java 7, foi uma possibilidade no Java 8 e, por fim, depois de muitas revisões e turbulências, virou uma realidade no Java 9.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Outra adição bastante significativa da linguagem foi o *JShell*, uma ferramenta de REPL (*Read-Eval-Print Loop*). Se você já programa em outras linguagens como Scala, Ruby e JavaScript, possivelmente já brincou com algum tipo de REPL. Ele nada mais é do que um ambiente interativo para você executar códigos Java, sem a necessidade de criar as clássicas classes de teste com o método `main`. Com ele, fica muito mais simples e divertido experimentar as novas APIs, como faremos quase que à exaustão aqui no livro.

HTTP/2 Client, *Streams Reativos* e as novas APIs de *Logging* e de *Stack-Walking* são outros exemplos de novidades que vamos explorar, além de surpresas como métodos privados em interfaces, arquivos de *properties* finalmente com suporte a *UTF-8*, e muitos novos métodos *default* das APIs de *Collection*, *Stream* e outros.

Tudo isso será extensivamente praticado durante o livro.

O seguinte link possui detalhes sobre o cronograma de lançamento e a lista completa das novas funcionalidades do JDK 9:

http://openjdk.java.net/projects/jdk9/

E além dele, o documento *API Specification Change Summary* lista em detalhes todas essas novidades e alterações:

http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-annex-1.html

1.2 PROPOSTAS QUE FICARAM DE FORA

A especificação do Java é dividida em várias *JEPs* (*JDK Enhancement Proposals*). Elas nada mais são do que propostas de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

tarefas, que podem ser mudanças, correções ou novas funcionalidades para a plataforma. Essa é uma ideia que nasceu dos PEPs, proposta similar da comunidade Python.

A JEP 0 é uma lista com todas as propostas:

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

Como você pode ver no link, são muitas! Nem todas vingaram e algumas, apesar de previstas para uma versão específica da linguagem, acabam ficando de fora. Isso é muitas vezes necessário para permitir maior tempo de desenvolvimento, revisão e testes. Foi assim com o *Jigsaw*, por exemplo, que vem sendo considerado há bem mais de 10 anos.

Como vimos na lista completa de funcionalidades, muitas novidades entraram no JDK 9.

Infelizmente, nem todas tiveram tempo suficiente para amadurecer. Entre as JEPs, a polêmica proposta de *Local-Variable Type Inference* ficou de fora:

http://openjdk.java.net/jeps/286

A feature permitiria, por exemplo, declarações deste tipo:

```
var list = new ArrayList<String>();
var stream = list.stream();
```

Repare no uso do `var`, que faria a inferência automática dos tipos `ArrayList<String>` e `Stream<String>`. Interessante, não é? Apesar disso, aposto que metade dos leitores está se contorcendo com a mera possibilidade.

Value Objects e *Collections Literals* são mais exemplos de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

propostas que vêm sendo adiadas de uma versão para a outra. Ainda assim, a evolução da linguagem é inquestionável. Muitas propostas interessantes entraram nessa nova versão e você verá as principais aqui no livro.

1.3 ACESSE E EXERCITE TODO O CÓDIGO VISTO

Todos os exemplos de código deste capítulo podem ser encontrados em:

https://github.com/Turini/livro-java-9

1.4 ENTRE EM CONTATO CONOSCO

Ficou com alguma dúvida? Não deixe de nos enviar. O fórum da Casa do Código foi criado exclusivamente para facilitar o seu contato conosco e com os demais leitores:

http://forum.casadocodigo.com.br/

Suas sugestões, críticas e melhorias serão muito mais do que bem-vindas!

Outro recurso que você pode usar para esclarecer suas dúvidas e participar ativamente na comunidade Java é o fórum do GUJ.

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

1.5 INSTALANDO O JAVA 9

Para continuar a partir daqui, você precisará baixar e instalar o

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

JDK 9:

http://jdk.java.net/9/

Após concluir, confira que o ambiente está preparado executando o comando `java -version` em seu terminal.

```
Java(TM) SE Runtime Environment
(build 9+181)
Java HotSpot(TM) 64-Bit Server VM
(build 9+181, mixed mode)
```

O `javadoc` da versão está disponível em:

http://download.java.net/java/jdk9/docs/api/overview-summary.html

As principais *IDEs* como *IntelliJ*, *NetBeans* e *Eclipse* já dão suporte ao JDK 9. Caso prefira, você pode fazer diretamente por uma dessas *IDEs* de sua preferência, mas, para fixar a sintaxe, você pode optar por realizar os testes e exemplos do livro com um simples editor de texto.

É hora de programar.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 2

JSHELL: JAVA 9 INTERATIVO

É completamente natural vincular novas versões de uma linguagem com suas principais features. Quando se fala em Java 7, rapidamente pensamos nos *diamond operators*. O mesmo acontece com *lambdas* no Java 8 e agora, no Java 9, com seu sistema de módulos: *Jigsaw*.

Apesar disso, para mim, que trabalho há anos na área de educação e já ensinei Java para centenas de pessoas, a feature mais incrível da nova versão da linguagem sempre será o JShell, que é o protagonista deste capítulo.

O grande motivo do meu entusiasmo é que, passados mais de 20 anos, finalmente foi lançada uma ferramenta completamente voltada para o lado pedagógico e para a fácil adoção da linguagem, que é de longe a mais utilizada e com maior comunidade de usuários do mundo — **são mais de 10 milhões de desenvolvedores!**

Um trecho de sua proposta que eu gosto bastante diz:

"O feedback imediato é importante ao aprender uma nova linguagem ou API. [...] Execução de código interativa é muito mais

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

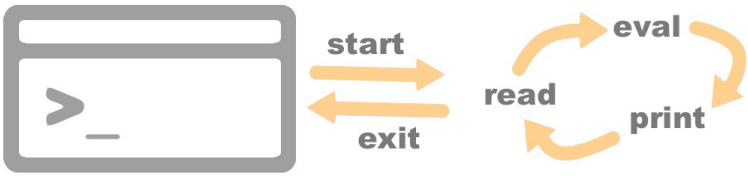
eficiente do que editar, compilar, executar e imprimir valores com `System.out.println` *s. Sem a cerimônia da* `class Teste { public static void main(String[] args) { ... }}` *, o aprendizado e a exploração são simplificados."*

A versão original, em inglês, está disponível no link:

http://openjdk.java.net/jeps/222

2.1 JSHELL, O REPL DO JAVA

Essa nova ferramenta de linha de comando do JDK nada mais é do que um ambiente interativo, extremamente amigável, no qual você pode executar códigos Java e receber feedback imediato. Esse conceito é conhecido como *REPL* (*Read-Eval-Print-Loop*), e existe desde o tempo do LISP, lá pelos anos 60. Ele também é bastante usado em linguagens como Scala, Clojure, Haskell e Ruby.



Você digita instruções e a ferramenta imediatamente lê, valida e retorna o resultado de sua execução. Isso faz com que o primeiro contato de usuários na linguagem seja rápido e até mesmo divertido.

E é importante perceber que o REPL não é voltado apenas para novos usuários. Ele abre espaço para testes rápidos, e é um

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

fortíssimo aliado na exploração de novas sintaxes, recursos e APIs. Eu passei o tempo inteiro com ele aberto, enquanto me atualizava com as novas propostas do Java para escrever este livro.

2.2 INICIANDO O JSHELL

Para iniciar o JShell, basta digitar a palavra `jshell` pelo terminal de seu sistema operacional preferido.

```
turini ~ $ jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell>
```

> Lembre-se que, para que ele funcione, você já precisará ter o Java 9 instalado conforme orientado no final do capítulo anterior. O comando será o mesmo para os diferentes sistemas operacionais, com pequenas particularidades em seu uso que devem aparecer em observações ao decorrer do livro.

Tudo pronto, você já está no ambiente REPL do Java. Ele inclusive já lhe deu a sugestão de um comando que pode e deve ser usado para saber por onde começar, o `/help intro` :

```
jshell> /help intro
|
|  intro
|
|  The jshell tool allows you to execute Java code, getting immed
iate results.
|  You can enter a Java definition (variable, method, class, etc)
, like:  int x = 8
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
|  or a Java expression, like:  x + x
|  or a Java statement or import.
|  These little chunks of Java code are called 'snippets'.
|
|  There are also jshell commands that allow you to understand an
d
|  control what you are doing, like:  /list
|
|  For a list of commands: /help
```

Saindo do loop

Toda vez que eu conheço uma nova ferramenta de linha de comando, lembro-me imediatamente de um artigo que li do Stack Overflow, falando que já ajudou milhões de pessoas a sair do *Vim* — um editor de texto bastante conhecido e talvez um tanto complexo para novos usuários.

https://stackoverflow.blog/2017/05/23/stack-overflow-helping-one-million-developers-exit-vim/

Para que nenhum leitor corra esse risco, vou desde já explicar que o JShell pode ser fechado com o comando:

```
/exit
```

E uma característica que eu gosto muito é que, se você digitar um comando incompleto, ele vai oferecer todas as opções disponíveis:

```
jshell> /e
|  Command: '/e' is ambiguous: /edit, /exit, /env
|  Type /help for help.
```

Além disso, a ferramenta também completará o restante do comando, caso seja a única opção disponível. Por exemplo, digitar `/ex`, ou `/exi`, terá o mesmo resultado que o `/exit`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
jshell> /ex
|  Goodbye
```

2.3 O CLÁSSICO HELLO WORLD COM REPL

No livro *Desbravando a Orientação a Objetos com Java* — também publicado pela Casa do Código —, eu fiz a introdução da linguagem com a classe `MeuPrimeiroPrograma`, que tinha a seguinte estrutura:

```
class MeuPrimeiroPrograma {

  public static void main(String[] args) {
    System.out.println("O primeiro de muitos!");
  }
}
```

A minha frase, logo depois de mostrar essa classe, foi exatamente essa:

> *Esse programa imprime um texto simples. Confuso? Não se preocupe, em breve cada uma dessas palavras terá seu sentido bem claro.*

O motivo disso é que, para eu explicar um simples `System.out`, precisei criar uma classe e declarar o método `main`, que tem modificador de visibilidade, recebe um `array` como parâmetro e mais um monte de palavras que até então não fazem o menor sentido para quem está dando o primeiro passo. Também precisei pedir para salvar o nome do arquivo exatamente como o nome da classe, para não ter de entrar ainda mais em

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

detalhes de convenções e de como funciona o processo de compilação e do *bytecode* gerado.

Com o JShell, podemos simplificar muito esse primeiro contato com a linguagem, sem toda essa burocracia e quantidade enorme de informações para novos usuários processarem. Para atingir o mesmo objetivo, precisaríamos de uma única linha:

```
jshell> System.out.println("O primeiro de muitos!")
O primeiro de muitos!
```

Legal, não é? Perceba inclusive a ausência do ponto e vírgula no final de expressão! Com o tempo, novos usuários vão, sim, precisar entender essa regra na linguagem, mas isso agora pode ser feito um pouco depois. Um passo de cada vez.

2.4 DECLARANDO VARIÁVEIS EXPLICITAMENTE

Você pode declarar variáveis da mesma forma que faria normalmente em seu código.

```
jshell> int age = 28;
age ==> 28
```

O JShell vai exibir o nome da variável seguida de uma seta `==>` e seu valor. Em qualquer momento, nesta mesma seção, você pode digitar o nome da variável e dar `enter`, para que ele retorne o valor atribuído:

```
jshell> age
age ==> 28
```

Claro, você não precisa sempre inicializar uma variável com valor. Neste caso, o JShell exibirá o valor *default*, caso exista, ou

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`null` como acontece no caso da `String`:

```
jshell> int age;
age ==> 0

jshell> String name;
name ==> null
```

Apesar de eu ter escrito, o uso do ponto e vírgula no final é opcional em todos esses casos.

2.5 DECLARANDO VARIÁVEIS DE FORMA IMPLÍCITA

Ao executar uma instrução no JShell, ele valida, executa e, quando existe um retorno, automaticamente atribui esse valor em uma variável. Caso você não tenha declarado essa variável de retorno, ele fará isso de forma implícita.

Quer um exemplo? Basta digitar qualquer valor e pressionar o `enter`:

```
jshell> "Rodrigo Turini";
$5 ==> "Rodrigo Turini"
```

Repare que, no exemplo, ele criou uma variável chamada `$5`, com meu nome. Essa variável implícita sempre começa com um símbolo de dólar seguido pelo ID do *snippet*, que é a forma como o JShell chama as instruções que você executa.

Para ver a lista com todos *snippets* usados, use o comando `/list`:

```
jshell> /list

   2 : age
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
   3 : int age;
   4 : String name;
   5 : "Rodrigo Turini";
```

Outro detalhe importante é que o JShell automaticamente identificou que o tipo da variável é `String`, por ser um texto entre aspas. A prova disso é que eu posso chamar métodos da `String` nessa minha variável `$5`, como por exemplo:

```
jshell> $5.toUpperCase()
$6 ==> "RODRIGO TURINI"
```

O mesmo acontece com o resultado de expressões como a seguir:

```
jshell> 1 + 1
$7 ==> 2
```

Neste caso, o JShell executou a expressão `1 + 1` e atribuiu seu resultado na variável implícita `$7`. Da mesma forma que a String, ele soube inferir que a soma de dois inteiros resultaria em um valor do tipo `int`.

2.6 AJUSTANDO OS NÍVEIS DE FEEDBACK

Como vimos, ao fazer declarações e executar expressões com retorno, por padrão, o JShell sempre fala o nome da variável e seu valor. Se você for um usuário avançado, talvez queira simplesmente desativar esses feedbacks da ferramenta.

Para fazer isso, basta executar:

```
/set feedback silent
```

Ao executar esse comando, você perceberá que até mesmo a palavra *jshell* vai sumir da tela, ficando apenas uma seta indicando

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

as linhas.

Ao criar uma nova variável, a diferença ficará bem clara:

```
-> String example;
->
```

Além do modo `normal`, que é o padrão, e do `silent` que vimos, o JShell possui mais dois níveis de feedback: o `concise` e o `verbose`. O mode `verbose` me parece o mais interessante para quem está começando, por deixar sempre claro qual o tipo das variáveis e seus retornos:

```
-> /set feedback verbose
|  Feedback mode: verbose

jshell> String example;
example ==> null
|  created variable example : String
```

Observe que, nesse caso, com feedback em modo `verbose`, além de apontar o nome da variável criada e seu valor inicial, o JShell também informou que criou a variável e que o seu tipo é `String`.

Caso a variável já exista, ele avisa que está sobrescrevendo:

```
jshell> String example;
example ==> null
|  modified variable example : String
|    update overwrote variable example : String
```

2.7 MÉTODOS, CLASSES E INSTRUÇÕES MAIS COMPLEXAS

Até agora, trabalhamos com tipos simples, declarações de variáveis e *statements* de uma única linha, mas o JShell vai muito

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

além disso. Nele você pode executar códigos de várias linhas, declarar métodos ou até mesmo classes.

Para testar, vamos escrever uma das possíveis soluções para o clássico desafio da sequência de *fibonacci*:

```
jshell> long fibonacci(long n) {
   ...>    if(n<2) return n;
   ...>    return fibonacci(n-1) + fibonacci(n-2);
   ...> }

| created method fibonacci(long)
```

Poderíamos ter escrito esse código todo em uma única linha, mas não existem limitações no uso de `enters` para termos um código mais organizado e legível.

Vamos experimentar? Chame o método *fibonacci*, que acabamos de criar, dentro de um `for` que vai de 0 a 5, imprimindo sua saída:

```
jshell> for(int i = 0; i < 5; i++ ) {
   ...>    System.out.println(fibonacci(i));
   ...> }
0
1
1
2
3
```

Perfeito, chegamos ao resultado esperado!

Editando snippets e criando classes

No lugar de ter escrito o método solto dessa forma, também poderíamos tê-lo declarado dentro de uma classe. Uma feature bastante interessante do JShell é que ele permite editar *snippets* de código. Basta descobrir o seu ID, digitando o comando `/list` ,

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

para a lista completa, ou `/list` e o nome do método que quer editar, como a seguir:

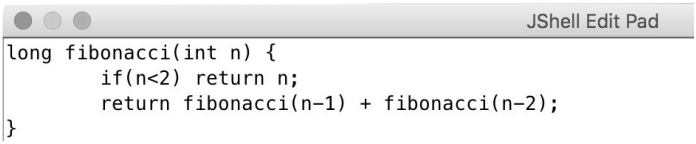
```
jshell> /list fibonacci

   4 : long fibonacci(long n) {
           if(n<2) return n;
           return fibonacci(n-1) + fibonacci(n-2);
       }
```

Agora que sabemos que o ID desse snippet é 4, podemos usar o comando `/edit` :

```
/edit 4
```

Um editor padrão do JShell aparecerá, para que você possa fazer a edição:



> Caso queira, você pode utilizar o comando `/set editor` para substituir o *JShell Edit Pad*, que é a opção padrão, por algum editor de sua preferência. Para usar o *vim*, por exemplo, basta executar a instrução `/set editor vim` . Cuidado para não ficar preso nele, aparentemente milhões de pessoas ficam.

Agora que estamos com o editor aberto, vamos envolver o método em uma classe chamada `Fibonacci` e também adicionar

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

um novo método para facilitar o teste que fizemos anteriormente. Esse método pode receber um parâmetro inteiro, que será o número final que deverá ser calculado.

O código ficará assim:

```
public class Fibonacci {

  private long fibonacci(long n) {
    if(n<2) return n;
    return fibonacci(n-1) + fibonacci(n-2);
  }

  public void loop(int end) {
    for (int i = 0; i < end; i++) {
      System.out.println(fibonacci(i));
    }
  }
}
```

Feita a edição, basta fechar o editor e digitar o comando `/list` para conferir que a nova classe está na lista de *snippets*:

```
jshell> /list

   4 : long fibonacci(long n) {
           if(n<2) return n;
       return fibonacci(n-1) + fibonacci(n-2);
       }
   5 : for(int i = 0; i < 5; i++ ) {
           System.out.println(fibonacci(i));
       }
   6 : public class Fibonacci {

           private long fibonacci(long n) {
              if(n<2) return n;
              return fibonacci(n-1) + fibonacci(n-2);
           }

           public void loop(int end) {
               for (int i = 0; i < end; i++) {
                   System.out.println(fibonacci(i));
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
            }
        }
    }
```

Ela está lá, mas o método anterior e o `for` que fizemos para testá-lo também! Podemos apagá-los com o comando `/drop`.

```
jshell> /drop 5

jshell> /drop 4
|  dropped method fibonacci(long)
```

> Além do comando `/list`, que já vimos, você também pode utilizar `/types`, `/methods` e `/vars` para listar apenas classes, métodos ou variáveis, respectivamente.

Vamos testar essa classe criada? Podemos criar uma nova instância e chamar seu método `loop`:

```
jshell> new Fibonacci().loop(5)
0
1
1
2
3
```

Excelente, novamente recebemos o resultado esperado.

2.8 ADICIONANDO IMPORTS

Até então, só usamos tipos primitivos e a `String`, mas poderíamos ter utilizado qualquer outra classe da API. Para testar, vamos experimentar editar a classe `Fibonacci` novamente, substituindo o *enhanced for* de seu método `loop` por um

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`Stream` do Java 8.

Você pode fazer isso digitando o comando `/edit Fibonacci` e substituindo a implementação do método pela seguinte:

```
public void loop(int end) {

    LongStream
        .range(0, end)
        .map(this::fibonacci)
        .forEach(System.out::println);
}
```

Agora basta executá-lo:

```
jshell> new Fibonacci().loop(5)
|  attempted to use class Fibonacci which cannot be instantiated
or its methods invoked until variable LongStream is declared
```

Opa, o erro diz que a classe não pode ser instanciada até que o `LongStream` seja declarado, e faz sentido. Da mesma forma que em suas classes Java, no JShell precisamos adicionar *imports* de tudo que esteja além dos pacotes *default* da linguagem. Para usar alguma classe da API de *Streams*, precisamos adicionar o `import` como a seguir:

```
import java.util.stream.*;
```

Tudo pronto! Execute o código novamente e você verá que agora tudo volta a funcionar.

Você pode ver a lista completa de `imports` com o comando:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.prefs.*
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
|    import java.util.regex.*
|    import java.util.stream.*
```

> **Atenção**
>
> Para listar os pacotes, usamos `/imports`, com a barra. Para importar de fato um pacote, usamos `import nome.do.pacote`, sem barra.

2.9 ATALHOS, TRUQUES E AUTO-COMPLETION

Uma outra forma ainda mais interessante de adicionar os `imports` necessários, como no exemplo que fizemos agora com o `LongStream`, é usando o atalho: `Shift + Tab i`.

No lugar de ter de digitar o `/import` com o nome do pacote, na próxima vez que for utilizar alguma classe nova, experimente pressionar as teclas `Shift` e `Tab` juntas, soltá-las e em seguida a tecla `i`. O `import` será feito automaticamente para você!

```
jshell> LocalDate
0: Do nothing
1: import: java.time.LocalDate
```

Interessante, não é? Mas ele não funciona com imports estáticos. Ao tentar importar o método `of`, do `Stream`, ele diz que não encontrou candidatos:

```
jshell> of
No candidate fully qualified names found to import.
```

Para esses casos, quando necessário, você pode declarar o

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

import explicitamente como a seguir:

```
jshell> import static java.util.stream.Stream.of;
```

Outro atalho que o ajuda a ficar mais produtivo é o `Shift + Tab v`, para declaração de variáveis com nomes significativos. Para testar, comece a criar uma nova instância de nossa classe `Fibonacci` no JShell como a seguir:

```
jshell> new Fibonacci()
```

E nesse momento, use o atalho `Shift + Tab v`. O tipo `Fibonacci` será escrito para você e o cursor aparecerá no local onde deve ser inserido o nome da variável, logo antes do sinal de igual:

```
jshell> Fibonacci _ = new Fibonacci()
```

Vou deixar aqui uma lista com outros atalhos do teclado que podem ajudá-lo bastante:

Atalho	Descrição
`Ctrl + a`	Move o cursor para o início da linha
`Ctrl + e`	Move o cursor para o final da linha
`Alt + b`	Volta uma palavra
`Alt + f`	Avança uma palavra
`Ctrl + r`	Busca por comandos ou instruções digitados

2.10 EXPLORANDO NOVAS APIS

Outro recurso que uso muito é o de *auto-completion*, que no caso do JShell é executado ao pressionar a tecla `Tab`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Comece a digitar a palavra *Stream*, com apenas as três primeiras letras, e pressione o `Tab`. O resultado será uma lista com todas as possibilidades de classes que começam dessa forma:

```
jshell> Str
StreamCorruptedException          StreamTokenizer
StrictMath                        String
StringBuffer                      StringBufferInputStream
StringBuilder                     StringIndexOutOfBoundsException
StringJoiner                      StringReader
StringTokenizer                   StringWriter
```

O comportamento é muito parecido com o das principais IDEs.

Ao terminar de digitar a palavra *Stream*, coloque o ponto final e pressione `Tab` novamente:

```
jshell> Stream.
Builder       builder()     class         concat(       empty()
generate(     iterate(      of(           ofNullable(
```

A lista com todos os métodos disponíveis aparecerá para você. Há ainda a opção de ver o *javadoc* de um determinado método, digitando seu nome até o sinal de abertura dos parênteses, e pressionando `Tab` novamente:

```
jshell> Stream.of(
Signatures:
Stream<T> Stream<T>.<T>of(T t)
Stream<T> Stream<T>.<T>of(T... values)

<press tab again to see documentation>
```

Ao fazer isso, o JShell nos mostra a lista de assinaturas do método, e dá a opção de pressionarmos o `Tab` novamente para listar mais detalhes da documentação:

```
jshell> Stream.of(
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
Stream<T> Stream<T>.<T>of(T t)
Returns a sequential Stream containing a single element.

Type Parameters:
T - the type of stream elements

Parameters:
t - the single element

Returns:
a singleton sequential stream

<press tab to see next documentation>
```

Essa facilidade torna o JShell um aliado ainda mais forte para a exploração de novas APIs. Desde que comecei a estudar o Java 9, não parei de usar! Mesmo em suas primeiras versões beta.

2.11 MAIS DETALHES E CONFIGURAÇÕES

Além dos exemplos vistos aqui, existem diversos outros que podem ser utilizados no decorrer do livro. Você pode, e eu recomendo bastante, conhecer e experimentá-los desde já, passeando pela lista de comandos presente no atalho `/help` .

São muitas as possibilidades! Algumas que não podem ficar de fora, e que valem uma menção especial, são:

Comando	Descrição
`/reload`	Para limpar sua seção do JShell
`/save nome-do-arquivo`	Para salvar todos os snippets da seção atual
`/open nome-do-arquivo`	Para abrir o jShell com todos os snippets salvos
`/open`	Para importar a declaração dessa classe existente

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`AlgumaClasse.java`	
`/history`	Para mostrar seu histórico de comandos executados na seção

2.12 DOWNLOAD DOS EXEMPLOS DESTE CAPÍTULO

Todos os exemplos que executei durante este capítulo foram salvos com o comando `/save` e estão disponíveis em:

https://github.com/Turini/livro-java-9

Para abri-los em seu terminal, basta digitar `/open exemplos-capitulo-jshell` na seção ativa do JShell, ou começar uma nova a partir do arquivo, digitando:

```
jshell exemplos-capitulo-jshell
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 3

ATUALIZAÇÕES NAS PRINCIPAIS APIS

Diversas APIs receberam melhorias na nova versão da linguagem. Neste capítulo, vamos ver as principais delas, com destaque especial para as Collections, Streams e Optionals.

Você pode e deve executar todos os exemplos de código que aparecerem aqui, para se adaptar à sintaxe e testar as possibilidades dos novos métodos. Para fazer isso, já é possível utilizar uma das IDEs que dão suporte ao Java 9, como NetBeans, Intellij IDEA e Eclipse.

Outra alternativa que eu recomendo bastante é que você utilize o ambiente REPL, o JShell, para testar e explorar todas essas novidades.

Acompanhe também pela documentação

Outro forte aliado no estudo e exploração de APIs é sua documentação. No Java 9, o *javadoc* agora tem suporte para HTML5, e também uma barra de busca, que facilita muito esse trabalho.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Java SE 9 & JDK 9

SEARCH: AIOOBE

Types

java.lang.**ArrayIndexOutOfBoundsE**xception

Members

java.lang.**ArrayIndexOutOfBoundsE**xception.ArrayIndexOutOfBoundsException()
java.lang.**ArrayIndexOutOfBoundsE**xception.ArrayIndexOutOfBoundsException(int)
java.lang.**ArrayIndexOutOfBoundsE**xception.ArrayIndexOutOfBoundsException(String)

A documentação dessa versão da linguagem está disponível em:

http://download.java.net/java/jdk9/docs/api/overview-summary.html

3.1 MELHORIAS NAS COLLECTIONS

A API de Collections está presente desde o JDK 1.2 e, desde lá, evoluiu muito. Em especial com a chegada dos `generics` no Java 5, operadores diamante no Java 7 e a possibilidade de transformação em Streams no Java 8.

Agora conseguimos fazer filtros, projeções e operações comuns com uma abordagem bem mais próxima à programação funcional. No Java 9, esse ciclo de evolução não parou.

Uma nova forma de criar mapas imutáveis

Para entender a motivação de forma prática, considere o seguinte mapa com os dias da semana:

```
Map<Integer, String> weekDays = new HashMap<>();
weekDays.put(1, "Domingo");
weekDays.put(2, "Segunda");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
weekDays.put(3, "Terça");
weekDays.put(4, "Quarta");
weekDays.put(5, "Quinta");
weekDays.put(6, "Sexta");
weekDays.put(7, "Sábado");

weekDays = Collections.unmodifiableMap(weekDays);
```

Para prepará-lo, foi preciso criar uma nova instância do `HashMap`, sua implementação mais comum. Logo em seguida, chamamos o método `put` inserindo cada uma de suas entradas e, por fim, usamos o `Collections#unmodifiableMap` para impedir que ele seja modificado. Nada muito complicado, mas foram várias linhas de código e passos para alcançar o objetivo simples de criar um conjunto de chaves e valores imutáveis.

A própria linguagem tem um antigo recurso conhecido como *double brace initialization*, que nos permite declarar esse mesmo mapa da seguinte forma:

```
weekDays = Collections.unmodifiableMap(new HashMap<>() {{
    put(1, "Domingo"), put(2, "Segunda"); [...]
}});
```

Essa é uma das funcionalidades escondidas do Java, e está assim por um bom motivo! Além de não ser nada recomendada, por envolver criação de várias classes anônimas, o código continua bastante verboso e agora com uma sintaxe esdrúxula. **Nem sempre menos linhas de código implicam em um código melhor, ou mais legível**.

As bibliotecas ajudam bastante

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Algumas bibliotecas oferecerem diferentes alternativas para simplificar a manipulação e complementar as funcionalidades do `Map` e das diferentes Collections. O *Guava*, do Google, é um deles.

Com ele, podemos escrever o mesmo código assim:

```
import com.google.common.collect.ImmutableMap;

Map<Integer, String> weekDays = ImmutableMap.of(
    1, "Domingo",
    2, "Segunda",
    3, "Terça",
    4, "Quarta",
    5, "Quinta",
    6, "Sexta",
    7, "Sábado"
);
```

Interessante, não é? E a sintaxe fica bem mais próxima do modelo de *collection literals*, bastante conhecido em linguagens funcionais. Em Ruby, por exemplo, um `Hash`, que é sua implementação equivalente, seria escrito assim:

```
# código Ruby
weekDays = {
    1 => "Domingo",
    2 => "Segunda",
    3 => "Terça",
    4 => "Quarta",
    5 => "Quinta",
    6 => "Sexta",
    7 => "Sábado"
}
```

Collection literals

O Brian Goetz, arquiteto da Oracle, vem falando sobre a possibilidade das *collection literals* no Java desde muito antes do Java 8. Ele inclusive já esboçou uma JEP de pesquisa com algumas

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

das possibilidades:

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

Apesar de a proposta ainda não ter entrado no Java 9, uma outra JEP intitulada *Convenience Factory Methods for Collections* traz uma série de métodos permitindo a criação de classes não modificáveis com uma sintaxe simplificada, como fazem as bibliotecas.

Convenience Factory Methods

No Java 9, com o novo método `of`, podemos criar o mesmo mapa assim:

```
Map<Integer, String> weekDays = Map.of(
    1, "Domingo",
    2, "Segunda",
    3, "Terça",
    4, "Quarta",
    5, "Quinta",
    6, "Sexta",
    7, "Sábado"
);
```

E ainda existem algumas variações de sintaxe que podem servir para quando você tem mapas um pouco maiores, que precisam ser construídos dinamicamente.

```
Map.ofEntries(
    Map.entry(1, "Domingo"),
    Map.entry(2, "Segunda"),
    Map.entry(3, "Terça"),
    // ...
);
```

Extensões nas Lists e Sets

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Da mesma forma que o `Map`, as interfaces `Set` e `List` também receberam implementações do método `of`, permitindo a construção de suas coleções como essa abordagem simplificada:

```
List<String> names = List.of("Rodrigo", "Vivian", "Alexandre");

Set<String> colors = Set.of("azul", "vermelho", "amarelo");
```

Vale reforçar que, em todos esses casos, as coleções criadas a partir desses métodos são imutáveis. Você não pode inserir novos valores depois que ela estiver instanciada. Vamos testar?

Experimente adicionar um novo valor em alguma delas.

```
List<String> names = List.of("Rodrigo", "Vivian", "Alexandre");
names.add("Paulo");
```

O resultado, já esperado, será uma `UnsupportedOperationException`.

```
java.lang.UnsupportedOperationException thrown:
    at ImmutableCollections.uoe
        (ImmutableCollections.java:70)
    at ImmutableCollections$AbstractImmutableList.add
        (ImmutableCollections.java:76)
    at (#2:1)
```

Alguns detalhes e erros comuns

Outro detalhe interessante é que não há garantia da implementação que será devolvida quando utilizamos algum desses métodos. Ao usar o `Set#of`, por exemplo, não temos garantia de que será criada uma instância do tipo `HashSet`, `TreeSet` ou qualquer outro.

Algo importante de ser notado é que o método estático `of` é chamado diretamente a partir das interfaces `List`, `Set` e `Map`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Como isso é possível?

Desde o Java 8 interfaces passaram a ter suporte a métodos estáticos! Mas uma característica importante é que eles **não são herdados**, portanto você não pode utilizá-lo a partir de alguma das implementações específicas dessas interfaces.

Isso significa que o código a seguir não compila:

```
ArrayList.of("a", "b", "c"); //erro de compilação
```

Você também não pode passar valores nulos para esses métodos, pois isso resultará em uma `NullPointerException`:

```
jshell> List.of("não", "pode", "ter", null);

  java.lang.NullPointerException thrown:
|        at List.of (List.java:1030)
```

Além disso, inserir chaves duplicadas em mapas e conjuntos causará um `IllegalArgumentException`.

```
jshell> Map.of(1, "a", 1, "b");

|  java.lang.IllegalArgumentException thrown: duplicate key: 1
|        at ImmutableCollections$MapN.<init> (ImmutableCollection
s.java:680)
|        at Map.of (Map.java:1326)
|        at (#5:1)
```

Entendendo o motivo das várias sobrecargas

Abra o JShell, digite `List.of`, e pressione `Tab`. Uma lista de possíveis implementações deverá aparecer e, talvez para sua surpresa, você verá que o método `of` possui diversas assinaturas, partindo de 0 até 10 argumentos e, por fim, uma nova sobrecarga que recebe um *var-args*.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
jshell> List.of(

Signatures:
List<E> List<E>.<E>of()
List<E> List<E>.<E>of(E e1)
List<E> List<E>.<E>of(E e1, E e2)
List<E> List<E>.<E>of(E e1, E e2, E e3)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E
 e8)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E
 e8, E e9)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E
 e8, E e9, E e10)
List<E> List<E>.<E>of(E... elements)
```

Essa é uma estratégia de performance bastante comum para evitar a alocação, inicialização e processo de *garbage collection* de arrays. E o motivo para ir de 0 a 10 é que pesquisas apontaram que a esmagadora maioria dos casos de uso de coleções imutáveis varia entre essa quantidade de argumentos.

É claro que você não precisa fazer isso sempre que for usar um var-args em seu código do dia a dia, a não ser que trabalhe em um contexto crítico de performance.

Há uma frase do Donald Knuth, um professor da universidade de Stanford, que ficou muito conhecida no mundo da programação:

"Otimização prematura é a raiz de todo mal".

Em resumo você precisa tomar cuidado com isso. No geral, sair fazendo micro-otimizações em seu código do dia a dia não é uma boa ideia, pois muitas vezes acaba adicionando uma complexidade

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

desnecessária.

Mas esse certamente é um cuidado necessário para o *core* de uma linguagem, e também para APIs e frameworks de manipulação de dados.

3.2 ATUALIZAÇÕES NOS STREAMS

A API de *Streams* também tem novidades. Uma delas é a introdução do método `ofNullable` que, diferente do tradicional método `of`, trata a possível `NullPointerException` caso o valor passado seja `null`.

Isso pode ser bastante conveniente para evitar os verbosos *null-checks* no meio de seu código, em especial em operações de transformação de valores. Quer um exemplo? Considere o mesmo mapa dos dias da semana:

```
Map<Integer, String> weekDays = new HashMap<>();
weekDays.put(1, "Domingo");
weekDays.put(2, "Segunda");
weekDays.put(3, "Terça");
weekDays.put(4, "Quarta");
weekDays.put(5, "Quinta");
weekDays.put(6, "Sexta");
weekDays.put(7, "Sábado");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Por que não usamos o `Map.of()` ?

Talvez você tenha ficado curioso sobre por que não utilizei o método `of()` no código anterior, e a resposta é simples: para facilitar os nossos testes. Logo vamos adicionar novos elementos nesse mapa e, como mencionado anteriormente, o método `of` cria objetos imutáveis.

Experimente transformá-lo em uma lista como a seguir, com apenas as iniciais de seus valores, ou seja, dos nomes dos dias da semana.

```
[Dom, Seg, Ter, Qua, Qui, Sex, Sáb]
```

Caso você esteja acostumado com os *Streams*, poderia resolver facilmente com uma operação de `flatMap` no conjunto de valores desse mapa, para transformá-lo em um `Stream<String>` , com o texto dos dias da semana. Logo em seguida, poderia fazer uma projeção com o método `map` mantendo apenas as primeiras 3 letras dessas *Strings*.

O código ficaria assim:

```
List<String> dayNames = weekDays
    .entrySet()
    .stream()
    .flatMap(e -> Stream.of(e.getValue()))
    .map(s -> s.substring(0,3))
    .collect(Collectors.toList());

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

Ao executá-lo, o resultado seria exatamente o esperado:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
[Dom, Seg, Ter, Qua, Qui, Sex, Sáb]
```

> Para que o código anterior funcione no JShell, é preciso importar a classe `Collectors`.
>
> ```
> jshell> import java.util.stream.Collectors;
> ```
>
> Lembre-se que, assim como na classe com método *main* ou em seus códigos do dia a dia, no JShell você precisa importar qualquer recurso que não esteja nos pacotes padrões da linguagem, como `java.lang` e `java.util`.

Parece tudo ok e o código funciona. Porém, o que aconteceria se algum desses valores fosse `null`? Podemos adicionar uma nova entrada no mapa para testar:

```
weekDays.put(8, null);
```

E agora, ao executar o código novamente, teríamos:

```
java.lang.NullPointerException thrown:
    at lambda$do_it$$1 (#95:1)
    at ReferencePipeline$3$1.accept (ReferencePipeline.java:195)
    at Streams$StreamBuilderImpl.forEachRemaining (Streams.java:4
11)
    //...
```

Dessa vez, o resultado foi uma `NullPointerException`.

Isso acontece porque o `flatMap` manteve a referência nula no *pipeline* do Stream. Para entender melhor, vendo isso acontecer, retire a linha que faz o `map` com `substring` do nosso Stream e execute o código novamente.

```
List<String> dayNames = weekDays
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    .entrySet()
    .stream()
    .flatMap(e -> Stream.of(e.getValue()))
    .collect(Collectors.toList());

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

A saída será:

```
[Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado, null]
```

Perceba que o `null` está lá! Como resolver? É aí que entra o novo método, o `ofNullable`. Experimente usá-lo no `flatMap`, dessa mesma forma:

```
List<String> dayNames = weekDays
    .entrySet()
    .stream()
    .flatMap(e -> Stream.ofNullable(e.getValue()))
    .collect(Collectors.toList());

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

Execute novamente e o resultado, conforme esperado, será a lista de dias da semana sem o `null`.

```
[Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado]
```

Agora sim, podemos voltar o método `map` de antes, pois seu código estará protegido do possível `NullPointerException`:

```
List<String> dayNames = weekDays
    .entrySet()
    .stream()
    .flatMap(e -> Stream.ofNullable(e.getValue()))
    .map(s -> s.substring(0,3))
    .collect(Collectors.toList());
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Alternativas anteriores

Antes do método `ofNullable`, poderíamos ter resolvido esse problema adicionando uma condição ternária no `flatMap`, retornando um `Stream#empty` caso o valor fosse `null`, ou ainda adicionando um `filter` a mais no Stream, tratando essa possibilidade:

```
Set<String> dayNames = weekDays
    .entrySet()
    .stream()
    .flatMap(e -> Stream.of(e.getValue()))
    .filter(s -> s != null)
    .map(s -> s.substring(0,3))
    .collect(Collectors.toList());
```

Isso resolveria o problema, mas teríamos o custo de adicionar etapas extras e mais verbosidade ao código.

Mais controle com dropWhile e takeWhile

A API de Stream já possui dois métodos que nos ajudam a pular os primeiros valores, ou adicionar um limite em uma coleção. Estes são os métodos `skip` e `limit`.

Com o `skip`, por exemplo, eu posso escrever esse trecho de código para pular os 5 primeiros valores desse Stream de inteiros e retornar apenas o restante:

```
IntStream.range(0,10)
    .skip(5)
    .forEach(System.out::println);
```

Nesse caso, a saída será:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
5
6
7
8
9
```

Já o `limit`, seu outro método, nos permite adicionar um limite máximo a essa sequência de inteiros, como no exemplo:

```
IntStream.range(0,10)
    .limit(3)
    .forEach(System.out::println);
```

Como definimos um limite de 3 elementos, a saída será apenas os inteiros:

```
0
1
2
```

Agora, com Java 9, a API recebeu mais dois métodos, chamados `dropWhile` e `takeWhile`, que permitem refinar esse controle dos métodos `skip` e `limit`. Eles aceitam um `Predicate` — que é uma condição lógica — como parâmetro em vez de apenas um número de elementos, como fazem as outras duas implementações.

O método `dropWhile` remove os elementos enquanto sua condição lógica for verdadeira. Na mesma lista de inteiros, por exemplo, podemos remover todos os números que sejam menor ou igual a 5 com o `dropWhile(e -> e <= 5)`:

```
IntStream.range(0,10)
    .dropWhile(e -> e <= 5)
    .forEach(System.out::println);
```

O código vai imprimir:

```
6
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
7
8
9
```

Nota

É importante perceber que, assim que algum dos elementos retornar `false`, o `dropWhile` vai parar de remover. Seu comportamento é não determinístico. Isto é, caso a sequência de inteiros não estivesse ordenada, como a seguir, o resultado seria um pouco diferente do esperado:

```
IntStream.of(3,5,6,7,3,2,1)
    .dropWhile(e -> e <= 5)
    .forEach(System.out::println);
```

O resultado seria:

```
6
7
3
2
1
```

Mesmo com os números finais 3, 2 e 1 sendo menores que 5, eles continuam aparecendo, pois o `dropWhile` aplicou a regra até chegar à primeira falha, que foi o número 6.

Seu outro método, chamado `takeWhile`, funciona de forma muito parecida, mas com o sentido contrário. No lugar de remover, ele vai manter os números enquanto a condição for verdadeira:

```
IntStream.range(0,10)
    .takeWhile(e -> e <= 5)
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

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

Para esse exemplo, o resultado será:

```
0
1
2
3
4
5
```

É muito simples e extremamente comum assimilar o uso desses métodos com Streams numéricos, mas eles também podem ser usados em outras situações. O código a seguir remove todas as Strings que antecedem a palavra `"no"` :

```
Stream.of("muitas", "novidades", "no", "java")
    .dropWhile(s -> !s.equals("no"))
    .forEach(System.out::println);
```

Ao executá-lo, será impresso o texto:

```
no
java
```

E, para o caso inverso, do `takeWhile` :

```
Stream.of("muitas", "novidades", "no", "java")
    .takeWhile(s -> !s.equals("no"))
    .forEach(System.out::println);
```

Ele mantém todas as strings até chegar na palavra `"no"` :

```
muitas
novidades
```

Criando loops com iterate

Outra adição interessante na API de Streams foi o método `iterate` , que oferece uma alternativa para construção de streams

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

sequenciais — ou até mesmo infinitos. Ele funciona de forma muito parecida com o `for` tradicional, que usa um *index*. Poderíamos usar o código a seguir para imprimir todos os números pares, de 0 a 100:

```
for (int i = 0; i <= 10; i = i + 1) {
    System.out.println(i);
}
```

E, com o `iterate`, faríamos o mesmo, assim:

```
Stream
    .iterate(1, n -> n <= 10, n -> n + 1)
    .forEach(System.out::println);
```

Ele recebe três parâmetros:

- Um *seed* inicial;
- Uma condição do tipo `Predicate`, definindo quando ele deve parar;
- Uma operação do tipo `UnaryOperator`, que é executada a cada iteração.

Há ainda uma sobrecarga que recebe apenas o seed inicial e a operação que deve ser executada, resultando em um Stream infinito. Para esse caso, você pode usar o método `limit`, por exemplo, para definir quantos elementos devem ser criados:

```
Stream
    .iterate(1, n -> n + 1)
    .limit(10)
    .forEach(System.out::println);
```

3.3 NOVOS COLLECTORS

A API também ganhou novas formas de agrupar e coletar

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

dados. Para testá-las, vamos utilizar a mesma classe `Book`, que vimos na introdução do livro, com uma pequena alteração. Todo livro agora tem uma lista de categorias às quais ele pertence. `PROGRAMMING`, `DESIGN` e `AGILE` são algumas dessas categorias, representadas pelo `enum`:

```
public enum Category {

   PROGRAMMING,
   DESIGN,
   AGILE,
   CERTIFICATION,
   BUSINESS
}
```

A classe `Book`, com adição da lista de categorias, ficará assim:

```
import java.util.*;

public class Book {

  private final String name;
  private final String author;
  private final List<Category> categories;

  // outros atributos

  public Book(String name, String author, Category ...categories)
{
    this.name = name;
    this.author = author;
    this.categories = List.of(categories);
  }

  public String getAuthor() {
    return author;
  }

  public String getName() {
      return name;
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  public List<Category> getCategories() {
      return categories;
  }

  @Override
  public String toString() {
    return "\nlivro: " + name
        +  "\nautor: " + author
        +  "\ncategorias: " + categories;
  }
}
```

Vamos também criar uma lista de livros, com suas categorias. Para organizar o código e facilitar nossos testes futuros, faremos isso em uma classe chamada `Books`, que encapsula o acesso aos dados dessa lista. Por enquanto, ela terá apenas o método `all`, que retorna todos os livros.

```
public class Books {

  public static List<Book> all() {

    return List.of(
        new Book(
            "Desbravando Java",
            "Rodrigo Turini",
            Category.PROGRAMMING
        ),
        new Book(
            "APIs Java",
            "Rodrigo Turini",
            Category.PROGRAMMING
        ),
        new Book(
            "Certificação Java",
            "Guilherme Silveira",
            Category.PROGRAMMING, Category.CERTIFICATION
        ),
        new Book(
            "TDD",
            "Mauricio Aniche",
            Category.PROGRAMMING, Category.AGILE
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
        ),
        new Book(
            "SOLID",
            "Mauricio Aniche",
            Category.PROGRAMMING
        ),
        new Book(
            "Guia da Startup",
            "Joaquim Torres",
            Category.BUSINESS
        )
    );
  }
}
```

Tudo pronto! Para imprimir o nome de todos os livros, por exemplo, podemos fazer:

```
Books.all().forEach(System.out::println);
```

Entendendo o Collectors.filtering

Queremos agora agrupar esses livros por autor, mas filtrando apenas os que pertencem à categoria de programação. Isso pode ser feito com um `filter` simples, com que já estamos acostumados, e com um `groupingBy` :

```
Books.all().stream()
    .filter(b -> b.getCategories().contains(Category.PROGRAMMING)
)
    .collect(Collectors.groupingBy(Book::getAuthor));
```

O resultado será:

```
{
    Guilherme Silveira=[
        livro: Certificação Java
        autor: Guilherme Silveira
        categorias: [PROGRAMMING, CERTIFICATION]
    ],
    Rodrigo Turini=[
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
        livro: Desbravando Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING],
        livro: APIs Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING]
    ],
    Mauricio Aniche=[
        livro: TDD
        autor: Mauricio Aniche
        categorias: [PROGRAMMING, AGILE],
        livro: SOLID
        autor: Mauricio Aniche
        categorias: [PROGRAMMING]
    ]
}
```

Isso resolve o problema. Entretanto, perceba que o autor Joaquim Torres, do *Guia da Startup*, não aparece em nosso mapa.

Essa é uma característica prevista pelo filter, claro. Mas em algum momento podemos querer continuar vendo todos os autores existentes, mantendo apenas a sua lista de livros vazia, caso ele não tenha nenhum que atenda à condição filtrada.

Essa foi a grande motivação de um dos novos Collectors da API, o `filtering` :

```
Books.all().stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.filtering(
            b -> b.getCategories().contains(Category.PROGRAMMING)
,
        Collectors.toList())));
```

A grande diferença é que agora o filtro é feito dentro do Collector, que agrupa os livros. O resultado será:

```
{
    Guilherme Silveira=[
        livro: Certificação Java
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
        autor: Guilherme Silveira
        categorias: [PROGRAMMING, CERTIFICATION]
    ],
    Joaquim Torres=[
    ],
    Rodrigo Turini=[
        livro: Desbravando Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING],
        livro: APIs Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING]
    ],
    Mauricio Aniche=[
        livro: TDD
        autor: Mauricio Aniche
        categorias: [PROGRAMMING, AGILE],
        livro: SOLID
        autor: Mauricio Aniche
        categorias: [PROGRAMMING]
    ]
}
```

Esse tipo de filtro é usado sempre em conjunto com o `groupingBy`, ou o `partitioningBy`.

Collectors.flatMapping

Além de agrupar os livros por seus autores, também vamos precisar de um relatório mostrando apenas as categorias que cada autor já escreveu. O código ficará muito parecido com o anterior, mas no lugar de filtrar, queremos usar um **mapping** para projetar o resultado dessa forma.

```
Books.all().stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.mapping(Book::getCategories,
        Collectors.toList())));
```

Parece tudo ok, mas ao executá-lo encontramos dois

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

problemas. O primeiro deles é que as categorias estão repetindo para os autores que escreveram mais de um livro na mesma categoria:

```
{
    Guilherme Silveira=[
        [PROGRAMMING, CERTIFICATION]
    ],
    Joaquim Torres=[
        [BUSINESS]
    ],
    Rodrigo Turini=[
        [PROGRAMMING], [PROGRAMMING]
    ],
    Mauricio Aniche=[
        [PROGRAMMING, AGILE], [PROGRAMMING]
    ]
}
```

Conseguimos resolver esse de uma forma bem simples! No lugar de uma lista, podemos usar um `Set`, que não permite a duplicação dos valores.

O código ficará assim:

```
Books.all().stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.mapping(Book::getCategories,
        Collectors.toSet())));
```

E o resultado será:

```
{
    Guilherme Silveira=[
        [PROGRAMMING, CERTIFICATION]
    ],
    Joaquim Torres=[
        [BUSINESS]
    ],
    Rodrigo Turini=[
        [PROGRAMMING]
    ],
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    Mauricio Aniche=[
        [PROGRAMMING, AGILE], [PROGRAMMING]
    ]
}
```

Agora resta um único problema, que talvez você não tenha percebido pela ausência de uma variável declarando o tipo do nosso mapa. Experimente guardar esse nosso código em uma variável, e você verá que o resultado será um mapa de `String` para um `Set` de listas!

```
Map<String,Set<List<Category>>> map =
Books.all().stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.mapping(Book::getCategories,
        Collectors.toSet())));
```

Agora repare também na saída do autor Mauricio Aniche, que tem livros de diferentes categorias.

```
Mauricio Aniche=[[PROGRAMMING, AGILE], [PROGRAMMING]]
```

A categoria programação se repete, inclusive, já que uma delas está em uma lista diferente das demais.

Para resolver esse problema, o `Collector` recebeu um novo método, chamado `flatMapping`. Ele funciona de forma análoga ao `flatMap` do Stream, que permite achatar os elementos de várias listas dentro de uma única coleção.

```
Books.all().stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.flatMapping(
            b -> b.getCategories().stream(),
        Collectors.toSet())));
```

Agora sim, temos um `Set<Category>` no valor do mapa, e o resultado será o esperado:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
{
    Guilherme Silveira=[
        PROGRAMMING, CERTIFICATION
    ],
    Joaquim Torres=[
        BUSINESS
    ],
    Rodrigo Turini=[
        PROGRAMMING
    ],
    Mauricio Aniche=[
        AGILE, PROGRAMMING
    ]
}
```

3.4 ATUALIZAÇÕES NO OPTIONAL

Desde que o `Optional` apareceu no Java 8, ficou fácil expressar em nossos métodos e APIs quando um valor ou retorno pode não existir. Nossa classe `Books`, por exemplo, poderia ter um método que retorna um livro de determinada categoria. Esse método pode ser usado em um algoritmo bem simples de sugestões, por exemplo, para quando um leitor escolher um livro, sugerirmos outro dessa mesma categoria.

Por enquanto, o critério é simples: dada uma categoria, quero qualquer livro que pertença a ela. A implementação poderia ser assim:

```
Books.all().stream()
    .filter(b -> b.getCategories().contains(category))
    .findAny();
```

O `filter` limita os livros que pertencem a essa categoria, e o `findAny` retornará qualquer um deles. Mas também precisamos levar em consideração que não queremos indicar o mesmo livro que o leitor acabou de escolher.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Para simplificar a regra, podemos então definir que a sugestão sempre será para um livro similar, das mesmas categorias, mas apenas se for de um autor diferente.

```
Books.all().stream()
    .filter(b -> b.getCategories().equals(book.getCategories()))
    .filter(b -> !b.getAuthor().equals(author))
    .findAny();
```

Perfeito, agora parece tudo ok. Vamos criar um novo método na classe `Books`, com essa implementação. Ele se chamará `findSimilar`, e receberá um livro como parâmetro. A partir desse livro, vamos conseguir saber as categorias e o autor para aplicar no filtro:

O código completo, na classe `Books`, ficará assim:

```
public class Books {

  public static List<Book> all() {

    return List.of(
        // todos os livros
    );
  }

  public static Optional<Book> findSimilar(Book book) {
      return Books.all().stream()
        .filter(b -> b.getCategories().equals(book.getCategories(
)))
        .filter(b -> !b.getAuthor().equals(book.getAuthor()))
        .findAny();
  }
}
```

Perceba o uso do `Optional` no retorno desse método. Nem sempre vão existir livros similares, portanto isso pode e deve ficar claro em sua assinatura.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Para testar, vamos declarar um novo livro de programação:

```
Book book = new Book(
    "Desbravando Java",
    "Rodrigo Turini",
    Category.PROGRAMMING
);
```

E agora passar esse livro como argumento para o método `findSimilar`:

```
Books.findSimilar(book);
```

O retorno será:

```
Optional[
    livro: SOLID
    autor: Mauricio Aniche
    categorias: [PROGRAMMING]
]
```

Perfeito! Ele indicou um outro livro da mesma categoria e de outro autor. Tudo conforme esperado.

Agora podemos fazer um novo teste, passando um livro que não tem similares:

```
Book book = new Book(
    "Guia da Startup",
    "Joaquim Torres",
    Category.BUSINESS
);

Books.findSimilar(book);
```

Como não existem similares, o código retorna um Optional vazio.

ifPresentOrElse

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Queremos agora imprimir a sugestão caso o livro exista, ou mostrar uma mensagem dizendo que não existem sugestões de livros similares. Uma implementação muito comum seria essa:

```
Optional<Book> similar = Books.findSimilar(book);

if (similar.isPresent()) {
    System.out.println(similar);
} else {
    System.out.println("Não existem similares");
}
```

Esse código resolve o problema, mas veja como ele se parece com os imperativos `ifs` que comparam se um valor é diferente de `null`. O `Optional` tem uma solução um pouco mais interessante e declarativa, que é o método `ifPresent`:

```
Books.findSimilar(book)
    .ifPresent(System.out::println);
```

O problema é que não existe um `else`, forçando-nos a utilizar a abordagem anterior sempre que queremos executar algum comportamento diferente nos casos em que o valor não existe. Agora podemos fazer isso com o novo método `ifPresentOrElse`, que permite passar uma expressão, do tipo `Runnable`, como fallback:

```
Books.findSimilar(book)
    .ifPresentOrElse(
        System.out::println,
        () -> System.out.println("Não existem similares")
    );
```

O resultado será exatamente igual ao do `if` com `isPresent`.

Optional.stream

Uma das introduções que eu mais gostei ao método

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`Optional` foi a possibilidade de projetá-lo em um `Stream`. Quer um exemplo prático? Basta ampliar a proporção do nosso método `findSimilar`.

Dada uma lista de livros, queremos retornar os possíveis conteúdos similares.

```
List<Book> books = List.of(
    new Book(
        "Desbravando Java",
        "Rodrigo Turini",
        Category.PROGRAMMING
    ),
    new Book(
        "Java 8 Prático",
        "Paulo Silveira",
        Category.PROGRAMMING
    ),
    new Book(
        "SOLID",
        "Mauricio Aniche",
        Category.PROGRAMMING
    ),
    new Book(
        "Guia da Startup",
        "Joaquim Torres",
        Category.BUSINESS
    )
);
```

Podemos fazer isso em um `stream`, que procura seu similar para cada elemento e, ao final, coleta o resultado para uma nova lista:

```
List<Optional<Book>> similars =
books.stream()
    .map(Books::findSimilar)
    .collect(Collectors.toList());
```

A variável `similars` conterá:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
[
    Optional[
        livro: SOLID
        autor: Mauricio Aniche
        categorias: [PROGRAMMING]
    ],
    Optional[
        livro: Desbravando Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING]
    ],
    Optional[
        livro: Desbravando Java
        autor: Rodrigo Turini
        categorias: [PROGRAMMING]
    ],
    Optional.empty
]
```

Esse código funciona, mas ao final ficamos com uma lista de Optionals, incluindo os vazios para casos em que não existem similares. Queremos uma lista com apenas os valores válidos, logo, poderíamos filtrar os opcionais vazios:

```
List<Book> similars =
books.stream()
    .map(Books::findSimilar)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());
```

Teremos na variável `similars`:

```
[
    livro: SOLID
    autor: Mauricio Aniche
    categorias: [PROGRAMMING],

    livro: Desbravando Java
    autor: Rodrigo Turini
    categorias: [PROGRAMMING],

    livro: Desbravando Java
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    autor: Rodrigo Turini
    categorias: [PROGRAMMING]
]
```

Chegamos ao resultado, mas ao custo de duas operações extras no pipeline do Stream.

Com o novo método `Optional#stream`, agora podemos substituir esse `filter` e `map` por um `flatMap`, como a seguir:

```
List<Book> similars =
books.stream()
    .map(Books::findSimilar)
    .flatMap(Optional::stream)
    .collect(Collectors.toList());
```

Ou ainda, caso prefira, fazer a operação de *flatten* de uma única vez:

```
List<Book> similars =
books.stream()
    .flatMap(b -> Books.findSimilar(b).stream())
    .collect(Collectors.toList());
```

As duas formas vão trazer o mesmo resultado. Existem casos em que usar o **method reference** traz melhores resultados, seja por detalhes internos do bytecode ou ainda por ele possuir a tipagem mais forte - já que a classe fica explicitamente definida na expressão - mas, nesse exemplo, a legibilidade seria o maior diferencial.

Encadeando optionals

Além do `ifPresentOrElse` e `stream`, o `Optional` finalmente recebeu um método `or`, que nos permite encadear diversas alternativas de fallback em um mesmo resultado.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Nossa busca de similares, por exemplo, poderíamos passar outras opções de livros como a seguir:

```
Books.findSimilar(book)
    .or(() -> Books.findSimilar(book2))
    .or(() -> Books.findSimilar(book3))
    .or(() -> Books.findSimilar(book4));
```

3.5 MAIS NOVIDADES

Foram diversas as novidades nas APIs, além das principais aqui listadas. O pacote `java.time` , por exemplo, recebeu uma forma bastante interessante de retornar *streams* com intervalo de datas:

```
Stream<LocalDate> dates =
    LocalDate.datesUntil(java9Release);
```

O `Scanner` também recebeu novos métodos que tiram proveito da API de Streams, como o `findAll` que, dada uma expressão regular, retorna um stream de possíveis resultados.

```
String regex = "\\d+";
String input = "esperei 3 anos pelo lançamento do java 9";

List<String> matches = new Scanner(input)
    .findAll(regex)
    .map(MatchResult::group)
    .collect(toList());
```

Nesse caso o resultado será uma lista com os valores 3 e 9.

Você pode ver uma lista completa de introduções pelo link:

http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-annex-1.html

3.6 DOWNLOAD DOS EXEMPLOS DESTE

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO

Todos os exemplos de código vistos aqui estão disponíveis no diretório deste capítulo, no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 4

HTTP/2 CLIENT API

Ao acessar um site pelo seu navegador, você está utilizando o protocolo HTTP, que existe desde a popularização da internet na década de 90. Em 2015, o *Internet Engineering Task Force* (IETF) publicou a especificação para o HTTP/2 que, diferente de sua versão mais comum, o HTTP/1.1, funciona de uma forma muito mais eficiente para troca de pacotes de dados entre cliente e servidor.

Alguns dos diferenciais que tornam essa nova versão tão interessante e performática em relação a sua antecessora são: o uso de header binários e comprimidos; o fato de as requisições e respostas serem paralelas automaticamente em uma única conexão; e a possibilidade de definição de prioridade de requisições do lado do cliente.

O JDK 9 não só suporta HTTP/2, como também oferece uma nova API para HTTP/2 e *WebSockets*.

4.1 O PRIMEIRO DOS INCUBATOR MODULES

O *HTTP/2 Client* entrou no JDK 9 como um módulo em incubação. Esse é um novo conceito da plataforma, com o objetivo de permitir que APIs passem por um período de testes antes de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

entrar definitivamente para a linguagem. O grande objetivo dessa iniciativa é reduzir a possibilidade de introdução de erros na plataforma, que tem o forte peso da retrocompatibilidade.

Módulos em incubação ainda não fazem parte do Java SE. Eles ficam em um pacote `jdk.incubator` e não são resolvidos por padrão na compilação ou execução de suas aplicações. Para usar um módulo em incubação, você precisa declará-lo explicitamente, como veremos adiante.

No JDK 10, essa API, em fase de incubação, poderá ser integrada oficialmente para o Java SE ou simplesmente removida. Essa segunda opção só aconteceria em caso de erros graves, em que seu uso fosse completamente inviabilizado.

4.2 PREPARANDO SEU AMBIENTE

Para utilizar uma API que está em fase de incubação, você precisará passar a opção `--add-modules` com seu *namespace* que, no caso do *HTTP/2 Client*, é o `jdk.incubator.httpclient`. Essa opção funciona exatamente da mesma forma caso você esteja utilizando o JShell:

```
jshell --add-modules jdk.incubator.httpclient
```

Ou compilando e executando pela linha de comando:

```
javac --add-modules jdk.incubator.httpclient
java --add-modules jdk.incubator.httpclient
```

4.3 REQUESTS HTTP DA FORMA ANTIGA

O Java suporta HTTP desde sua primeira versão, mas a sua

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

implementação, conhecida como *HttpURLConnection API*, possui uma série de limitações e problemas de design. Falta de documentação em alguns de seus comportamentos, sua característica de suportar muitos tipos diferentes de conexões (`http` , `ftp` , `gopher` etc.) em um mesmo design agnóstico, e a limitação ao HTTP/1.1 são alguns deles. Outro ponto é que seu uso era bloqueante, isto é, era permitida apenas uma *thread* por requisição e resposta.

O código a seguir faz uma requisição HTTP para o site da Casa do Código, usando *HttpURLConnection API*, e imprime todas as linhas do HTML de sua resposta:

```
URL url = new URL("https://www.casadocodigo.com.br/");
URLConnection urlConnection = url.openConnection();

BufferedReader reader = new BufferedReader(
    new InputStreamReader(urlConnection.getInputStream()));

String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
reader.close();
```

Experimente executá-lo. O resultado, de forma simplificada, será:

```
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<meta name="viewport" content="width=device-width">

<title>Livros de Java [...] e muito mais - Casa do Código</title>

<meta property="og:title"
    content="Casa do Código - Livros para o programador"/>

<!-- e mais um monte de linhas html -->
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Funciona e não parece tão assustador, especialmente em um exemplo tão simples. O grande problema está na quantidade de limitações e dificuldade de manutenção desse tipo de código. E não é por menos, estamos falando de uma API escrita 20 anos atrás. Ela é tão antiga quanto o próprio HTTP/1.1.

Alternativas

Existem bibliotecas que ajudam com esse trabalho. O *HttpClient* da Apache e o Jetty estão entre as mais conhecidas.

- Apache HttpClient:

  https://hc.apache.org/httpcomponents-client-ga/

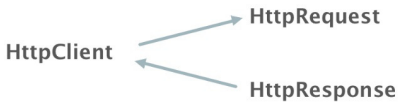
- Jetty:

  http://eclipse.org/jetty

O problema dessas APIs é que não tiram proveito dos novos recursos da linguagem, como expressões lambda. Elas também são dependências externas bem mais pesadas, e que exigem um setup maior do que uma solução nativa da própria plataforma.

4.4 O NOVO HTTP/2 CLIENT API

A nova API lida com conexões HTTP a partir das três principais classes abstratas: `HttpClient`, `HttpRequest` e `HttpResponse`.



E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Todas essas classes vêm do pacote `jdk.incubator.http`. Se você estiver usando o JShell, que eu recomendo bastante, pode declará-lo desde já com o comando:

```
import jdk.incubator.http.*
```

A primeira delas, o `HttpClient`, tem a responsabilidade principal de criar e enviar requisições. Para criar uma instância desse tipo, podemos usar seu *factory method* `newHttpClient`:

```
HttpClient httpClient = HttpClient.newHttpClient();
httpClient.send(...)
```

Seu método `send` recebe um `HttpRequest` como parâmetro, e retorna um `HttpResponse`. Um request pode ser construído a partir do método `newBuilder`, que constrói uma instância do tipo `HttpRequest.Builder`.

Esse builder é responsável por preparar a requisição com, pelo menos, as informações de URI e o método HTTP que deverá ser utilizado. Para quebrar esse processo em partes, vamos criar uma URI apontando para o site que deverá ser chamado:

```
URI uri = new URI("https://turini.github.io/livro-java-9/");
```

E agora criar um `HttpRequest` que prepara uma requisição do tipo `GET` para essa URI:

```
HttpRequest request =
    HttpRequest.newBuilder().uri(uri).GET().build();
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Perceba que o builder é uma interface fluente, que retorna uma instância dele mesmo com os valores inseridos. Por fim, seu método `build` fecha o ciclo com a construção de um `HttpRequest` pronto para uso. Diversas outras informações podem ser passadas nesse builder, como veremos adiante.

Além do `HttpRequest`, o método `send` também recebe um `HttpResponse.BodyHandler`, um tipo responsável por informar como a *Client API* deve ler e transformar esse resultado. Para conhecer as possibilidades, podemos digitar o começo do método e usar o atalho `Tab` pelo ambiente do JShell:

```
jshell> HttpResponse.BodyHandler.as
    asByteArray()
    asByteArrayConsumer(
    asFile(
    asFileDownload(
    asString(
```

Esse já é um diferencial bem grande e importante da nova API. É extremamente simples dizer como o valor da resposta HTTP deve ser projetado.

Podemos enviar a requisição passando o `request`, e informando que queremos o resultado como uma `String`:

```
HttpResponse<String> response =
    httpClient.send(request, HttpResponse.BodyHandler.asString())
;
```

Com o `HttpResponse` preparado, podemos consultar o **status code** da resposta, a versão HTTP usada e o seu conteúdo.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
System.out.println("http version: " + response.version());
System.out.println("status code: " + response.statusCode());
System.out.println("content: " + response.body());
```

A API ficou bastante expressiva! É extremamente simples entender seu funcionamento pela documentação e nomenclatura dos métodos.

Nosso código completo ficou assim:

```
URI uri = new URI("https://turini.github.io/livro-java-9/");

HttpClient httpClient = HttpClient.newHttpClient();

HttpRequest request =
    HttpRequest.newBuilder().uri(uri).GET().build();

HttpResponse<String> response =
    httpClient.send(request, HttpResponse.BodyHandler.asString())
;

System.out.println("http version: " + response.version());
System.out.println("status code: " + response.statusCode());
System.out.println("content: " + response.body());
```

Vamos executar? Se tudo correr bem, o resultado será:

```
http version: HTTP_2
status code: 200
content: Hello, HTTP/2 client!
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

O CÓDIGO NÃO FUNCIONOU?

É importante lembrar que, como estamos fazendo uma requisição HTTP, você precisará estar conectado à internet para testar esse código. Caso esteja offline, o resultado será um `java.net.ConnectException thrown: Operation timed out`.

Vale lembrar também que o módulo do HTTP/2, que ainda está em fase de incubação, precisa ser explicitamente declarado conforme vimos na seção de preparação do ambiente do início deste capítulo.

Opcionalmente, agora que já conhecemos seus tipos, podemos deixar o código um pouco menos verboso com uso de `imports` estáticos e removendo as variáveis intermediárias, que podem ficar em uma única expressão fluente.

```
jshell> import static jdk.incubator.http.HttpClient.newHttpClient
;

jshell> import static jdk.incubator.http.HttpRequest.newBuilder;

jshell> import static jdk.incubator.http.HttpResponse.BodyHandler
.asString;

HttpResponse<String> response = newHttpClient().send(
    newBuilder()
      .uri(new URI("https://turini.github.io/livro-java-9/"))
      .GET().build()
    , asString());
```

Com essa nova possibilidade, conseguimos fazer requisições HTTP de forma nativa e bastante eficiente, sem depender de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

bibliotecas externas e tirando bastante proveito dos recursos da linguagem em uma API amigável.

4.5 LIDANDO COM REDIRECTS

Caso ainda não tenha feito, e se deparado com um problema desse tipo, experimente acessar a URL do *google.com* (sem `.br`) usando o cliente HTTP, da mesma forma que fizemos para o site da Casa do Código e para URL de exemplo do repositório do livro:

```
String content = HttpClient.newHttpClient().send(
    HttpRequest.newBuilder()
        .uri(new URI("https://google.com/"))
        .GET().build(),
    HttpResponse.BodyHandler.asString()).body();
```

Ao executar, o HTML de resposta será parecido com o seguinte, com status 302:

```
<html>
<head>
  <title>302 Moved</title>
</head>
<body>
  <h1>302 Moved</h1>
  The document has moved
  <a href="https://www.google.com.br/">here</a>.
</body>
</html>
```

Isso acontece pois o *google.com*, em condições normais quando acessado por um servidor do Brasil, faz um *redirect* para o *google.com.br*, e a política de redirects do *HTTP/2 Client* por padrão não acompanha o redirecionamento das requisições.

Podemos configurar nosso cliente HTTP usando alguma das demais políticas de *redirect* definidas no *enum*

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`HttpClient.Redirect`:

- `HttpClient.Redirect.NEVER` é o padrão, que já conhecemos, por não acompanhar nenhum tipo de redirecionamento.
- `HttpClient.Redirect.ALWAYS` é justamente o oposto do `NEVER`, que sempre acompanha, independente da origem ou do protocolo.
- `HttpClient.Redirect.SAME_PROTOCOL` apenas permite redirecionamentos para o mesmo protocolo. Uma requisição HTTP só pode ser redirecionada para outro HTTP, por exemplo.
- `HttpClient.Redirect.SECURE` permite que o *redirect* sempre ocorra, com exceção de quando a requisição vem do protocolo HTTPS e tenta ser redirecionada para HTTP.

Agora que já conhecemos as possibilidades, podemos configurar o nosso `HttpClient` com alguma dessas políticas.

4.6 CONFIGURANDO O HTTPCLIENT

Até agora, usamos o `HttpClient.newHttpClient()` para criar novas instâncias desse tipo. Porém, para preparar esse objeto com configurações adicionais como sua política de redirects, versão, prioridade entre outros, podemos usar o `HttpClient.Builder`:

```
HttpClient.Builder httpClientBuilder = HttpClient.newBuilder();
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Com ele, podemos facilmente definir, por exemplo, a política de redirects como `SECURE` :

```
HttpClient.newBuilder()
  .followRedirects(HttpClient.Redirect.SECURE)
  .build();
```

Ou ainda, deixar explícito qual a versão do HTTP que deve ser utilizada.

```
HttpClient.newBuilder()
  .followRedirects(HttpClient.Redirect.SECURE)
  .version(HttpClient.Version.HTTP_2)
  .build();
```

O código completo, desse mesmo exemplo do Google, ficará assim:

```
HttpClient.newBuilder()
  .followRedirects(HttpClient.Redirect.SECURE)
  .version(HttpClient.Version.HTTP_2)
  .build()
  .send(HttpRequest.newBuilder()
    .uri(new URI("https://google.com/"))
    .GET()
    .build(),
  HttpResponse.BodyHandler.asString()).body();
```

E o resultado, de forma simplificada, será a página final do Google Brasil:

```
<html>
  <head>
    <meta content="Feliz Dia dos Pais!" property="description">
    <title>Google</title>
  </head>
</body>
  <input title="Pesquisa Google" name="q">
  <input value="Pesquisa Google" name="btnG" type="submit">
  <input value="Estou com sorte" name="btnI" type="submit">
</body>
</html>
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Essas são algumas das muitas configurações que você pode fazer com uso dos *builders* do `Http Client` . *Proxy*, *Cookies* e *Authentication* são algumas das várias opções. Outro exemplo muito comum seria a definição dos *headers* em uma requisição:

```
HttpRequest.newBuilder()
  .header("Content-Type", "application/x-www-form-urlencoded")
  .header("Accept", "text/plain");
```

Como conhecer todos eles? Na prática!

Experimente fazer diferentes tipos de requisições e ir além dos exemplos daqui, para se familiarizar com a nova API.

Você pode ver mais possibilidades do protocolo HTTP/2 em:

https://http2.github.io

4.7 BUSCANDO LIVROS COM HTTP/2 CLIENT

Agora que já conhecemos um pouco mais da anatomia da nova API de *HTTP/2 Client*, podemos tirar proveito desse recurso em nossa classe `Books` , criada para encapsular o acesso aos dados dos livros de nossa editora. No lugar da lista fixa que o método `all` retorna, podemos consumir as informações a partir da URL a seguir:

https://turini.github.io/livro-java-9/books.csv

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Nota do autor

Ao acessar a URL, você perceberá que a lista de livros está em formato CSV (*comma-separated values*) no lugar de um JSON ou mesmo XML, que são formatos bastante comuns para esse tipo de troca de informações em serviços web. O motivo da escolha é que o Java ainda não tem uma forma nativa para leitura de JSONs, e não queremos complicar o setup dos testes e desviar o foco dos recursos da linguagem com o uso de bibliotecas externas.

O JSON API, que ficou de fora dessa versão, está sendo considerado para o JDK 10: http://openjdk.java.net/jeps/198

Listando e projetando livros

Podemos usar o mesmo código de antes para fazer uma requisição HTTP, agora para a URL `/books.csv`. Para evitar variáveis desnecessárias no código, vou retornar o resultado diretamente para uma String.

```
String csv = HttpClient.newHttpClient()
  .send(HttpRequest.newBuilder()
    .uri(new URI("https://turini.github.io/livro-java-9/books.csv
))
    .GET().build(),
    HttpResponse.BodyHandler.asString()).body();
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Para evitar confusões nos `imports` das classes, vou evitar os estáticos! No dia a dia, você pode e deve usá-los sempre que sentir que isso pode melhorar a legibilidade e simplificar seu código. Um exemplo seria na própria criação do cliente http:

```
HttpClient.newHttpClient()
```

Perceba que a palavra *HttpClient* já está presente no nome do método, portanto nesse caso usar um importe estático certamente será uma boa opção, evitando ambiguidades sem comprometer a fácil compreensão do código.

Execute para ter certeza de que tudo está funcionando. O resultado deverá ser todo texto do conteúdo CSV da URL:

```
"Desbravando Java e Orientação a Objetos,2014-03-18,Rodrigo Turin
i,PROGRAMMING\nExplorando APIs e bibliotecas Java,2014-03-18,Rodr
igo Turini,PROGRAMMING\nJava 8 Prático,2014-03-18,Rodrigo Turini,
PROGRAMMING\nIntrodução e boas práticas em UX Design,2014-06-23,F
abricio Teixeira,DESIGN\nConhecendo o Adobe Photoshop CS6,2012-06
-05,Tárcio Honório Zemel,DESIGN\nEdição e organização de fotos co
m Adobe Lightroom,2012-06-05,Rafael Giovanni Steil,DESIGN\nMétric
as Ágeis,2017-06-02,Raphael Donaire Albino,AGILE\nScrum: Gestão á
gil para projetos de sucesso,2013-06-17,Rafael Sabbagh Armony,AGI
LE\nTest-Driven Development,2012-09-10,Mauricio Finavaro Aniche,A
GILE\nDireto ... p,2012-06-05,Joaquim Jose Rodrigues Torres,BUSIN
ESS\nO Mantra da Produtividade,2015-08-05,Dionatan de Souza Moura
,BUSINESS\nJava SE 7 Programmer I,2013-01-30,Guilherme de Azevedo
 Silveira,CERTIFICATION\nJava SE 8 Programmer I,2014-11-05,Mário
Amaral,CERTIFICATION\nZend Certified Engineer,2016-04-22,Matheus
Marabesi,CERTIFICATION\n"
```

Precisamos agora converter esse CSV com todos os dados em uma lista de livros.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Perceba que existe um `\n` separando cada uma das linhas. Portanto, podemos usar essa informação para separar cada um desses livros, com o método `split` da `String`, e depois envolver o resultado dentro de um `Stream`, que vai nos ajudar a fazer as demais operações nesse conteúdo.

```
Stream.of(csv.split("\n"))
```

O próximo passo será pegar o valor de cada uma dessas linhas, separado por vírgula, e transformar em uma classe `Book`. Podemos criar um método na classe `Books` para encapsular esse trabalho. Uma possível implementação seria:

```
public class Books {
    //outros métodos...

    public static Book create(String line) {
        String[] split = line.split(",");
        String name = split[0];
        String author = split[2];
        Category category = Category.valueOf(split[3]);
        return new Book(name, author, category);
    }
}
```

Agora podemos projetar esse valor com a operação de `map` do `Stream` em cada uma das linhas, e coletar o resultado em uma lista:

```
Stream.of(csv.split("\n"))
    .map(Books::create)
    .collect(Collectors.toList());
```

O método `all` da classe `Books` ficará assim:

```
public class Books {
    //outros métodos...

    public static List<Book> all() {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
        String csv = HttpClient.newHttpClient()
        .send(HttpRequest.newBuilder()
            .uri(new URI("https://turini.github.io/livro-java-9/b
ooks.csv"))
            .GET().build(),
        HttpResponse.BodyHandler.asString()).body();

        return Stream.of(csv.split("\n"))
            .map(Books::create)
            .collect(Collectors.toList());
    }
}
```

Mas ele ainda não compila.

```
Error:
unreported exception java.net.URISyntaxException;
must be caught or declared to be thrown
new URI("https://turini.github.io/livro-java-9/books.csv")

Error:
unreported exception java.io.IOException;
must be caught or declared to be thrown
HttpClient.newHttpClient()
```

Isso acontece pois, ao criar a URI, não estamos tratando a possível `URISyntaxException` e nem o `IOException` ou a `InterruptedException`, que pode acontecer na execução do `HttpClient`. Podemos resolver com um *try/catch* simples, que propaga qualquer erro em caso de falhas:

```
public class Books {
    //outros métodos...

    public static List<Book> all() {
        try {
            String csv = HttpClient.newHttpClient()
            .send(HttpRequest.newBuilder()
                .uri(new URI("https://turini.github.io/livro-java
-9/books.csv"))
                .GET().build(),
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
            HttpResponse.BodyHandler.asString()).body();

            return Stream.of(csv.split("\n"))
                .map(Books::create)
                .collect(Collectors.toList());

        } catch (Exception e) {
            throw new RuntimeException("Não foi possível conectar
 ", e);
        }
    }
}
```

Agora sim, tudo parece ok. A classe `Books` completa ficará assim:

```
public class Books {

    public static List<Book> all() {
        try {
            String csv = HttpClient.newHttpClient()
            .send(HttpRequest.newBuilder()
                .uri(new URI("https://turini.github.io/livro-java
-9/books.csv"))
                .GET().build(),
            HttpResponse.BodyHandler.asString()).body();

            return Stream.of(csv.split("\n"))
                .map(Books::create)
                .collect(Collectors.toList());

        } catch (Exception e) {
            throw new RuntimeException("Não foi possível conectar
 ", e);
        }
    }

    public static Book create(String line) {
        String[] split = line.split(",");
        String name = split[0];
        String author = split[2];
        Category category = Category.valueOf(split[3]);
        return new Book(name, author, category);
    }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    public static Optional<Book> findSimilar(Book book) {
        return Books.all().stream()
            .filter(b -> b.getCategories().equals(book.getCategor
ies()))
            .filter(b -> !b.getAuthor().equals(book.getAuthor()))
            .findAny();
    }
}
```

Para testá-la, podermos chamar seu método `all`, imprimindo o nome de cada um dos livros:

```
Books.all().stream().map(Book::getName)
    .forEach(System.out::println);
```

Se tudo correu bem, o resultado será:

```
Desbravando Java e Orientação a Objetos
Explorando APIs e bibliotecas Java
Java 8 Prático
Introdução e boas práticas em UX Design
Conhecendo o Adobe Photoshop CS6
Edição e organização de fotos com Adobe Lightroom
Métricas Ágeis
Scrum: Gestão ágil para projetos de sucesso
Test-Driven Development
Direto ao Ponto
Guia da Startup
O Mantra da Produtividade
Java SE 7 Programmer I
Java SE 8 Programmer I
Zend Certified Engineer
```

4.8 REQUISIÇÕES ASSÍNCRONAS

Em todos nossos exemplos, utilizamos o método `httpClient.send(...)`, que funciona de forma totalmente síncrona. Ele bloqueia o código até que a resposta esteja completa, mas nem sempre é isso que queremos, em especial em requisições

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

demoradas ou que não influenciam no restante do código.

Uma outra novidade da nova API é que você consegue enviar requisições assíncronas facilmente. O processo continua executando em *background*, enquanto a *thread* principal fica livre para executar outras tarefas.

E seu uso é bastante simples! No lugar do método `send`, como estávamos fazendo, basta utilizar o `sendAsync`:

```
httpClient.sendAsync(request, ...)
```

A grande diferença no código é que, no lugar do `client` retornar um `HttpResponse`, ele retornará um `CompletableFuture`, que nos informará quando essa resposta está pronta para uso:

```
CompletableFuture<HttpResponse<String>> response =
HttpClient.newHttpClient()
    .sendAsync(HttpRequest.newBuilder()
        .uri(new URI("https://turini.github.io/livro-java-9/books
.csv"))
        .GET().build(),
        HttpResponse.BodyHandler.asString());
```

Com esse `CompletableFuture` em mãos, poderíamos, por exemplo, perguntar logo em seguida se ele está pronto com seu método `isDone`. Se estiver, imprimimos a saída; caso contrário, cancelamos sua execução.

```
if (response.isDone()) {
    System.out.println(response.get().body());
} else {
    System.out.println("cancelando o request");
    response.cancel(true);
}
```

Esse código poderá ou não funcionar, dependendo do tempo e

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

da latência que o servidor responderá. Nesse caso, em que a quantidade de informações é bem pequena, ele deverá funcionar na maior parte das vezes, mas não é bem isso que queremos em nosso código.

Uma forma um pouco mais interessante de trabalhar com esse `CompletableFuture` seria adicionando um *callback* para que, quando ele estiver completo, nos avise imprimindo o valor de sua saída. Isso pode ser feito com o método `whenComplete`:

```
response.whenComplete((r,t) -> System.out.println(r.body()))
```

Além do `HttpResponse`, com a resposta da requisição que foi disparada, a expressão lambda desse método recebe um `Throwable t` como parâmetro, para que possamos propagar o motivo de uma possível exception em sua execução.

> Você pode usar qualquer outro método do `CompletableFuture`, além do `isDone` e do `whenComplete` aqui vistos. Algumas das outras possibilidades são: o `completeExceptionally`, para casos de exception, e `thenApply` e `thenApplyAsync` para compor as funções que devem ser executadas.

O código da execução assíncrona poderia ficar assim:

```
HttpClient.newHttpClient()
    .sendAsync(HttpRequest.newBuilder()
        .uri(new URI("https://turini.github.io/livro-java-9/books
.csv"))
        .GET().build(),
        HttpResponse.BodyHandler.asString())
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    .whenComplete((r,t) -> System.out.println(r.body()));
```

E o resultado, assim como antes, será a conteúdo da resposta HTTP, que é o nosso CSV:

```
Desbravando Java e Orientação a Objetos,2014-03-18,Rodrigo Turini
,PROGRAMMING
Explorando APIs e bibliotecas Java,2014-03-18,Rodrigo Turini,PROG
RAMMING
Java 8 Prático,2014-03-18,Rodrigo Turini,PROGRAMMING
Introdução e boas práticas em UX Design,2014-06-23,Fabricio Teixe
ira,DESIGN
Conhecendo o Adobe Photoshop CS6,2012-06-05,Tárcio Honório Zemel,
DESIGN
Edição e organização de fotos com Adobe Lightroom,2012-06-05,Rafa
el Giovanni Steil,DESIGN
Métricas Ágeis,2017-06-02,Raphael Donaire Albino,AGILE
Scrum: Gestão ágil para projetos de sucesso,2013-06-17,Rafael Sab
bagh Armony,AGILE
Test-Driven Development,2012-09-10,Mauricio Finavaro Aniche,AGILE
Direto ao Ponto,2015-08-17,Paulo Roberto Celidonio Caroli,BUSINES
S
Guia da Startup,2012-06-05,Joaquim Jose Rodrigues Torres,BUSINESS
O Mantra da Produtividade,2015-08-05,Dionatan de Souza Moura,BUSI
NESS
Java SE 7 Programmer I,2013-01-30,Guilherme de Azevedo Silveira,C
ERTIFICATION
Java SE 8 Programmer I,2014-11-05,Mário Amaral,CERTIFICATION
Zend Certified Engineer,2016-04-22,Matheus Marabesi,CERTIFICATION
```

4.9 CRIANDO UM ARQUIVO CSV NO RETORNO

Nosso exemplo assíncrono funciona, mas, no geral, não queremos retornar o corpo do `HttpResponse` como uma `String`, da forma que fizemos em todos os exemplos daqui. Poderíamos facilmente usar algum outro tipo de resposta, como o `HttpResponse.BodyHandler.asFile()`, que transforma o resultado em um arquivo.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

A mudança seria bem pequena. No lugar do `HttpResponse.BodyHandler.asString()`, como fizemos até agora, faremos:

```
HttpResponse.BodyHandler.asFile(Paths.get("books.csv"));
```

Perceba que o método recebe um `File`, como parâmetro, indicando em qual arquivo a resposta deverá ser escrita. No método de *callback*, `whenComplete`, podemos avisar que o arquivo foi criado e apontar o caminho de seu diretório, como a seguir:

```
whenComplete((r,t) ->
  System.out.println("arquivo salvo em: "+ r.body().toAbsolutePat
h()));
```

O código completo ficará assim:

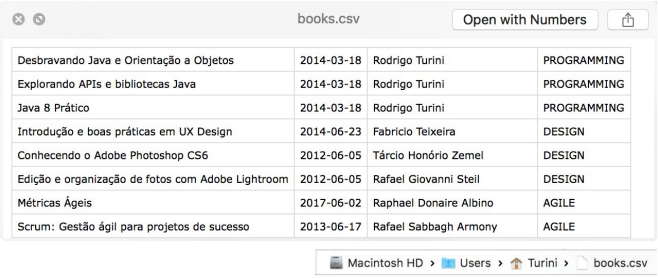
```
HttpClient.newHttpClient()
    .sendAsync(HttpRequest.newBuilder()
        .uri(new URI("https://turini.github.io/livro-java-9/books
.csv"))
        .GET().build(),
        HttpResponse.BodyHandler.asFile(Paths.get("books.csv")))
    .whenComplete((r,t) ->
        System.out.println("arquivo salvo em: "+ r.body().toAbsol
utePath()));
```

E, ao executá-lo, a resposta em meu caso foi:

```
arquivo salvo em: /Users/Turini/books.csv
```

Conforme esperado, o arquivo foi criado nesse mesmo diretório.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com



books.csv Open with Numbers

Desbravando Java e Orientação a Objetos	2014-03-18	Rodrigo Turini	PROGRAMMING
Explorando APIs e bibliotecas Java	2014-03-18	Rodrigo Turini	PROGRAMMING
Java 8 Prático	2014-03-18	Rodrigo Turini	PROGRAMMING
Introdução e boas práticas em UX Design	2014-06-23	Fabricio Teixeira	DESIGN
Conhecendo o Adobe Photoshop CS6	2012-06-05	Tárcio Honório Zemel	DESIGN
Edição e organização de fotos com Adobe Lightroom	2012-06-05	Rafael Giovanni Steil	DESIGN
Métricas Ágeis	2017-06-02	Raphael Donaire Albino	AGILE
Scrum: Gestão ágil para projetos de sucesso	2013-06-17	Rafael Sabbagh Armony	AGILE

Macintosh HD › Users › Turini › books.csv

4.10 OUTRAS NOVIDADES DO HTTP/2 CLIENT API

A API não para por aí. Além das requisições HTTP, síncronas e assíncronas, agora também temos uma forma simples de trabalhar com *WebSockets*. Caso você não esteja familiarizado com o termo, esse é um protocolo de comunicação entre dois *endpoints*, do lado do cliente e servidor.

Um exemplo seria um cliente que imprime todas as mensagens que recebe através de um servidor *WebSocket*:

```
WebSocket.newBuilder(
    new URI("ws://url-do-meu-websocket"),
    new WSListener()
)
.buildAsync().join();
```

A classe `WSListener` é uma implementação simples da nova interface `WebSocket.Listener`, que define como o *listener* deve se comportar quando novas mensagens são recebidas:

```
public class WSListener implements WebSocket.Listener {
  @Override
  public CompletionStage<T> onText(WebSocket webSocket,
      CharSequence message, WebSocket.MessagePart part) {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    webSocket.request(1);

    return CompletableFuture.completedFuture(message)
      .thenAccept(System.out::println);
  }
}
```

Se quiser saber mais detalhes desse e outros detalhes avançados da nova API, poderá gostar de ver a definição principal de sua proposta:

http://openjdk.java.net/jeps/110

E também a documentação de *WebSockets*:

http://download.java.net/java/jdk9/docs/api/jdk/incubator/http/WebSocket.MessagePart.html

4.11 DOWNLOAD DOS EXEMPLOS DESTE CAPÍTULO

Todos os exemplos de código vistos aqui estão disponíveis no diretório deste capítulo, no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 5

REACTIVE STREAMS

Trabalhar com fluxos assíncronos, não bloqueantes, sempre foi um desafio muito grande. Neste capítulo, vamos entender alguns desses problemas na prática e também como o JDK 9 evoluiu a arquitetura de seu pacote `java.util.concurrent`, oferecendo suporte à elegante solução de *Reactive Streams*. Ela, apesar de existir há bastante tempo, se tornou muito popular nos dias de hoje e está presente em diferentes linguagens.

Apesar do nome extremamente familiar, *Reactive Streams* não tem relação direta com a API de *Streams* do Java com que já estamos acostumados. Essa é uma confusão completamente comum. O termo *Streams*, nesse caso, remete a fluxo, ou seja, fluxos reativos, que é um conceito bastante conhecido e foi criado a partir do caminho definido pelo famoso manifesto reativo.

Se nunca ouviu falar no manifesto, certamente vai se interessar:

http://www.reactivemanifesto.org

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Existem diversas implementações de fluxos reativos na JVM:

- *RXJava* https://github.com/ReactiveX/RxJava
- *Akka Streams* http://doc.akka.io/docs/akka/current/java/stream/index.html
- *Project Reactor* https://projectreactor.io
- E o próprio *Reactive Streams* http://www.reactive-streams.org

Além disso, houve uma iniciativa de definição de APIs comuns no projeto *Reactive Streams*, que pode ser encontrada em:

https://github.com/reactive-streams/reactive-streams-jvm

O grande desafio do JDK 9 não é criar implementações que façam esse trabalho, pois elas já existem e funcionam muito bem. O objetivo principal é definir um conjunto de interfaces, padronizando esse processo em uma estrutura clara. Foi então que surgiu a Flow API, que vamos explorar durante o capítulo.

Por onde começar?

Podemos entender o problema de forma prática, em uma situação real de nossa editora.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

5.1 EMISSÃO DE NOTAS FISCAIS

Sempre que vendemos um livro, precisamos emitir uma nota fiscal. Essa nota precisa ter, pelo menos, os dados do cliente, do produto e o valor pago. Certamente muitas outras informações são necessárias, mas podemos representar esses dados de uma forma simplificada com a classe:

```
public class NF {

  private String client;
  private String book;
  private double amount;

  public NF(String client, String book, double amount) {
    this.client = client;
    this.book = book;
    this.amount = amount;
  }

  public boolean hasValidAmount() {
    return amount > 0;
  }
}
```

Toda nota precisa ser enviada para a prefeitura de nossa cidade. Para simular esse processo, podemos criar uma classe que simule um *web service* que faz a integração, inclusive com uma demora de 5 segundos, para nos aproximar de um exemplo de execução real.

```
public class WSPrefeitura {

  public static void emit(NF nf) {
    try {
      System.out.println("emitindo...");
      Thread.sleep(5000);
      System.out.println("emitido!");
    } catch(Exception e) {
      System.out.println("falha ao emitir a nf");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    }
  }
}
```

Agora que já temos essa estrutura definida, podemos criar um fluxo para que, a cada venda, uma nova nota seja criada e enviada para a prefeitura. O código ficaria parecido com este:

```
// venda concluída
System.out.println("Gerando a nota");
NF nf = new NF("Turini", "Livro Java 9", 39.99);
WSPrefeitura.emit(nf);
// outras ações necessárias
System.out.println("Parabéns pela sua compra");
```

Parece ok, certo? Não há nada de diferente ou especial nesse código. Vamos executar:

```
Gerando a nota
emitindo...
emitido!
Parabéns pela sua compra
```

Tudo correu conforme esperado, mas, devido à demora do processo de comunicação do nosso serviço de emissão de nota fiscal, o cliente precisou ficar esperando.

Esse código em um loop demoraria bastante, já que seria necessário, pelo menos, os 5 segundos de espera a cada nota. Além disso, corremos o risco de um erro acontecer durante o envio da nota e, no lugar de uma mensagem de parabenização pela compra, o cliente receberia um erro. Não queremos nada disso.

Apesar de o processo ser obrigatório, ele poderia facilmente ser paralelizado. Não queremos bloquear a compra, ou a liberação do livro para o leitor, enquanto emitimos uma nota fiscal. **Nenhuma dessas etapas precisa ser síncrona.**

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

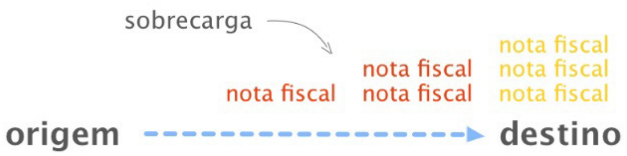
Emitindo de forma assíncrona

No lugar dessa abordagem bloqueante, queremos emitir as notas fiscais de forma assíncrona, em paralelo ao processo de venda. Na teoria, poderíamos fazer isso de uma forma simples, bastando executar o código de emissão em uma *thread* separada.

O problema é que existem vários detalhes com que precisamos nos preocupar ao lidar com fluxos de processamento de dados em execuções assíncronas. Um dos maiores deles é quando você envia um volume de dados arbitrário como este, já que em um mesmo dia podemos vender 10 livros, ou milhões deles.

Toda venda vai ser enviada para emissão de nota fiscal, mas qual a garantia de que o componente que vai receber todas essas notas dará conta de processá-las?

Esse é o famoso problema de processamentos em que um componente de origem envia dados sem saber ao certo se isso está em uma quantidade maior do que aquela com a qual o consumidor pode lidar. É necessário que exista um mecanismo de segurança, para evitar esse tipo de sobrecarga ou, pelo menos, avisar que ele está acontecendo.



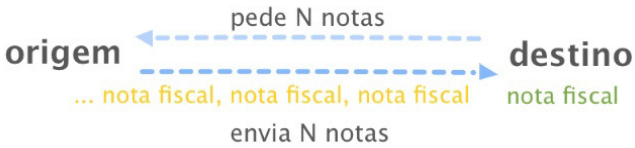
No exemplo da imagem, o componente de origem está recebendo e emitindo várias notas, mas perceba que muitas delas

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

estão bloqueadas enquanto as outras estão sendo processadas. Os dados chegam em uma velocidade muito maior do que podem ser atendidos.

Poderíamos sim, antes de enviar, conferir se o componente de destino tem capacidade de receber mais notas. Porém, para isso, teríamos de voltar ao modelo síncrono e ao problema original.

A forma de resolver esse problema seria mudar completamente essa estratégia de envio. No lugar de a **origem empurrar uma quantidade arbitrária de dados para o destino**, podemos fazer com que o destino **puxe apenas a quantidade de dados que ele certamente poderá atender**. Só então, os dados são enviados e na quantidade certa.



Essa é a grande motivação por trás do conceito de fluxos reativos, *Reactive Streams*, que é uma iniciativa criada para fornecer um padrão de processamento de fluxos assíncronos como este que vimos. Ela leva em consideração o *back pressure* e outras preocupações fundamentais que precisamos ter em processamentos assíncronos.

No formato reativo, nosso componente vai sempre pedir uma nota, e só então o publicador enviará. Aconteceu um erro? Acabaram as notas? Não se preocupe, a arquitetura conta com todos esses detalhes para você, e oferece formas simples de definir

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

como o componente precisa reagir em cada um desses casos.

Flow API

O JDK 9 introduz a *Flow API*, no pacote `java.util.concurrent`, responsável por prover suporte a esse padrão de fluxos reativos. Ela é composta pela classe `Flow`, e as interfaces internas: `Publisher`, `Subscription`, `Subscriber` e `Processor`. Cada uma delas será utilizada para a solução do nosso problema de notas fiscais, para que suas responsabilidades e seu uso fiquem bem claros.

De forma resumida, um `Publisher` é o responsável por enviar os dados para o `Subscriber`, **na medida certa em que ele é capaz de atender**. O `Subscription` é o link entre esses dois, responsável por dizer quando e quantos itens podem ser enviados.

5.2 PRIMEIRO CONTATO COM FLUXO REATIVO

Vamos fazer o mesmo trabalho de antes, da emissão da nota fiscal, agora tirando proveito dessa nova API de fluxos reativos. Essa será a oportunidade perfeita para entender cada uma dessas interfaces, em um exemplo real.

A primeira etapa seria criando um publicador, representado

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

pela interface `Publisher`, que será o responsável pelo envio dos dados da nota para emissão.

A nova API oferece uma implementação simplificada, no mesmo pacote `java.util.concurrent`, chamada `SubmissionPublisher`. Esta possui uma série de métodos convenientes para facilitar esse primeiro contato. Repare no uso do *generics*, indicando que a classe a seguir será um publicador de notas fiscais:

```
SubmissionPublisher<NF> publisher = new SubmissionPublisher<>();
```

Para um primeiro teste, podemos usar o seu método `consume`, que abstrairá toda a parte do `Subscriber`. Ele recebe um `Consumer` como parâmetro, que indicará qual ação deverá ser executada sempre que esse publicador enviar novos dados.

Em nosso caso, queremos chamar o método `emit`, do `WSPrefeitura`:

```
publisher.consume(WSPrefeitura::emit);
```

Para esse caso, nenhum comportamento de como lidar em caso de falhas é definido, mas apenas o que deve ser executado a cada nova nota enviada. O controle de quantas notas podem ser enviadas para evitar sobrecargas no componente fica totalmente escondido também, você não precisa se preocupar com nada disso.

Em outras palavras, você só se preocupa em criar o publicador e o que deve ser feito quando algo for publicado.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Tudo pronto, já temos um publicador e uma ação definida que será executada a cada novo dado enviado. Agora só precisamos criar uma nota fiscal, e enviar com esse nosso publicador.

Isso pode ser feito com uso do método `submit` :

```
NF nf = new NF("Turini", "Livro Java 9", 39.99);
publisher.submit(nf);
```

Vamos testar? De forma parecida com o primeiro exemplo, nosso código agora ficará assim:

```
SubmissionPublisher<NF> publisher = new SubmissionPublisher<>();
publisher.consume(WSPrefeitura::emit);

// venda concluída
System.out.println("Gerando a nota");
NF nf = new NF("Turini", "Livro Java 9", 39.99);
publisher.submit(nf);
// outras ações necessárias
System.out.println("Parabéns pela sua compra");
```

E agora, diferente do exemplo síncrono, o resultado será:

```
Gerando a nota
emitindo...
Parabéns pela sua compra
<<< depois de 5 segundos >>>
emitido!
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Não apareceu a saída?

Se no lugar do JShell você estiver compilando e executando na linha de comando, ou pela IDE, a saída provavelmente não será exibida. Isso acontece porque a execução da aplicação termina antes que a thread secundária, de emissão de notas fiscais, seja executada.

Para resolver você pode travar a execução com um Scanner, por exemplo, pedindo para que o usuário digite algum valor quando quiser terminar a aplicação.

```
System.out.println("Aperte o enter para sair");
new Scanner(System.in).nextLine();
```

Simples, não é? Bastou criar um publicador, informar o que deve ser feito sempre que uma nota fiscal for enviada e usar seu método `submit` para fazer esse envio.

Perceba que, como a execução da API de Flow é assíncrona, a thread principal pode continuar suas tarefas, exibindo a mensagem de sucesso da compra, sem interferência da demora e possíveis falhas do processo de emissão dessa nota fiscal.

Para que o uso de diferentes threads fique bem evidente, podemos imprimir o nome delas ao lado das mensagens do código. O `WSPrefeitura` ficará assim:

```
public class WSPrefeitura {

  public static void emit(NF nf) {
    try {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
      String thread = Thread.currentThread().getName();
      System.out.println("emitindo na thread " + thread);
      Thread.sleep(5000);
      System.out.println("emitido!");
    } catch(Exception e) {
      System.out.println("falha ao emitir a nf");
    }
  }
}
```

E o código de execução principal:

```
SubmissionPublisher<NF> publisher = new SubmissionPublisher<>();
publisher.consume(WSPrefeitura::emit);

// venda concluída
String thread = Thread.currentThread().getName();
System.out.println("thread principal: " + thread);
System.out.println("Gerando a nota");
NF nf = new NF("Turini", "Livro Java 9", 39.99);
publisher.submit(nf);
// outras ações necessárias
System.out.println("Parabéns pela sua compra");

System.out.println("Aperte o enter para sair");
new Scanner(System.in).nextLine();
```

Agora, ao executar novamente, a saída será:

```
thread principal: main
Gerando a nota
emitindo na thread ForkJoinPool.commonPool-worker-1
Parabéns pela sua compra
<<< depois de 5 segundos >>>
emitido!
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

PARA SABER MAIS: FORK/JOIN API

A execução principal aconteceu na *thread* `main`, e a emissão da nota, dentro do `WSPrefeitura`, em uma *thread* chamada `ForkJoinPool.commonPool-worker-1`.

Por padrão, o `SubmissionPublisher` usa a classe `ForkJoinPool`, da API de Fork/Join do Java 7. 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.

Caso não conheça, certamente vai gostar dos exemplos que aparecem logo no início de sua documentação:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

Consumindo dados em diferentes threads

Além dessa execução padrão, com o uso do `ForkJoin`, também podemos definir outros tipos de *executors*, para que o nosso publicador envie as notas fiscais em diferentes threads **para diferentes números de subscribers**.

Para que isso aconteça, precisamos passar como parâmetro um `ExecutorService`, que é uma interface existente na API de `java.util.concurrent`, com objetivo de desacoplar o processo de execução de tarefas em diferentes threads. Podemos usar a

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

classe `java.util.concurrent.Executors` , por exemplo, para criar pools com um número fixo de threads que devem ser utilizadas em uma determinada execução.

```
Executors.newFixedThreadPool(2)
```

Além desse executor, também precisamos passar um número máximo indicando a capacidade de cada subscriber. Não se preocupe tanto com ele agora, vamos discutir mais sobre esse número adiante.

Para criar um publicador com essas configurações, podemos fazer:

```
ExecutorService executorService = Executors.newFixedThreadPool(2)
;
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>(executorService, 1);
```

Neste caso, o `SubmissionPublisher` vai usar um `fixedThreadPool` de duas threads, e estamos avisando que cada `Subscriber` interno criado pelo método `consume` terá a capacidade de atender um elemento por vez. Vamos testar?

Além de criar o *publisher*, dessa forma que vimos, vamos registrar o primeiro `Subscriber` implicitamente a partir do método `consume` , indicando o emissor de notas fiscais exatamente como fizemos antes:

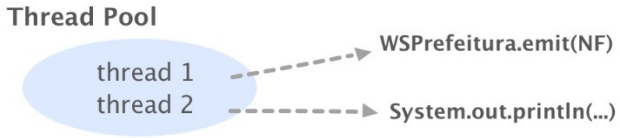
```
publisher.consume(WSPrefeitura::emit);
```

Mas agora também vamos registrar um segundo `Subscriber` , que apenas imprimirá uma mensagem com o nome da *thread* que está sendo executada. Assim, a diferença ficará bem clara.

```
publisher.consume(data -> {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  System.out.println("A outra thread é "
    + Thread.currentThread().getName());
});
```



O código completo ficará assim:

```
ExecutorService executorService = Executors.newFixedThreadPool(2)
;
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>(executorService, 1);

publisher.consume(WSPrefeitura::emit);

publisher.consume(data -> {
  System.out.println("A outra thread é "
    + Thread.currentThread().getName());
});

NF nf = new NF("Turini", "Livro Java 9", 39.99);
publisher.submit(nf);
```

E em meu caso, o resultado da execução foi:

```
emitindo na thread pool-2-thread-1
A outra thread é pool-2-thread-2
emitido!
```

Perceba que foram usadas duas threads diferentes, `thread-1` e `thread-2`, do mesmo `pool-2`. Esses números de pool e threads podem aparecer diferentes em seus testes, claro. O importante aqui é perceber que o trabalho está sendo feito em diferentes threads de um mesmo pool, que foi passado como parâmetro para o publicador.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Adicionando mais threads, seu código vai rodar mais rápido? Não sabemos, e você precisa ter certeza de que o ambiente que está executando esse código tem essa quantidade específica de recursos disponíveis. Muitas vezes o *overhead* de utilizar essa abordagem poderá tornar a execução mais lenta. É necessário tomar cuidado com o uso de *executors*.

Capacidade dos Subscribers

Eu mencionei vagamente que, para criar um publicador com um executor de threads diferente, também precisaríamos indicar a capacidade de cada `Subscriber`. Escolhi o número 1 e disse que isso significava que cada um deles teria a capacidade de atender um elemento por vez.

O que isso realmente significa? Não se preocupe, vamos descobrir.

Podemos começar experimentando criar um publicador com o número zero, ou negativo, dessa capacidade:

```
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>(executorService, 0);
```

O resultado será uma *exception*, indicando que o valor precisa ser positivo.

```
java.lang.IllegalArgumentException thrown:
capacity must be positive
  at SubmissionPublisher.<init>
  (SubmissionPublisher.java:252)
```

Ok, vamos manter 1.

```
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>(executorService, 1);
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Lá no início, quando falei do fluxo reativo, mencionei que, diferente da solução inicial, o componente de destino ia **puxar a quantidade de elementos que ele é capaz de atender**. É justamente isso que esse número indica. Mas como estamos usando o método `consume` do `SubmissionPublisher`, toda a parte de implementação das outras interfaces `Subscriber` e `Subscription` está escondida.

É importante perceber que o método `consume` não está definido na interface `Publisher`, ele só existe na implementação específica do `SubmissionPublisher`. É justamente por isso que não declaramos o tipo do nosso publicador de forma genérica, pela interface.

```
Flow.Publisher<NF> publisher =
  new SubmissionPublisher<>();

publisher.consume(WSPrefeitura::emit);
```

Em outras palavras, o código anterior não compila:

```
jshell>
|  publisher.consume(WSPrefeitura::emit);
|  ^
|  Error:
|  cannot find symbol
|    symbol:   method consume(WSPrefeitura::emit)
|  publisher.consume(WSPrefeitura::emit);
|  ^---------------^
```

Outro detalhe importante é que em nenhum momento definimos o que deve acontecer caso esse processamento falhe.

Para entender melhor todos esses detalhes que estão implícitos nos métodos convenientes do `SubmissionPublisher`, vamos partir para uma outra estratégia, implementando nosso próprio `Subscriber` e manipulando diretamente seus comportamentos.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

5.3 CRIANDO SEU PRÓPRIO SUBSCRIBER

Vimos que o método `consume` do `SubmissionPublisher` abstrai toda a complexidade do `Subscriber` no papel do fluxo reativo. Isso é ótimo, especialmente quando estamos dando os primeiros passos com essa nova API, mas existem situações em que vamos querer definir comportamentos específicos para cada uma das fases desse fluxo.

Para isso, podemos criar a nossa própria implementação da interface `Subscriber`, que possui a seguinte assinatura:

```
public class Flow {
  public static  interface Subscriber<T> {
    void onSubscribe(java.util.concurrent.Flow$Subscription);
    void onNext(T t);
    void onError(java.lang.Throwable);
    void onComplete();
  }
}
```

São apenas 4 métodos e seus comportamentos são bem simples. De forma extremamente resumida: o `onSubscribe` é o ponto de partida. Sempre que um `Publisher` registrar o `Subscriber`, esse método será chamado.

O `onNext` define o que deve ser executado sempre que um publicador enviar alguma informação. Por fim, os métodos `onError` e `onComplete` são executados sempre que um erro acontecer ou o processo terminar, respectivamente.

Não se preocupe se a responsabilidade de algum deles não ficou tão clara, vamos experimentar tudo isso na prática. Por onde começar? Criando a classe `NFSubscriber`.

```
public class NFSubscriber {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
}
```

Agora, para que ela seja um subscriber de verdade, você vai precisar implementar essa interface `Subscriber` e os 4 métodos mencionados. O esqueleto inicial da classe ficará assim:

```
import java.util.concurrent.*;
import static java.util.concurrent.Flow.*;

public class NFSubscriber implements Subscriber<NF> {

  @Override
  public void onSubscribe(Subscription subscription) {
    // ...
  }

  @Override
  public void onNext(NF nf) {
    // ...
  }

  @Override
  public void onError(Throwable t) {
    // ...
  }

  @Override
  public void onComplete() {
    // ...
  }
}
```

Tudo pronto, agora vamos às implementações.

Implementando o onSubscribe

Vimos que sempre que um publicador registrar um novo `Subscriber`, o método `onSubscribe` será chamado. Talvez isso não faça tanto sentido agora, já que antes fazíamos apenas:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>();

publisher.consume(WSPrefeitura::emit);
```

No lugar disso, além do publicador, vamos criar a nossa classe `NFSubscriber`, e de alguma forma precisaremos vincular esses dois. Essa forma é o método `subscribe`:

```
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>();

NFSubscriber subscriber = new NFSubscriber();

publisher.subscribe(subscriber);
```

Nada muito diferente, não é? E curiosamente é a única coisa que vai mudar na hora que formos testar essa nossa implementação! Mas vamos por partes.

Sempre que o `publisher.subscribe(...)` for chamado, aquele primeiro método `onSubscribe` também será. Uma forma simples de implementá-lo seria:

```
@Override
public void onSubscribe(Subscription subscription) {
  this.subscription = subscription;
  subscription.request(1);
}
```

Ele recebe uma `Subscription` como parâmetro, que é basicamente o link entre as duas pontas. Essa interface é responsável por informar ao publicador que seu assinante está preparado para receber novos dados.

Você deve estar se perguntando o que é esse `subscription.request(1)` do final. Ele é justamente aquele número mágico de antes, indicando que o assinante está pronto

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

para receber **1 novo item**, uma nota fiscal para ser emitida.

Implementando o onNext

A cada execução, o método `onNext` será chamado. Ele é o responsável por **definir qual comportamento deve ser executado na classe**. Em nosso caso, de emissão das notas fiscais, vamos chamar o *web service* da prefeitura exatamente como antes:

```
@Override
public void onNext(NF nf) {
  WSPrefeitura.emit(nf);
  subscription.request(1);
}
```

Novamente estamos usando a `subscription.request(1)` para informar ao publicador que, nesse ponto, depois de executar o web service, estamos preparados para receber mais um elemento.

Legal, não é? Ele faz o trabalho e pede mais um. O publicador envia, ele faz o trabalho, e pede outro, e segue assim até que o fluxo acabe. Essa é a implementação prática daquela estratégia de *pull*

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

que comentamos no começo, que livra o assinante do problema de receber mais do que é capaz de atender.

Processando uma quantidade específica

Imagine que, no lugar de notas fiscais, estamos liberando cupons de desconto de forma reativa. Não são cupons infinitos, mas sim 10. Em outras palavras, apenas os 10 primeiros que fizerem uma compra vão ganhar um cupom de desconto para a próxima compra.

Para esses casos, nos quais temos uma quantidade fixa de elementos que podem ser executados, podemos solicitar os itens de uma só vez no método `onSubscribe`, e não solicitar mais nada no `onNext`. A implementação ficaria assim:

```
@Override
public void onSubscribe(Subscription subscription) {
  this.subscription = subscription;
  subscription.request(10);
}

@Override
public void onNext(NF nf) {
  WSPrefeitura.emit(nf);
  // nada aqui... quando passar de 10, acabou.
}
```

Isto é, não é porque a `subscription` está pedindo 10 que o nosso assinante vai receber e tratar 10 itens de uma só vez, em paralelo. Esse número indica apenas **o limite que ele pode atender**.

Não quer definir um limite, nem pedir 1 a cada vez que chamar o `onNext`? Tudo bem, você pode usar o

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

`subscription.request(Long.MAX_VALUE)` para dizer que o `Subscriber` atende um número indefinido de dados.

Implementando onError e onComplete

Os outros dois métodos da interface são ainda mais simples. O `onError` define como o `Subscriber` deve se comportar em caso de alguma *exception* acontecer no meio do processo. Poderíamos enviar um e-mail para a equipe de desenvolvedores, por exemplo, ou ativar qualquer estratégia de *fallback* que faça sentido.

Para o exemplo simples, vamos apenas imprimir a *stacktrace* com a mensagem:

```
@Override
public void onError(Throwable t) {
  t.printStackTrace();
}
```

Seu último método, o `onComplete`, indica o que deve ser executado quando o trabalho acabar. Na prática, isso acontece quando o método `publisher.close()` é executado.

Em nossa classe, imprimiremos uma mensagem avisando que todas as notas já foram emitidas.

```
@Override
public void onComplete() {
  System.out.println(
    "Todas as notas foram emitidas");
}
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Tudo pronto, a implementação está completa. Sei que é muita informação, e talvez isso tudo fique parecendo um monstro de 7 cabeças caso esse seja seu primeiro contato com a abordagem reativa. Apesar disso, se você olhar com calma, o código completo não ficou nada assustador:

```
import java.util.concurrent.*;
import static java.util.concurrent.Flow.*;

public class NFSubscriber implements Subscriber<NF> {

  private Subscription subscription;

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }

  @Override
  public void onNext(NF nf) {
    WSPrefeitura.emit(nf);
    subscription.request(1);
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    System.out.println("Todas as notas foram emitidas");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  }
}
```

Vamos testar?

A única coisa que muda em relação aos testes anteriores é que, dessa vez, vamos criar o `NFSubscriber` e registrá-lo no `Publisher` com o método `subscribe`, no lugar do `consume`.

```
SubmissionPublisher<NF> publisher =
  new SubmissionPublisher<>();

NFSubscriber subscriber = new NFSubscriber();

publisher.subscribe(subscriber);

// venda concluída
String thread = Thread.currentThread().getName();

System.out.println("thread principal: " + thread);
System.out.println("Gerando a nota");

NF nf = new NF("Turini", "Livro Java 9", 39.99);

publisher.submit(nf);

// outras ações necessárias

System.out.println("Parabéns pela sua compra");
publisher.close();

System.out.println("Aperte o enter para sair");
new Scanner(System.in).nextLine();
```

Executando esse fluxo, teremos um output parecido com:

```
thread principal: main
Gerando a nota
emitindo na thread ForkJoinPool.commonPool-worker-1
Parabéns pela sua compra

<<< depois de 5 segundos >>>
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
emitido!
Todas as notas foram emitidas
```

5.4 ENTENDENDO O PROCESSOR

Além das interfaces `Publisher`, `Subscription` e `Subscriber`, que definem esse fluxo que vimos, a API de Flow do JDK 9 também possui uma interface `Processor`. Esta funciona basicamente como uma forma de encadear chamadas de diferentes `Publisher`s para seus `Subscriber`s.

Nós podemos criar vários `Processor`s em um mesmo fluxo:

Na prática, ele pode ser usado para aplicar transformações nos elementos, filtrar ou interromper a execução caso alguma condição não seja atendida. Quer um exemplo?

Considere que, em alguns casos, o valor da compra será zero. Isso pode acontecer quando um cliente utilizar um cupom de 100% de desconto. Nesses casos, não queremos emitir a nota.

Sim, poderíamos resolver com um simples `if` antes de chamar o `publisher.submit(nf)`. Entretanto, o trabalho estaria sendo feito na *thread* principal, de forma síncrona.

É aí que entra a interface `Processor`. Ela pode ficar no meio do fluxo de emissão das notas, funcionando como um filtro de notas elegíveis para emissão, de forma assíncrona.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Uma curiosidade é que ela é apenas um estereótipo. Se você olhar sua definição, verá que um `Processor` nada mais é do que um `Subscriber` e `Publisher` juntos.

```
public class Flow {
  public static interface Processor<T,R>
      extends Subscriber<T>, Publisher<R> {
  }
}
```

Em outras palavras, não há nenhum novo método definido. Isso certamente pode parecer confuso, porém, na prática, não é nada especial. Vamos implementar?

Podemos criar uma classe `NFFilterProcessor` para filtrar as notas com valor inválido.

```
public class NFFilterProcessor {

}
```

E agora, implementar a interface `Processor`, que vai nos obrigar a escrever os mesmos 4 métodos de antes, do `Subscriber`. O esqueleto da classe com esses métodos ficará assim:

```
public class NFFilterProcessor
  extends SubmissionPublisher<NF>
  implements Flow.Processor<NF, NF> {

  @Override
  public void onSubscribe(Subscription subscription) {
    //...
  }

  @Override
  public void onNext(NF nf) {
    //...
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  @Override
  public void onError(Throwable t) {
    //...
  }

  @Override
  public void onComplete() {
    //...
  }
}
```

Perceba que estamos herdando de `SubmissionPublisher` também, para não termos de nos preocupar com todo o trabalho de publicação que já está bem definido nessa classe.

A implementação será basicamente a mesma de antes, do `Subscriber`. A grande diferença estará no método `onNext`, que só vai continuar o fluxo se o valor da nota recebida for válido. Em caso positivo, ele chama o `submit` do `Publisher`; caso contrário, exibe uma mensagem de aviso sem executar nada.

```
@Override
public void onNext(NF nf) {
  if (nf.hasValidAmount()) {
    submit(nf);
  } else {
    System.out.println(
      "Nota com valor menor ou igual a zero"
    );
  }
  subscription.request(1);
}
```

Outra diferença é que, no método `onComplete`, no lugar de imprimir uma mensagem dizendo que as notas foram emitidas, ele vai apenas delegar para o método `close` do publicador, que encadeará esse trabalho em nossa implementação `NFSubscriber`.

```
@Override
public void onComplete() {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  close();
}
```

O código completo do filtro ficará assim:

```
import java.util.concurrent.*;
import static java.util.concurrent.Flow.*;

public class NFFilterProcessor
  extends SubmissionPublisher<NF>
  implements Processor<NF, NF> {

  private Subscription subscription;

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }

  @Override
  public void onNext(NF nf) {
    if (nf.hasValidAmount()) {
      submit(nf);
    } else {
      System.out.println(
        "Nota com valor menor ou igual a zero");
    }
    subscription.request(1);
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    close();
  }
}
```

Nada muito diferente do que já vimos, mas a grande diferença será no momento de encadear as execuções. Dessa vez, para

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

executar o exemplo, vamos criar um publicador, um *subscriber* e um *processor*.

O publicador vai registrar o *processor*, que vai, por sua vez, registrar o *subscriber*. Essa ordem é importante para o funcionamento da nossa lógica.

```
SubmissionPublisher<NF> publisher = new SubmissionPublisher<>();
NFSubscriber subscriber = new NFSubscriber();
NFFilterProcessor filter = new NFFilterProcessor();

publisher.subscribe(filter);
filter.subscribe(subscriber);
```

O restante do código será o mesmo. Para testar o nosso filtro, além da nota válida, vamos criar uma com o valor zero.

```
SubmissionPublisher<NF> publisher = new SubmissionPublisher<>();
NFSubscriber subscriber = new NFSubscriber();
NFFilterProcessor filter = new NFFilterProcessor();

publisher.subscribe(filter);
filter.subscribe(subscriber);

// venda concluída
String thread = Thread.currentThread().getName();
System.out.println("thread principal: " + thread);
System.out.println("Gerando a nota");

NF nf = new NF("Turini", "Livro Java 9", 39.99);

publisher.submit(nf);

NF nf2 = new NF("Turini", "Livro Java 9", 0);

publisher.submit(nf2);

// outras ações necessárias
System.out.println("Parabéns pela sua compra");
publisher.close();

System.out.println("Aperte o enter para sair");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
new Scanner(System.in).nextLine();
```

Ao executar, o filtro será aplicado e a saída será:

```
thread principal: main
Gerando a nota
emitindo na thread ForkJoinPool.commonPool-worker-1
Parabéns pela sua compra
Nota com valor menor ou igual a zero

<<< depois de 5 segundos >>>

emitida!
Todas as notas foram emitidas
```

Os `Processor` s também são muito úteis para aplicar transformações nos itens que estão sendo enviados, com a vantagem de serem executados dentro do fluxo assíncrono, sem interferir no restante do trabalho.

5.5 DOWNLOAD DOS EXEMPLOS DESTE CAPÍTULO

Todos os exemplos de código vistos aqui estão disponíveis no diretório deste capítulo, no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 6

JUNTANDO AS PEÇAS

Vimos algumas atualizações e novas APIs do Java 9, mas, até agora, executamos todos os exemplos diretamente pelo ambiente interativo do JShell — ou, para quem preferiu, em classes de testes com o tradicional método `main` .

Agora que já conhecemos bem e tivemos bastante oportunidade de praticar essa ferramenta de *REPL*, nosso objetivo será criar uma aplicação simples que listará alguns livros, consumidos pelo *HTTP/2 Client* e, ao escolher um deles, disparará o processo de nota fiscal criado com a API de *Reactive Streams*. Em outras palavras, vamos juntar todas as peças que vimos nos capítulos anteriores do livro dentro de um mesmo projeto!

Essa será a oportunidade perfeita para relembrar a estrutura de um projeto tradicional, além dos processos de compilação e execução de código. Assim, poderemos compará-lo com a nova abordagem do sistema de módulos, Jigsaw, e entender a importância e funcionamento dessa modularização de código e do próprio JDK.

É importante perceber que o foco aqui não é a criação do projeto em si, ou mesmo a implementação desse código. Nosso objetivo será praticar e relembrar a anatomia de um projeto

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

tradicional em Java, conceitos como *classpath* e o processo de compilação em si.

6.1 CRIANDO UM NOVO PROJETO

Para começar, vamos criar um projeto da forma tradicional, que em breve deve servir como parâmetro de comparação com a nova versão modular. Esse também vai ser um bom ponto de partida para você saber na prática qual será o esforço inicial de migrar os seus projetos existentes, caso queira tirar proveito desse novo recurso.

O projeto vai se chamar `bookstore` e as classes devem ficar dentro do pacote principal `br.com.casadocodigo`. Você pode criar a estrutura do projeto pela linha de comando pelo seu terminal, ou no gerenciador de arquivos de seu sistema operacional favorito. O importante é que, ao final, a estrutura dos diretórios fique assim:

```
└── bookstore
    └── src
        └── br
            └── com
                └── casadocodigo
```

DICA

Para criar pela linha de comando, em seu terminal, você pode usar o `mkdir` com o parâmetro `-p` conforme a seguir.

```
mkdir -p bookstore/src/br/com/casadocodigo/
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Agora que já temos os diretórios organizados, vamos criar uma classe `Main.java` dentro de `src/br/com/casadocodigo/` . Por enquanto, ela deve imprimir apenas uma mensagem de título da lista de livros que temos disponíveis:

```
package br.com.casadocodigo;

public class Main {

  public static void main(String ...args) {

    System.out.println("\nLista de livros disponíveis \n");
  }
}
```

> A essa altura, as principais *IDEs* como *IntelliJ*, *NetBeans* e *Eclipse* já dão suporte ao JDK 9 e até mesmo à criação de projetos modulares. Caso prefira, você pode fazer diretamente por uma dessas *IDEs* de sua preferência.
>
> Eu particularmente gosto e recomendo que você experimente um pouco na linha de comando, pois isso incentiva a entender e relembrar a fundo o funcionamento desse processo importante de compilação e execução. **Veremos que, em projetos modulares, ele será um pouco diferente dessa forma com que estamos acostumados.**

Para compilar, sem uso de *IDEs*, você pode rodar o seguinte comando `javac` a partir da raiz do projeto:

```
javac src/br/com/casadocodigo/Main.java
```

E o `java` , para executar:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
java -cp src/ br.com.casadocodigo.Main
```

Experimente executar os comandos. O resultado, conforme esperado, deve ser o texto que estamos imprimindo no método `main` dessa primeira classe:

```
Lista de livros disponíveis
```

Demos o primeiro passo! Nosso projeto já está ganhando formato.

Perceba que estamos usando o parâmetro `-cp`, apontando para o **classpath** do projeto, que é a pasta `src`. **Esse é um conceito-chave para entender uma das maiores motivações do sistema de módulos e da própria modularização da plataforma**, que será extensivamente estudada no próximo capítulo.

Quando você executa um programa em Java, a máquina virtual (*JVM*) procura pelas suas classes compiladas, e carrega esse *bytecode* conforme for precisando. O *classpath* é a forma de dizer para as aplicações, incluindo as ferramentas do JDK, onde devem procurar em nosso sistema de arquivos para encontrar a definição, o *bytecode*, dessas classes.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Talvez você já tenha ouvido falar em *classpath hell*, ou *JAR hell*, que são termos muito comuns para definir um grande problema nesse mecanismo de carregamento de classes do Java. Outra frase que ficou muito famosa foi a *"The classpath is dead"*, do Mark Reinhold, que é engenheiro-chefe da plataforma Java na Oracle.

O projeto que estamos construindo vai nos ajudar a entender esse problema na prática, assim como a solução e funcionamento do *module path*, que é seu substituto.

Talvez você não esteja tão acostumado com esse processo todo, afinal, a maioria de nós não compila e executa classes na linha de comando no dia a dia, e nem deveria! As IDEs abstraem boa parte desse processo e das preocupações do trabalho, mas é fundamental entendê-las.

6.2 LISTANDO LIVROS COM HTTP/2 CLIENT

Nosso projeto por enquanto só exibe um texto, mas queremos que ele liste todos os livros utilizando aquele arquivo *CSV*, do capítulo de *HTTP/2 Client*. Para isso, vamos precisar do enum `Category`, que pode ser colocado dentro do pacote `br.com.casadocodigo.model`:

```
package br.com.casadocodigo.model;

public enum Category {

  PROGRAMMING,
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  DESIGN,
  AGILE,
  CERTIFICATION,
  BUSINESS
}
```

E da classe `Book.java`, exatamente da forma que vimos anteriormente:

```
package br.com.casadocodigo.model;

import java.util.*;

public class Book {

  private final String name;
  private final String author;
  private final List<Category> categories;

  // outros atributos

  public Book(String name, String author, Category ...categories)
{
      this.name = name;
      this.author = author;
      this.categories = List.of(categories);
  }

  public String getAuthor() {
      return author;
  }

  public String getName() {
      return name;
  }

  public List<Category> getCategories() {
      return categories;
  }

  @Override
  public String toString() {
      return "\nlivro: " + name
          +  "\nautor: " + author
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
          +  "\ncategorias: " + categories;
  }
}
```

Agora que já temos a representação de livros e suas categorias, podemos criar a classe `Books.java`, no pacote `br.com.casadocodigo.data`, que faz o consumo do *endpoint* com alguns livros em formato *CSV*. O código do arquivo, já com a declaração de pacote e `import` s, ficará assim:

```
package br.com.casadocodigo.data;

import br.com.casadocodigo.model.*;
import java.util.*;
import java.util.stream.*;
import java.net.URI;
import jdk.incubator.http.*;

public class Books {

  public static List<Book> all() {
    try {
      String csv = HttpClient.newHttpClient()
      .send(HttpRequest.newBuilder()
        .uri(new URI("https://turini.github.io/livro-java-9/books
.csv"))
        .GET().build(),
      HttpResponse.BodyHandler.asString()).body();

      return Stream.of(csv.split("\n"))
        .map(Books::create)
        .collect(Collectors.toList());

    } catch (Exception e) {
      throw new RuntimeException("Não foi possível conectar ", e)
;
    }
  }

  public static Book create(String line) {
    String[] split = line.split(",");
    String name = split[0];
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    String author = split[2];
    Category category = Category.valueOf(split[3]);
    return new Book(name, author, category);
  }

  public static Optional<Book> findSimilar(Book book) {
    return Books.all().stream()
      .filter(b -> b.getCategories().equals(book.getCategories())
)
      .filter(b -> !b.getAuthor().equals(book.getAuthor()))
      .findAny();
  }
}
```

Com essas adições, que já vimos aqui no livro, a estrutura completa de nosso projeto com todos seus pacotes e classes será:

```
└── bookstore
    └── src
        └── br
            └── com
                └── casadocodigo
                    ├── Main.java
                    ├── data
                    │   └── Books.java
                    └── model
                        ├── Book.java
                        └── Category.java
```

Tudo pronto, podemos listar os livros em nossa classe `Main`. Para facilitar, vamos adicionar o index do livro ao lado de seu nome, como se fosse um ID. Assim, o usuário poderá usar esse mesmo número no momento de escolher um dos livros.

Para resolver de uma forma simples, podemos criar um range de inteiros do tamanho dessa lista de livros, retornada pelo serviço web, imprimindo seu *index* e o nome do livro nessa mesma posição da lista.

```
List<Book> books = Books.all();
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
IntStream.range(0, books.size())
  .forEach(i -> {
    System.out.println(i + " - " + books.get(i).getName());
  });
```

> Lembre-se de que, como o código está consumindo um *endpoint* web, você precisará estar conectado à internet. Caso esteja offline, o resultado será um inevitável `java.net.ConnectException thrown: Operation timed out` . Como alternativa, para continuar offline, você pode criar alguns livros e retornar apenas essa lista.

Nossa classe `Main.java` , com esse código de listagem dos livros, agora está assim:

```
package br.com.casadocodigo;

import java.util.*;
import java.util.stream.*;
import br.com.casadocodigo.data.*;
import br.com.casadocodigo.model.*;

public class Main {

  public static void main(String ...args) {

    System.out.println("\nLista de livros disponíveis \n");

    List<Book> books = Books.all();

    IntStream.range(0, books.size())
      .forEach(i -> {
        System.out.println(i + " - " + books.get(i).getName());
      });
    }
}
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Se estiver fazendo pela linha de comando, pode compilar todos os arquivos como a seguir:

```
javac src/br/com/casadocodigo/Main.java \
  src/br/com/casadocodigo/model/Category.java \
  src/br/com/casadocodigo/model/Book.java \
  src/br/com/casadocodigo/data/Books.java
```

E o resultado, talvez inesperado, será um erro de compilação em `Books.java` ! Ele diz que o pacote do HTTP/2 Client não é visível.

```
src/br/com/casadocodigo/data/Books.java:7:
error: package jdk.incubator.http is not visible
import jdk.incubator.http.*;
                    ^
```

E faz sentido. Ele está naquele módulo de incubação, que por padrão não fica disponível no JDK, lembra?

Como já vimos, para usá-lo, precisaremos declarar explicitamente com o parâmetro `--add-modules` .

```
javac --add-modules jdk.incubator.httpclient \
  src/br/com/casadocodigo/Main.java \
  src/br/com/casadocodigo/model/Category.java \
  src/br/com/casadocodigo/model/Book.java \
  src/br/com/casadocodigo/data/Books.java
```

Agora sim, ao compilar, apenas um *warning* será exibido com aviso de que o módulo `jdk.incubator.httpclient` está em incubação.

```
warning: using incubating module(s): jdk.incubator.httpclient
1 warning
```

Vamos executar novamente? Lembre-se de passar esse parâmetro do módulo `httpclient` , assim como fizemos durante a compilação:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
java --add-modules jdk.incubator.httpclient \
    -cp src \
    br.com.casadocodigo.Main
```

O resultado será:

```
Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
3 - Introdução e boas práticas em UX Design
4 - Conhecendo o Adobe Photoshop CS6

<<< outros livros da lista >>>
```

Pronto, já temos a parte HTTP de nosso projeto! O próximo passo será permitir que o leitor escolha um livro, para disparar o processo assíncrono de emissão da nota fiscal, visto no capítulo anterior.

6.3 INTEGRANDO A EMISSÃO DE NOTAS FISCAIS

Precisaremos das classes `NF` e `WSPrefeitura`, que criamos no capítulo anterior. A classe `NF.java` pode ficar dentro do pacote `br.com.casadocodigo.model`, junto com as outras representações de modelo de negócio do nosso projeto.

```
package br.com.casadocodigo.model;

public class NF {

  private String client;
  private String book;
  private double amount;

  public NF(String client, String book, double amount) {
    this.client = client;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    this.book = book;
    this.amount = amount;
  }

  public boolean hasValidAmount() {
    return amount > 0;
  }
}
```

Aquele `WSPrefeitura`, que tem um delay de 5 segundos para simular o processo de comunicação com outro serviço da prefeitura, ficará dentro de um novo pacote, chamado `br.com.casadocodigo.service`:

```
package br.com.casadocodigo.service;

import br.com.casadocodigo.model.*;

public class WSPrefeitura {

  public static void emit(NF nf) {
    try {
      System.out.println("emitindo...");
      Thread.sleep(5000);
      System.out.println("emitido!");
    } catch(Exception e) {
      System.out.println("falha ao emitir a nf");
    }
  }
}
```

E também vamos precisar do `NFSubscriber.java`, que é nossa implementação de `Flow.Subscriber`, para definir todo o comportamento que deve ser executado quando uma nova nota fiscal é publicada em nosso sistema.

```
package br.com.casadocodigo.service;

import java.util.concurrent.*;
import java.util.concurrent.Flow.*;
import br.com.casadocodigo.model.*;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
public class NFSubscriber implements Subscriber<NF> {

  private Subscription subscription;

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }

  @Override
  public void onNext(NF nf) {
    WSPrefeitura.emit(nf);
    subscription.request(1);
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    System.out.println(
        "Todas as notas foram emitidas");
  }
}
```

Como última etapa, para organizar um pouco mais a estrutura do projeto, vamos encapsular todo o código do publicador da *Flow API* dentro de uma nova classe chamada `NFEmissor`. Essa classe vai ser nosso link de comunicação com o processo de emissão das notas.

Sempre que criarmos um emissor desse tipo, um novo `Flow.Publisher` deverá ser criado, já registrando aquela nossa implementação de `Flow.Subscriber`. Isso pode ser feito em seu construtor.

```
public class NFEmissor {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  private SubmissionPublisher<NF> publisher;

  public NFEmissor() {
    this.publisher = new SubmissionPublisher<>();
    publisher.subscribe(new NFSubscriber());
  }
}
```

Um método `emit` de `NFEmissor` vai receber o nome do cliente e o livro escolhido, que são todas as informações necessárias para construção e publicação da nota. O valor pode ser fixo, não queremos nos preocupar com isso nesse momento.

```
public void emit(String clientName, Book book) {
  NF nf = new NF(clientName, book.getName(), 39.99);
  publisher.submit(nf);
}
```

E por fim, ainda em `NFEmissor`, um método `close`, que será apenas um delegate para o método `close` do `SubmissionPublisher`. Assim, podemos definir o final do processo explicitamente, que resultará na chamada do método `onComplete` de nosso `NFSubscriber`.

```
public void close() {
  this.publisher.close();
}
```

A classe `NFEmissor` completa ficará assim:

```
package br.com.casadocodigo.service;

import java.util.concurrent.*;
import java.util.concurrent.Flow.*;
import br.com.casadocodigo.model.*;

public class NFEmissor {

  private SubmissionPublisher<NF> publisher;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  public NFEmissor() {
    this.publisher = new SubmissionPublisher<>();
    publisher.subscribe(new NFSubscriber());
  }

  public void emit(String clientName, Book book) {
    NF nf = new NF(clientName, book.getName(), 39.99);
    publisher.submit(nf);
  }

  public void close() {
    this.publisher.close();
  }
}
```

Pronto, nosso projeto já tem todo o código necessário para a publicação das notas. Vamos ver se compila?

Podemos agora executar o comando `javac` para todas as classes dos três pacotes existentes:

```
javac --add-modules jdk.incubator.httpclient \
  src/br/com/casadocodigo/Main.java \
  src/br/com/casadocodigo/model/*.java \
  src/br/com/casadocodigo/data/*.java \
  src/br/com/casadocodigo/service/*.java
```

Uma outra forma interessante de compilar todas as classes do projeto na linha de comando, sem precisar ficar se preocupando em adicionar os novos pacotes ou esquecer algo para trás, seria compondo o comando do `javac` com uma busca que lista todos os arquivos que terminam com a extensão `.java`. Em ambientes *UNIX*, você pode fazer isso assim:

```
javac --add-modules jdk.incubator.httpclient \
  $(find . -name "*.java")
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

6.4 INTERAGINDO COM A LISTA DE LIVROS

Queremos agora deixar nosso código interativo, para que o usuário possa escolher um dos livros e encadear todo esse processo. Ao final, além da mensagem de sucesso, podemos sugerir um livro similar, utilizando o método `findSimilar` que criamos anteriormente na classe `Books`.

Vamos começar pedindo para o usuário digitar o número do livro. Um `Scanner` simples, com a entrada padrão do terminal, pode ser usado para ler o valor que será digitado.

```
Scanner scanner = new Scanner(System.in);

System.out.println(
    "\nDigite o número do livro que quer comprar: \n");

int number = scanner.nextInt();
```

Agora que temos esse número pronto, vamos buscar o livro que está nessa posição da lista:

```
Book book = books.get(number);
```

E exibir a escolha no terminal.

```
System.out.println(
    "O livro escolhido foi: "
    + book.getName()
);
```

Também vamos precisar do nome do leitor, para emissão da nota fiscal. Podemos pedir de forma parecida, também com uso do `Scanner`.

```
System.out.println(
    "Informe seu nome, para
que possamos emitir a nota fiscal");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
String name = scanner.nextLine();
```

Tendo essas duas informações prontas, só precisamos criar uma instância daquele nosso `NFEmissor` passando os valores para seu método `emit`, que executará o publicador, dando início ao fluxo reativo, assíncrono, de emissão das notas.

```
NFEmissor emissor = new NFEmissor();
emissor.emit(name, book);
```

A última etapa será sugerir um livro similar, caso ele exista. Vamos fazer isso com o novo `ifPresentOrElse`, da API de `Optional`.

```
Books.findSimilar(book)
  .ifPresentOrElse(
  similar -> System.out.println(
    "\nTalvez você também goste do livro: " + similar.getName()),
  () -> System.out.println(
    "\nNão temos nenhuma sugestão de livro similar no momento ")
  );
```

Confesso que, se não tomarmos cuidado com esse novo método do `Optional`, nosso código pode ficar bastante confuso e pouco legível. Uma forma de melhorar seria extrair essas duas expressões que ele recebe como parâmetro. A chamada ficará assim:

```
Books.findSimilar(book)
  .ifPresentOrElse(showSimilar, noSuggestions);
```

E as expressões:

```
private static Consumer<Book> showSimilar = similar -> {
  System.out.println(
    "\nTalvez você também goste do livro: "
    + similar.getName());
};
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
private static Runnable noSuggestions = () -> {
  System.out.println(
    "\nNão temos nenhuma sugestão de
    livro similar no momento");
};
```

Outro detalhe importante é que, para evitar erros de conversão do valor digitado pelo usuário, precisamos envolver o código em um `try/catch`. O código completo da nossa classe `Main`, com todas essas alterações, deve ficar assim:

```
package br.com.casadocodigo;

import br.com.casadocodigo.data.*;
import br.com.casadocodigo.model.*;
import br.com.casadocodigo.service.*;
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

public class Main {

  public static void main(String ...args) {

    System.out.println("\nLista de livros disponíveis \n");

    List<Book> books = Books.all();

    IntStream.range(0, books.size())
      .forEach(i -> {
        System.out.println(i + " - " + books.get(i).getName());
      });

    Scanner scanner = new Scanner(System.in);

    System.out.print("\nDigite o número do livro que quer comprar
: ");

    try {
      int number = scanner.nextInt();
      Book book = books.get(number);

      System.out.println("\nO livro escolhido foi: "
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
            + book.getName());

        System.out.print("\nInforme seu nome,
            para que possamos emitir a nota fiscal: ");

        scanner = new Scanner(System.in);
        String name = scanner.nextLine();
        NFEmissor emissor = new NFEmissor();
        emissor.emit(name, book);

        System.out.println("Obrigado!");

        Books.findSimilar(book)
          .ifPresentOrElse(showSimilar, noSuggestions);

        // segura a execução para o código assíncrono terminar
        System.out.println("Aperte o enter para sair");
        new Scanner(System.in).nextLine();

        emissor.close();

      } catch(Exception e) {
        System.err.println("Ops, aconteceu um erro: "+ e);
      }
    }

    private static Consumer<Book> showSimilar = similar -> {
      System.out.println(
        "\nTalvez você também goste do livro: "
        + similar.getName());
    };

    private static Runnable noSuggestions = () -> {
      System.out.println(
        "\nNão temos nenhuma sugestão de livro similar no momento")
;
    };
}
```

Perceba que usamos um `Scanner` para que a execução do código não termine antes da execução assíncrona de emissão da nota.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Podemos testar. Basta compilar o código novamente e executar. A lista de livros deve ser carregada e, assim como esperamos, a aplicação vai nos pedir para escolher um deles e informar o nome que deve ser usado na nota fiscal. Um exemplo de iteração seria:

```
Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
3 - Introdução e boas práticas em UX Design
4 - Conhecendo o Adobe Photoshop CS6
5 - Edição e organização de fotos com Adobe Lightroom
6 - Métricas Ágeis
7 - Scrum: Gestão ágil para projetos de sucesso
8 - Test-Driven Development
9 - Direto ao Ponto
10 - Guia da Startup
11 - O Mantra da Produtividade
12 - Java SE 7 Programmer I
13 - Java SE 8 Programmer I
14 - Zend Certified Engineer

Digite o número do livro que quer comprar: 3

O livro escolhido foi:
Introdução e boas práticas em UX Design

Informe seu nome, para que possamos emitir a nota fiscal:
Rodrigo Turini

Obrigado!
emitindo...

Talvez você também goste do livro:
Conhecendo o Adobe Photoshop CS6

emitido!
Todas as notas foram emitidas
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

6.5 REVENDO O PROJETO E RESPONSABILIDADES

Apesar de simples, já temos uma quantidade suficiente de código e APIs sendo usados. Ao final, nosso projeto ficou com a seguinte estrutura:

```
└── bookstore
    └── src
        └── br
            └── com
                └── casadocodigo
                    ├── Main.java
                    ├── data
                    │   └── Books.java
                    ├── model
                    │   ├── Book.java
                    │   ├── Category.java
                    │   └── NF.java
                    └── service
                        ├── NFEmissor.java
                        ├── NFSubscriber.java
                        └── WSPrefeitura.java
```

Temos as classes `Book` e `Category`, representando livros e suas categorias. A classe `Books` fica com a responsabilidade de acessar os dados do *endpoint* web e projetá-los para cada um desses tipos. O pacote `br.com.casadocodigo.service` inteiro fica com a responsabilidade de emissão de notas fiscais, tirando proveito da nova *Flow API*.

É uma estrutura e código extremamente simples, claro. E sim, poderíamos ter modelado as classes e escrito esse mesmo código de mil outras maneiras. O objetivo aqui não é a construção de um e-commerce, ou solução completa de vendas, mas chegar a um código suficiente, e que envolva um pouco de cada uma das novidades que vimos até agora, para a migração para o Jigsaw.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

A grande verdade é que **nós já estamos usando o sistema de módulos do JDK 9, ainda que involuntariamente.**

Você não é obrigado a declarar e criar projetos modulares ao atualizar para o Java 9, nem mesmo entender ou conhecer esse conceito — apesar de ser muito recomendado. Para permitir isso e pela flexibilidade de migração ou atualização da versão que você está usando da plataforma, todo projeto criado sem a definição de módulos fica em um módulo especial chamado **unnamed module**.

Ele lê e tem acesso a todos os outros módulos, isto é, a todas as classes e APIs públicas da plataforma, da mesma forma que os projetos de antes dessa grande mudança. Esse e outros detalhes, assim como a motivação, funcionamento e melhores práticas dessa novidade do Java, serão estudados e colocados em prática exaustivamente a partir de agora.

Pronto para modularizar?

6.6 DOWNLOAD DO PROJETO COMPLETO

Se quiser, você pode baixar o projeto completo que está disponível na pasta deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 7

JAVA MODULAR

A palavra módulo e o sistema de módulos em si, *Jigsaw*, foram mencionados diversas vezes aqui no livro. Mas, afinal, o que é um módulo? Quando criar um e como deve ser feita a divisão de módulos em uma aplicação? Por que eu deveria fazer isso em meus projetos?

Chegou o momento em que vamos entender todas essas perguntas e encontrar, de forma prática, suas respostas. A vantagem, as características e todos os detalhes fundamentais desse novo conceito e organização da plataforma serão o foco deste novo capítulo.

Por onde começar? Pelo problema da abordagem atual e pela famosa frase *"the classpath is dead"*.

7.1 ANTES DO JAVA 9

Durante a criação de nossa `bookstore`, mencionamos o *classpath*, usado pelo ambiente de execução do Java para encontrar as classes compiladas de seu projeto, assim como de todas as suas dependências externas. Mesmo no caso de nosso projeto, que é muito pequeno, o *classpath* teria um monte de informações, como:

```
loaded java.lang.Object
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
loaded java.io.Serializable
loaded java.lang.CharSequence
loaded java.lang.String
loaded java.lang.Throwable
...
loaded br.com.casadocodigo.Main
loaded br.com.casadocodigo.model.Book
loaded br.com.casadocodigo.model.Category
loaded br.com.casadocodigo.data.Books
...
```

Claro que, em projetos maiores, com bibliotecas e dependências externas, o resultado será ainda maior. Certamente precisaremos usar algo como o *hibernate* para persistir os dados, e *log4j*, por exemplo, para registrar os logs da aplicação.

```
loaded org.hibernate.Session
loaded org.hibernate.query.Query
loaded org.hibernate.transform.Transformers
...
loaded org.slf4j.Logger
loaded org.slf4j.LoggerFactory
...
```

Perceba que, nesse ponto, **já não existe mais a organização dos JARs ou qualquer tipo de agrupamento**. Todas as classes são listadas sequencialmente. Quando a JVM procura por uma classe específica, isso é feito nessa mesma sequência, definida pelo *classpath*. Ao encontrar a primeira — nem sempre única —, a busca termina.

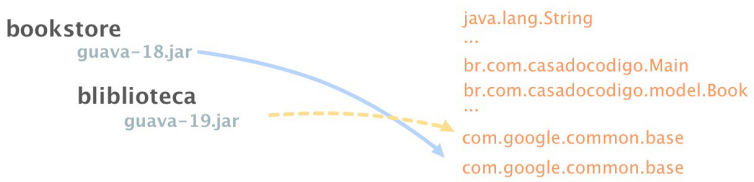
O que pode dar errado? **Um monte de coisas**.

A máquina virtual procura pelas classes dinamicamente, em tempo de execução e de forma preguiçosa — carregando os recursos em memória apenas quando, e se, precisar. Essa estratégia é ótima por uma série de motivos, mas, por ocorrer em tempo de execução, o que aconteceria se a JVM não encontrasse algum

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

recurso lá no *classpath*? Ela iria falhar enquanto a aplicação estivesse rodando, e seria tarde demais. Esse é um cenário bem diferente do ideal, que seria descobrir essa falha muito antes da execução.

Outro erro clássico é quando existem duplicações de classes no *classpath*. Isso pode acontecer quando há algum conflito de versões nas dependências de seu projeto, por exemplo, quando você usa a versão X do *Guava* e alguma outra dependência traz a versão Y junto com ela.



As duas versões diferentes vão estar no **classpath**. Qual a máquina virtual vai escolher? A que aparecer primeiro. É essa que seu código espera e precisa? Muitas vezes, não. Novamente, só vamos descobrir isso quando for tarde demais e a aplicação estiver rodando.

Esses são alguns dos motivos que deram origem ao termo *classpath-hell*, ou *jar-hell*. Esse funcionamento é extremamente frágil, já que não temos como ficar sabendo e não conseguimos prevenir erros antes de tudo estar pronto e a aplicação estar no ar. Esse grande potencial de falhas nos traz a famosa frase "the *classpath* is dead", do Mark Reinhold, engenheiro chefe da plataforma Java na Oracle e um dos grandes responsáveis pelos avanços enormes das últimas versões do JDK.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

A solução? Uma forma de estruturar melhor aplicações, e o próprio código da plataforma Java.

7.2 QUE VENHA O MODULAR

Há muito tempo se diz sobre modularizar a plataforma Java e, diferente de todos os meus outros livros, neste eu finalmente posso dizer que isso aconteceu. Com muitas revisões, polêmicas e turbulências, o Java 9 introduz o *Java Platform Module System* (JPMS), conhecido popularmente como projeto Jigsaw.

Se você nunca teve contato com o conceito, com outras bibliotecas de modularização (como o OSGi) ou com os *early accesses* do próprio Jigsaw, certamente está se perguntando: **afinal, o que é um módulo?**

Ele nada mais é do que um grupo de pacotes e recursos (como imagens, arquivos de configuração, XML etc.), relacionados de alguma forma. No código de nosso projeto, por exemplo, poderíamos agrupar toda a lógica de emissão das notas fiscais em um módulo de notas fiscais. Todos os pacotes, interfaces e dependências externas necessárias para esse trabalho estariam agrupados.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Claro, essa é apenas uma das diferentes formas de estruturar um projeto em módulos, como veremos em detalhes adiante.

Uma das grandes vantagens dessa nova estrutura e possibilidade da linguagem é que, diferente da abordagem atual do *classpath*, as informações do módulo **precisam ficar disponíveis da mesma forma durante as fases de compilação e execução**. Isso garante uma integridade muito maior nos projetos, evitando muitos problemas da abordagem atual do legado *classpath* ou, pelo menos, reportando-os muito antes, em tempo de compilação.

Além disso, a estrutura traz diversas vantagens de performance, segurança e especialmente um **encapsulamento mais forte em nossos projetos**. Com ela, podemos ter implementações públicas, porém **visíveis apenas por dentro do nosso módulo ou partes especificamente selecionadas de nosso projeto**.

Isso resolve um problema muito grande de quem implementa bibliotecas, APIs, ou mesmo dos próprios arquitetos do JDK, que não conseguiam esconder totalmente os detalhes internos de implementações de suas APIs públicas — **o `sun.misc.Unsafe` está aí até hoje para nos provar isso.**

Claro que módulos são bem mais do que um grupo de pacotes, e existem muitas outras vantagens além das aqui brevemente mencionadas. Mas no lugar de um monte de teorias, isso tudo será visto na prática e extensivamente testado daqui para a frente.

Vamos ao código?

7.3 MODULARIZANDO A BOOKSTORE

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

No capítulo anterior, juntamos todas as peças soltas que havíamos construído ao longo dos primeiros capítulos em um mesmo projeto, que chamamos de `bookstore`. Nele já podemos listar livros que são carregados dinamicamente via *endpoint web*, com o uso do *HTTP/2 Client*, e simular o processo de emissão de notas fiscais de forma assíncrona e reativa, com uso da *Flow API*.

O projeto terminou com a seguinte estrutura:

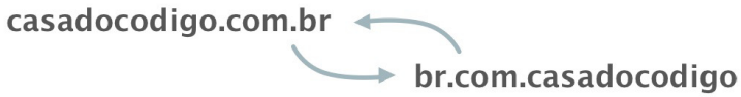
```
└── bookstore
    └── src
        └── br
            └── com
                └── casadocodigo
                    ├── Main.java
                    ├── data
                    │   └── Books.java
                    ├── model
                    │   ├── Book.java
                    │   ├── Category.java
                    │   └── NF.java
                    └── service
                        ├── NFEmissor.java
                        ├── NFSubscriber.java
                        └── WSPrefeitura.java
```

Queremos agora modularizar nossa `bookstore`, transformando todas essas pequenas *peças* em diferentes módulos do mesmo projeto. Como ponto de partida, antes mesmo de entrar na discussão sobre como dividir projetos em diferentes módulos, **precisamos entender a estrutura de um projeto desse tipo e o que muda nos processos de compilação e execução.** Por onde começar?

Todo módulo precisa de um nome, uma forma única de identificá-lo. Esse nosso módulo principal é onde ficará a classe `Main.java` e, por enquanto, todo o restante do projeto pode se

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

chamar `br.com.casadocodigo`. Perceba que, assim como em nossos pacotes, é uma convenção nomeá-los com o domínio reverso da aplicação. O famoso *reverse-domain-name pattern*.



Agora que já temos isso definido, a migração para torná-lo explicitamente modular não é nada complicada. O primeiro passo será criar um novo diretório, dentro do source do projeto, com o nome do módulo.

Portanto, criaremos o diretório `br.com.casadocodigo` e moveremos todos os pacotes e classes para dentro dele. A estrutura do projeto ficará assim:

```
└── bookstore
    └── src
        └── br.com.casadocodigo
            └── br
                └── com
                    └── casadocodigo
                        ├── Main.java
                          ...
```

Todo módulo também precisa de um arquivo especial chamado `module-info.java` que, como o próprio nome indica, vai definir todas suas informações e configurações. A estrutura inicial desse arquivo também é bem simples.

```
module br.com.casadocodigo {

}
```

Nesse primeiro momento, você não precisa configurar nada,

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

além dessa definição inicial que fizemos. Já existe uma série de configurações implícitas na definição de um módulo, como vamos discutir adiante.

Outro detalhe é que `module` agora é uma palavra reservada da linguagem, da mesma forma que `class`, `enum`, `interface` e outras. Ele não precisa de modificadores de visibilidade, como `public`, assim como as classes, mas possui alguns modificadores para definir como será acessível, além de outras configurações que vamos descobrir ao longo do texto.

Nossa `bookstore` agora tem a seguinte estrutura:

```
└── bookstore
    └── src
        └── br.com.casadocodigo
            ├── br
            │   └── com
            │       └── casadocodigo
            │           ├── Main.java
            │           ├── data
            │           │   └── Books.java
            │           ├── model
            │           │   ├── Book.java
            │           │   ├── Category.java
            │           │   └── NF.java
            │           └── service
            │               ├── NFEmissor.java
            │               ├── NFSubscriber.java
            │               └── WSPrefeitura.java
            └── module-info.java
```

Pronto, nosso projeto já é **explicitamente** modular.

Nada complicado, não é? Bastou criar um diretório e arquivo a mais.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Relembrando as convenções

Vimos aqui que todo módulo precisa de um nome e que, por convenção, é recomendado o uso do *reverse-domain-name*, assim como fazemos há anos nos pacotes. Outra convenção é do nome do arquivo de definição do módulo, o `module-info`, e de sua localização.

Nenhuma dessas são obrigações, portanto você pode nomear seus módulos de forma diferente e fugir de algumas dessas regras, assim como você pode nomear classes Java diferente dos arquivos em que elas estão e não seguir a regra do nome dos pacotes. Apesar disso, convenções existem por um motivo e é extremamente recomendado que você as siga. Elas facilitam muito a curva de aprendizagem de quem for começar a trabalhar em seus projetos, ou precisar manter no futuro, e evitam uma série de problemas, como a ambiguidade em nomes de diferentes projetos e bibliotecas.

Compilando projetos modulares

A forma de compilar e executar o projeto também muda. No lugar do `-cp`, indicando o *classpath* do projeto, agora vamos usar um `--module-path`, ou `-p`, que é o diretório no qual a máquina virtual poderá encontrar todo o *bytecode* e as definições dos módulos de nosso projeto.

Por padrão, isso é feito em uma pasta chamada `mods`, que ficará na mesma hierarquia do diretório `src`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
└── bookstore
    ├── mods
    │       └── br.com.casadocodigo
    │         ...
    │        └── module-info.class
    └── src
        └── br.com.casadocodigo
              ...
        └── module-info.java
```

Além do parâmetro `--module-path mods`, também precisamos apontar para o compilador onde devem ser gerados o bytecode do módulo, os `.class` e seus pacotes. Podemos fazer com o comando `-d`, que vai criar o diretório caso ele já não exista. Queremos que todo código compilado do módulo `br.com.casadocodigo` fique no diretório `mods/br.com.casadocodigo`.

Sabendo disso, o comando completo para compilar o projeto ficará assim:

```
javac -d mods/br.com.casadocodigo \
    --module-path mods \
    src/br.com.casadocodigo/module-info.java \
    $(find . -name "*.java")
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Perceba que usei o `$(find . -name "*.java")` mencionado no capítulo anterior, que é uma forma simples de buscar pela linha de comando todas as classes de um diretório em sistemas UNIX. Se você estiver em um Windows ou simplesmente preferir fazer de outra forma, pode passar manualmente cada um desses pacotes ou arquivos. Um exemplo seria:

```
javac -d mods/br.com.casadocodigo \
  --module-path mods \
  src/br.com.casadocodigo/module-info.java
  src/br.com.casadocodigo/br/com/casadocodigo/Main.java \
  src/br.com.casadocodigo/br/com/casadocodigo/model/Category
.java \
  src/br.com.casadocodigo/br/com/casadocodigo/model/Book.jav
a \
  src/br.com.casadocodigo/br/com/casadocodigo/data/Books.jav
a
...
```

Vamos experimentar? Executamos e, opa, há erro no *HTTP/2 Client*.

```
./src/br.com.casadocodigo
/br/com/casadocodigo/data/Books.java:7:
error: package jdk.incubator.http is not visible
import jdk.incubator.http.*;
                    ^
(package jdk.incubator.http is declared
in module jdk.incubator.httpclient,
but module br.com.casadocodigo does not read it)
```

Perceba que o erro nos diz exatamente o que aconteceu. O pacote `jdk.incubator.http`, que usamos na classe `Books`, existe dentro do módulo chamado `jdk.incubator.httpclient`,

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

mas **nosso módulo não o está lendo**. Essas mensagens de erro do compilador estão bem claras, ajudando bastante quando caímos em problemas assim — e acredite, vamos cair.

O que aconteceu aqui não foi muito diferente de quando o projeto não era modular. O `httpclient` está em incubação e não é incluído no Java SE por padrão, lembra? Estávamos resolvendo esse problema com o parâmetro extra `--add-modules jdk.incubator.httpclient` no `javac`. No lugar disso, essa e outras configurações do módulo agora ficam dentro do `module-info.java`.

Nosso módulo **precisa** do `jdk.incubator.httpclient` para funcionar, portanto podemos definir isso com a instrução `requires`, como a seguir.

```
module br.com.casadocodigo {
    requires jdk.incubator.httpclient;
}
```

Experimente compilar novamente:

```
javac -d mods/br.com.casadocodigo \
    --module-path mods/ \
    src/br.com.casadocodigo/module-info.java \
    $(find . -name "*.java")
```

Sucesso. Exatamente como antes, um warning do módulo em incubação será exibido e, agora, toda a estrutura do módulo com os arquivos compilados foi criada dentro da pasta `mods`.

```
├── mods
│   └── br.com.casadocodigo
│       ├── br
│       │   └── com
│       │       └── casadocodigo
│       │           ├── Main.class
│       │           ├── data
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
│       │           │   └── Books.class
│       │           ├── model
│       │           │   ├── Book.class
│       │           │   ├── Category.class
│       │           │   └── NF.class
│       │           └── service
│       │               ├── NFEmissor.class
│       │               ├── NFSubscriber.class
│       │               └── WSPrefeitura.class
│       └── module-info.class
│
└── src
    └── br.com.casadocodigo
    ...
```

Executando projetos modulares

Para executar o projeto, assim como na compilação, vamos usar o `--module-path`, apontando o local onde os arquivos compilados podem ser encontrados pela máquina virtual do Java. Além disso, precisamos usar o parâmetro `-m` apontando o nome completo da classe `Main`, que será executada. **Esse nome completo agora inclui, além do pacote, o nome do módulo ao qual a classe pertence.**

```
java --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

Funciona! O projeto foi executado e teve o mesmo comportamento de antes do modular, claro.

```
Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
...

Digite o número do livro que quer comprar:
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
//...
```

Perceba que agora o nome completo de uma classe é composto por:

```
<nome-do-modulo>/<nome-do-pacote>.<nome-da-classe>
```

7.4 TRABALHANDO COM DIFERENTES MÓDULOS

Já sabemos o que é um módulo e, inclusive, migramos o nosso projeto completo para dentro de um. Usar o sistema de módulos dessa forma, tendo um único módulo, pode até ser interessante pelas otimizações da JVM e detalhes de performance que veremos adiante. Mas, apesar disso, ainda temos diversos problemas de encapsulamento e na organização do código, especialmente conforme a aplicação for crescendo.

Muitos desses problemas de encapsulamento passam batidos nos seus projetos do dia a dia, e também nas APIs e bibliotecas que são desenvolvidas.

Um exemplo seria o `WSPrefeitura`, aquele componente usado pelo sistema de emissão de notas fiscais. O problema é que, por ser público, tendo o modificador de visibilidade `public`, ele fica acessível e poderia ser usado por **qualquer ponto da aplicação**, mesmo sendo algo completamente específico e que **deveria ser utilizado apenas pelo emissor de notas**.

Existem formas de contornar isso, como por exemplo tirando o `public` e deixando a visibilidade `default`, para que a classe fique acessível apenas pelo mesmo pacote. Em exemplos pequenos, isso pode ser suficiente, mas ao custo de travar todas as

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

implementações que entrarem nessa situação em um mesmo pacote — conforme a aplicação for crescendo, esse problema ficará mais e mais evidente. E nem sempre essa classe vai ser nossa, para que possamos editar sua visibilidade e esconder apenas dentro de um pacote.

Em frameworks e bibliotecas, esse problema é ainda pior. Tudo o que for público pode ser usado por quem importar essa dependência, por esse JAR — e nem sempre queremos que isso aconteça. É extremamente comum encontrar implementações internas, que deveriam ser usadas apenas no core da biblioteca, visíveis para quem a importou.

Outro detalhe curioso é que, basta o pacote ter o mesmo nome para você conseguir acessar as classes com visibilidade `default` — mesmo sendo de outro projeto! Em outras palavras, não há garantia alguma de que alguém que importa o seu JAR não criou um pacote com o mesmo nome que o das suas classes para acessar as implementações internas. Confesso que fiz isso mais vezes do que me orgulho.

Em resumo, o grande problema é que, até agora, na plataforma Java, **ser público significa ser acessível**. Esse foi um ponto muito importante que mudou com a introdução do sistema de módulos, do JDK 9.

Os pacotes de um módulo só são acessíveis se forem explicitamente exportados e, ainda assim, os outros módulos não podem acessar esse conteúdo sem declarar isso explicitamente, como fizemos com o `requires` do `httpclient` no `module-info.java` do projeto. Essa mudança traz a possibilidade de um encapsulamento muito mais forte das nossas implementações e, de

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

certa forma, nos ajuda a criar designs de APIs mais limpos e significativos.

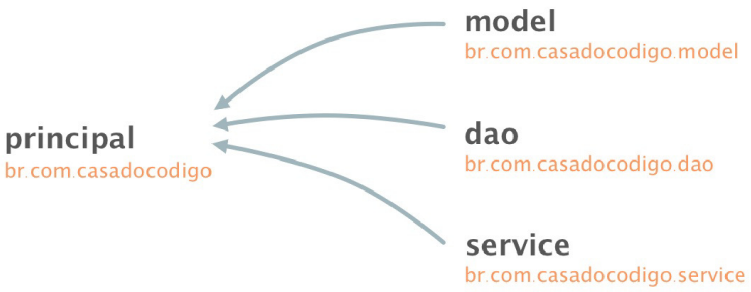
Mas afinal, como definir o que fica em cada módulo?

Podemos e vamos quebrar a bookstore em mais módulos, para que os problemas mencionados estejam evidentes e a vantagem dessa abordagem fique bem clara. A dúvida é: como essa divisão deve ser feita? É provavelmente a primeira pergunta que todo arquiteto ou desenvolvedor de um software modular deve se fazer.

Há uma discussão grande sobre como fazer isso e, sinceramente, não há bem uma resposta definitiva. Existem, sim, vantagens sobre uma abordagem ou outra, que eu vou defender aqui, mas a grande verdade é que o contexto, tamanho e escopo do projeto podem influenciar nessa decisão. Não é uma via de mão única.

Uma abordagem seria a divisão dos módulos por camadas de seu projeto, conhecida como `package-by-layer`. De forma resumida, você poderia ter um módulo com todos os seus DAOs, outro com os web services, outro com os modelos e assim por diante. Claro que, com o tempo, você poderia melhorar essa divisão separando, por exemplo, os módulos de modelos de livros dos módulos de nota fiscal e outros domínios da aplicação.

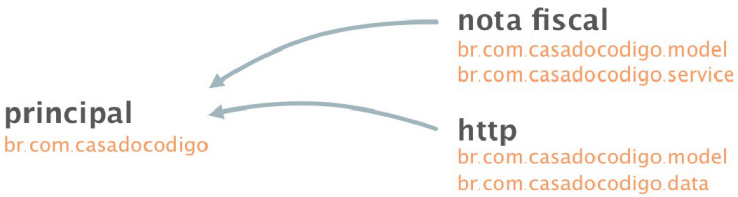
E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com



A vantagem dessa abordagem é a simplicidade. Fica mais fácil saber o que colocar em cada lugar, sem se preocupar tanto com os detalhes. O problema é o encapsulamento. Para alterar nossa implementação reativa de emissão das notas fiscais, por exemplo, precisaríamos mexer em diferentes módulos.

Isso nos leva a uma outra abordagem muito comum e recomendada, em que os módulos são divididos por funcionalidades. Ela é conhecida como *package-by-feature*.

Mesmo em um sistema pequeno como a nossa *bookstore*, essa divisão pode ser vista de forma clara. Temos uma funcionalidade de emissão das notas fiscais, outra de consumo de um *endpoint* web com *HTTP/2 Client*, e o core da aplicação, que tem a classe `Main`, juntando todas essas implementações.



E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Vamos partir para essa segunda abordagem, para sentirmos a diferença e suas principais vantagens de forma prática. Assim, esse benefício e ganho no encapsulamento certamente ficarão bem claros.

É hora de colocar as mãos na massa.

Melhorando a divisão dos módulos

Antes de qualquer coisa, vamos apagar a pasta `mods` com todo o nosso projeto compilado. Como o projeto será melhor divido, não queremos confusão com o *bytecode* da estrutura antiga.

Na linha de comando, podemos fazer isso assim:

```
rm -rf mods/
```

Podemos começar criando o módulo de notas fiscais, que poderá se chamar `br.com.casadocodigo.nf` . Isso precisa ser feito exatamente como antes. Vamos criar uma pasta com o nome desse módulo, e um arquivo `module-info.java` com sua definição inicial:

```
module br.com.casadocodigo.nf {

}
```

Podemos agora mover todos as classes e pacotes específicos dessa nossa funcionalidade de notas fiscais. Faremos isso com a classe `br/com/casadocodigo/model/NF.java` , e o diretório `br/com/casadocodigo/service` completo.

Ao final, o projeto ficará com esta estrutura:

```
└── src
    ├── br.com.casadocodigo
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
│   ├── br
│   │   └── com
│   │       └── casadocodigo
│   │           ├── Main.java
│   │           ├── data
│   │           │   └── Books.java
│   │           └── model
│   │               ├── Book.java
│   │               └── Category.java
│   └── module-info.java
└── br.com.casadocodigo.nf
    ├── br
    │   └── com
    │       └── casadocodigo
    │           ├── model
    │           │   └── NF.java
    │           └── service
    │               ├── NFEmissor.java
    │               ├── NFSubscriber.java
    │               └── WSPrefeitura.java
    └── module-info.java
```

Há um detalhe bem importante aqui. Diferentes módulos não podem ter pacotes iguais e, agora, ambos os módulos `br.com.casadocodigo` e `br.com.casadocodigo.nf` possuem um pacote `.model`. Ao tentar compilar o projeto, inevitavelmente receberíamos um:

```
error: package exists in another module:
br.com.casadocodigo.model;
```

Para evitar esse tipo de problema, é comum que o pacote base de dentro do módulo acompanhe seu nome. No caso do módulo de notas fiscais, os modelos ficariam dentro de `br.com.casadocodigo.nf.model`. Caso esteja fazendo isso pela linha de comando, como eu, lembre-se de editar todos os `import`s dos lugares que usem essas classes.

```
└── br.com.casadocodigo.nf
    ├── br
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    |    └── com
    |         └── casadocodigo
    |             ├── nf
    |             |    └── model
    |             |          └── NF.java
    |             └── service
    |                 ├── NFEmissor.java
    |                 ├── NFSubscriber.java
    |                 └── WSPrefeitura.java
    └── module-info.java
```

Vamos compilar?

A ordem dos módulos precisa ser respeitada no momento da compilação, portanto, o módulo de notas será o primeiro.

```
javac -d mods/br.com.casadocodigo.nf \
    --module-path mods \
    src/br.com.casadocodigo.nf/module-info.java \
    $(find src/br.com.casadocodigo.nf -name "*.java")
```

E ele falha. Veja o erro:

```
src/br.com.casadocodigo.nf/br/com/casadocodigo/service/NFEmissor.
java:16:
error: cannot find symbol
    public void emit(String clientName, Book book) {
                                        ^
  symbol:   class Book
  location: class NFEmissor
```

O `NFEmissor` precisa do modelo `Book`, que ficou no módulo principal. Não queremos ter um problema de dependências cíclicas agora, que veremos adiante. Assim, a princípio, vamos trazer os outros dois modelos para dentro desse módulo de notas. Não se preocupe, logo vamos discutir sobre o evidente erro nessa decisão.

Mova as classes `Book` e `Category` para o pacote `br.com.casadocodigo.nf.model`. Não se esqueça de atualizar os `import`s.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
└── br.com.casadocodigo.nf
    ├── br
    │   └── com
    │       └── casadocodigo
    │           └── nf
    │             ├── model
    │             │     ├── NF.java
    │             │     ├── Book.java
    │             │     └── Category.java
    │             └── service
    │                 ├── NFEmissor.java
    │                 ├── NFSubscriber.java
    │                 └── WSPrefeitura.java
    └── module-info.java
```

Agora sim, podemos compilar novamente e tudo deve funcionar. Precisamos agora compilar o módulo principal.

```
javac -d mods/br.com.casadocodigo \
    --module-path mods \
    src/br.com.casadocodigo/module-info.java
    $(find src/br.com.casadocodigo -name "*.java")
```

E temos outra falha.

```
error: package br.com.casadocodigo.nf.service is not visible
import br.com.casadocodigo.nf.service.*;
                            ^
(package br.com.casadocodigo.nf.service is declared
in module br.com.casadocodigo.nf,
but module br.com.casadocodigo does not read it)
```

Ele diz que, apesar de o pacote `br.com.casadocodigo.nf.service` existir no módulo de notas fiscais, ele não é visível. Isso vai ao encontro com o que eu disse sobre **ser público não significar mais ser acessível**. Não basta o pacote existir e suas classes serem públicas. Ele precisa ser explicitamente exportado para que outros módulos o consigam usar. Dessa forma, tudo fica extremamente encapsulado dentro de seus módulos.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

A alteração é simples. O módulo principal, `br.com.casadocodigo`, **precisa** dos pacotes do módulo de notas fiscais, portanto, em seu `module-info.java`, ele vai declarar isso exatamente como fizemos para usar o `httpclient`, com uso do `requires`.

```
module br.com.casadocodigo {
    requires jdk.incubator.httpclient;
    requires br.com.casadocodigo.nf;
}
```

Na outra ponta, o módulo `br.com.casadocodigo.nf` precisa deixar claro o que deve ficar acessível, ou seja, o que quer **exportar** para os outros módulos. Isso pode ser feito com o `exports`, em cada um de seus pacotes que devem ser acessíveis. Por enquanto, vamos deixar os dois:

```
module br.com.casadocodigo.nf {
    exports br.com.casadocodigo.nf.service;
    exports br.com.casadocodigo.nf.model;
}
```

Agora sim, podemos recompilar os dois módulos. Começando pelo de notas fiscais:

```
javac -d mods/br.com.casadocodigo.nf \
    --module-path mods \
    src/br.com.casadocodigo.nf/module-info.java
    $(find src/br.com.casadocodigo.nf -name "*.java")
```

E agora o principal:

```
javac -d mods/br.com.casadocodigo \
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    --module-path mods \
    src/br.com.casadocodigo/module-info.java
    $(find src/br.com.casadocodigo -name "*.java")
```

Pronto, tudo compila. Podemos executar o projeto novamente, com o mesmo comando de antes, para ter certeza de que continua funcionando.

```
java --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

Sucesso!

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Controlando ainda mais o encapsulamento

O `exports`, que vimos aqui, define que um determinado pacote pode ser acessível para qualquer outro módulo do projeto. No lugar disso, podemos ter um controle mais fino de quem pode acessar qual pacote, definindo especificamente quais os módulos que podem acessá-los com uso do `exports to`, como a seguir:

```
exports
    <pacote>
to
    <modulo1>, <modulo2>;
```

Em nosso caso, poderíamos deixar explícito que o pacote `service` só pode ficar acessível para o módulo principal da aplicação, e nenhum outro que exista ou venha existir no futuro.

```
exports
    br.com.casadocodigo.nf.service
to
    br.com.casadocodigo;
```

Além disso, podemos usar o `open` que, diferente do `exports`, indica que todas as classes (sejam públicas ou privadas) de um determinado pacote podem ser acessíveis via reflection, em tempo de execução.

Cuidado com o excesso de responsabilidades

Há um problema bem grande no design da nossa divisão de módulos. Nosso módulo de notas fiscais ficou com mais

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

responsabilidades do que deveria, já que, além de encapsular todo o processo de emissão das notas, ele também ficou com a declaração da classe `Book` e seu `enum Category`.

O problema é claro: sempre que algum outro ponto do nosso projeto precisar da classe `Book`, será necessário importar a solução completa de notas fiscais, que não tem nada a ver com isso. Não podemos deixar assim.

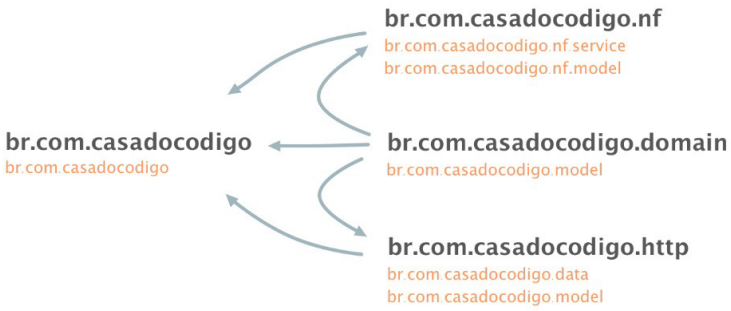
Aqui é importante perceber que, apesar de o sistema de módulos ajudar muito no problema de encapsulamento, **não existe bala de prata**. Não basta começar a usá-lo em seus projetos para ter APIs coesas e bem definidas. É preciso tomar muito cuidado com a forma como você vai organizar isso tudo na prática.

Vamos corrigir?

Criando os módulos domain e http

Umas das possíveis soluções para esse problema seria dividindo ainda mais as partes do nosso projeto. Podemos criar um novo módulo, chamado `br.com.casadocodigo.domain`, com as classes de domínio da aplicação — que vão ser usadas em todos os lugares —, e um outro, chamado `br.com.casadocodigo.http`, com a classe `Books` que encapsula toda a comunicação HTTP.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com



Vamos começar pelo módulo `domain`. Basta fazer o mesmo processo de antes, como no passo a passo:

— Criar um diretório chamado `br.com.casadocodigo.domain`.

```
└── bookstore
    └── src
        └── br.com.casadocodigo
                ...
        └── br.com.casadocodigo.nf
            ...
        └── br.com.casadocodigo.domain
```

— Mover o pacote `model` e as classes `Book` e `Category` para dentro desse novo módulo.

```
└── br.com.casadocodigo.domain
    └── br
        └── com
            └── casadocodigo
                └── model
                    ├── Book.java
                    └── Category.java
```

Perceba que a classe `NF.java` continua no módulo de notas fiscais. Diferente das outras, ela existe especificamente para essa

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

feature e não deve ser usada em nenhum outro ponto da aplicação.

Não esqueça de atualizar a declaração do pacote de ambas as classes `Book` e `Category` para `br.com.casadocodigo.model` .

Além disso, será necessário:

- No módulo de http, tirar o `.nf` do `import br.com.casadocodigo.nf.model` da classe `Books.java` .
- No módulo de notas fiscais, adicionar o `import br.com.casadocodigo.model` na classe `NFEmissor.java` . Você vai precisar manter o que já está lá, com o `.nf` , pois a classe `NF.java` continua neste pacote.
- No módulo principal, tirar o `.nf` do `import br.com.casadocodigo.nf.model` da classe `Main.java` .

Por fim, criar o arquivo `module-info.java` , dentro do diretório `br.com.casadocodigo.domain` , com sua definição.

```
module br.com.casadocodigo.domain {
    exports br.com.casadocodigo.model;
}
```

Já estamos declarando o `exports` , indicando que o pacote `model` deve ficar acessível para todos que incluírem o módulo `domain` , exatamente da forma como estávamos fazendo no arquivo `module-info.java` do módulo de notas. Assim, todos que adicionarem o `requires` `br.com.casadocodigo.domain` vão ter acesso às classes `Book` e `Category` , e isso fica

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

completamente desacoplado do restante do código.

Vamos criar o próximo módulo? Os passos são os mesmos:

- Criar o diretório com nome do módulo que, dessa vez, será `br.com.casadocodigo.http` .
- Mover a classe `Books` para o pacote `br.com.casadocodigo.http` .
- Criar o arquivo `module-info.java` , com sua definição.

```
module br.com.casadocodigo.http {
}
```

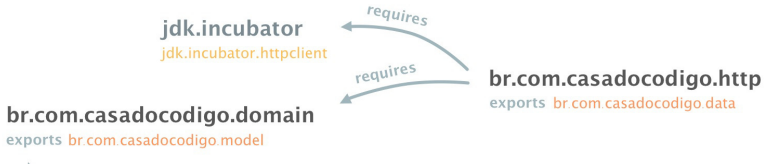
O módulo de `http` **precisa** da classe `Book` , portanto, vamos adicionar o `requires` :

```
module br.com.casadocodigo.http {
    requires br.com.casadocodigo.domain;
}
```

Além disso, agora é o módulo http que precisa do `httpclient` , no lugar de nosso módulo principal. Podemos mover a linha do `requires` do módulo `httpclient` para lá:

```
module br.com.casadocodigo.http {
    requires jdk.incubator.httpclient;
    requires br.com.casadocodigo.domain;
}
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com



Pronto, já definimos tudo o que o módulo **precisa**. Agora precisamos definir o que o módulo **exporta**, que nesse caso será o pacote `http` com a classe `Books`.

```
module br.com.casadocodigo.http {
    exports br.com.casadocodigo.http;
    requires jdk.incubator.httpclient;
    requires br.com.casadocodigo.model;
}
```

Com todas essas alterações, a estrutura completa do projeto fica assim:

```
└── src
    ├── br.com.casadocodigo
    │   ├── br
    │   │   └── com
    │   │       └── casadocodigo
    │   │           └── Main.java
    │   └── module-info.java
    ├── br.com.casadocodigo.http
    │   ├── br
    │   │   └── com
    │   │       └── casadocodigo
    │   │           └── http
    │   │               └── Books.java
    │   └── module-info.java
    ├── br.com.casadocodigo.domain
    │   ├── br
    │   │   └── com
    │   │       └── casadocodigo
    │   │           └── model
    │   │               ├── Book.java
    │   │               └── Category.java
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    │   └── module-info.java
    └── br.com.casadocodigo.nf
        ├── br
        │   └── com
        │       └── casadocodigo
        │           ├── model
        │           │   └── NF.java
        │           └── service
        │               ├── NFEmissor.java
        │               ├── NFSubscriber.java
        │               └── WSPrefeitura.java
        └── module-info.java
```

E o arquivo `br.com.casadocodigo/module-info.java` fará `requires` em todos os módulos, já que ele é o ponto principal do projeto e faz uso de todas as funcionalidades.

```
module br.com.casadocodigo {
    requires br.com.casadocodigo.nf;
    requires br.com.casadocodigo.domain;
    requires br.com.casadocodigo.http;
}
```

Além do `br.com.casadocodigo.domain/module-info.java`, que agora define e exporta o pacote `.model`, com as classes `Book` e `Category`.

```
module br.com.casadocodigo.domain {
    exports br.com.casadocodigo.model;
}
```

O `br.com.casadocodigo.http/module-info.java` precisa dos módulos `httpclient` do Java, e de nosso `domain`. Ele exporta o pacote `http`, com a classe `Books`, que encapsula todo o seu trabalho.

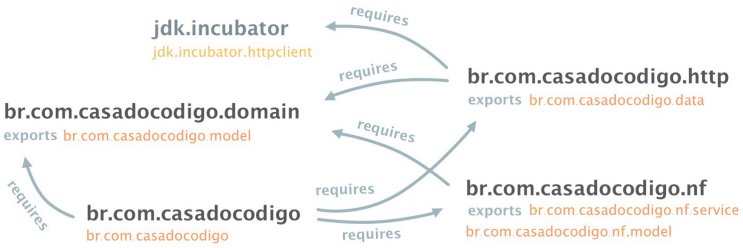
```
module br.com.casadocodigo.http {
    exports br.com.casadocodigo.http;
    requires jdk.incubator.httpclient;
    requires br.com.casadocodigo.domain;
}
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

O `br.com.casadocodigo.nf/module-info.java` precisa do módulo `domain`, com a classe `Book`, e exporta o pacote `service` que tem o emissor de notas fiscais. Agora que as classes de domínio estão bem distribuídas no projeto, não precisamos mais exportar o pacote `model` daqui, com a classe `NF.java`. **Isso garante um encapsulamento mais forte da parte dessa API**.

```
module br.com.casadocodigo.nf {
    exports br.com.casadocodigo.nf.service;
    requires br.com.casadocodigo.domain;
}
```

Visualmente, a divisão do projeto fica assim:



Vamos compilar tudo?

Lembre-se de apagar o diretório `mods`, com a antiga estrutura compilada, e de compilar as partes do projeto na ordem correta. Vale lembrar que as IDEs fazem todo esse trabalho por nós, claro. O objetivo aqui é entendê-lo a fundo, antes das facilidades dos editores.

```
javac -d mods/br.com.casadocodigo.domain \
    --module-path mods \
    src/br.com.casadocodigo.domain/module-info.java \
    $(find src/br.com.casadocodigo.domain -name "*.java")
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
javac -d mods/br.com.casadocodigo.http \
    --module-path mods \
    src/br.com.casadocodigo.http/module-info.java \
    $(find src/br.com.casadocodigo.http -name "*.java")

javac -d mods/br.com.casadocodigo.nf \
    --module-path mods \
    src/br.com.casadocodigo.nf/module-info.java \
    $(find src/br.com.casadocodigo.nf -name "*.java")

javac -d mods/br.com.casadocodigo \
    --module-path mods \
    src/br.com.casadocodigo/module-info.java \
    $(find src/br.com.casadocodigo -name "*.java")
```

Tudo compila. Legal, não é?

Para nos certificarmos que tudo continua a funcionar, podemos executar o projeto de novo:

```
java --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

Tudo certo!

Essa divisão e encapsulamento mais forte ajudam muito na manutenibilidade e evolução do código. Você só deixa passar, explicitamente, o que realmente deve ser acessado por fora de seus módulos. Isso o força a pensar se aquela informação que você está abrindo realmente pode ou precisa se espalhar pelos outros pontos do projeto.

E uma dica importante: **cuidado com a organização de seus pacotes.** Em nosso próprio exemplo, no módulo de notas fiscais, estamos quebrando o encapsulamento ao deixar todas as classes no pacote `service`, que é aberto e pode ser usado pelos demais. Isto é, mesmo sendo uma dependência e implementação interna desse módulo, as classes `WSPrefeitura` e `NFSubscriber` estão abertas

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

e podem ser usadas por fora desse módulo.

```
br.com.casadocodigo.nf
exports br.com.casadocodigo.nf.service
        └─ NFEmissor.java
        └─ WSPrefeitura.java
        └─ NFSubscriber.java
        br.com.casadocodigo.nf.model
        └─ NF.java
```

A solução é simples: organize seus pacotes, separando as implementações das APIs públicas. O `NFEmissor` deveria estar em um pacote separado, e apenas esse pacote deveria estar aberto para os demais módulos. Evite ao máximo expor suas implementações.

```
br.com.casadocodigo.nf
exports br.com.casadocodigo.nf.service
        └─ NFEmissor.java

        br.com.casadocodigo.nf.internal
        └─ WSPrefeitura.java
        └─ NFSubscriber.java
        br.com.casadocodigo.nf.model
        └─ NF.java
```

Mais definições de dependências entre módulos

Perceba que, por bem ou por mal, todos os nossos módulos dependem do `br.com.casadocodigo.domain`, onde ficam as classes de domínio importantes para todas nossas regras de negócio. O módulo `http`, por exemplo, retorna uma lista do tipo `Book`, que pertence ao `domain`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Em outras palavras, todo módulo que for usar o `br.com.casadocodigo.http` vai precisar adicionar esses dois `requires`:

```
requires br.com.casadocodigo.http;
requires br.com.casadocodigo.domain;
```

O mesmo acontece com o módulo de notas fiscais, que precisa do livro do tipo `Book` para iniciar o seu processo de emissão assíncrona. De certa forma, tudo está conectado a essa classe, e talvez a outras que possam aparecer futuramente nesse módulo que chamamos de `domain`.

Para um cenário como esse, que é extremamente comum, podemos deixar explícita essa dependência na declaração dos módulos com o uso da palavra reservada `transitive`.

O arquivo `br/com/casadocodigo/http/module-info.java` ficará assim:

```
module br.com.casadocodigo.http {
    exports br.com.casadocodigo.http;
    requires jdk.incubator.httpclient;
    requires transitive br.com.casadocodigo.domain;
}
```

E o `br/com/casadocodigo/nf/module-info.java` ficará:

```
module br.com.casadocodigo.nf {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    exports br.com.casadocodigo.nf.service;
    requires transitive br.com.casadocodigo.domain;
}
```

Dessa forma, sempre que alguém importar algum desses dois módulos, implicitamente estará importando o de `domain` junto.

Vamos testar? Experimente **tirar a declaração** do `domain` no `module-info.java` principal do projeto.

Ele ficará assim:

```
module br.com.casadocodigo {
    requires br.com.casadocodigo.nf;
    requires br.com.casadocodigo.http;
}
```

Não esqueça de ajustar os módulos `http` e `nf`, declarando a dependência ao módulo `domain` como transitiva.

E agora, ao compilar novamente todos os módulos, mesmo sem estar declarando explicitamente a necessidade do `domain`, o módulo principal vai compilar, já que está recebendo esse outro módulo de forma implícita.

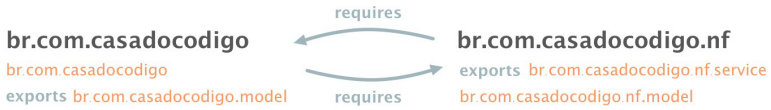
O problema das dependências cíclicas e o static

Talvez você tenha se perguntado porque, logo no início da modularização do projeto, eu disse para mover o pacote `model`

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

completo — com as classes `Book` e `Category`, inclusive — para o módulo de `nf`, que precisava delas para funcionar. Pense no que aconteceria se essas duas classes tivessem ficado no módulo principal.

Já sabe a resposta? Se não, a imagem a seguir deve ajudar.



Repare que o módulo principal precisa do módulo `nf` que, por sua vez, precisa do módulo principal para compilar. É o famoso **problema de dependências cíclicas**.

Como resolver? Bem, em nosso contexto, eu diria que ter separado essas classes em um módulo foi uma boa saída. Assim, podemos sempre garantir o encapsulamento forte, só declarando essa dependência quando, e se, precisarmos. Essa será, muitas vezes, a melhor solução para os seus projetos também.

Apesar disso, nem sempre essa decisão vai fazer sentido ou mesmo será suficiente para resolver o problema. Nesses casos, podemos usar o modificador `static` ao definir a dependência, informando ao compilador que aquele relacionamento está ali apenas para que o código compile, mas dando uma garantia de que, em execução, ele estará disponível de alguma outra forma.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Se você está familiarizado com Maven, certamente pode assemelhar isso com seu escopo *provided*. Quando você usa um servidor de aplicação, por exemplo, ele já vem com o JPA e outras implementações do Java EE. Mesmo que o JPA já exista lá no servidor, para quando o projeto estiver sendo executado, em sua máquina na fase de desenvolvimento **você precisará ter essa dependência declarada para que o código compile**. O escopo *provided* cuida disso, garantindo que, no momento de empacotar o projeto, o JPA não será inserido em duplicidade no servidor.

No caso do *static*, do projeto modular, você está declarando aquela dependência para que o módulo compile **individualmente**, já que precisa de classes que estão no outro módulo. Apesar disso, o JDK sabe que deve ignorar essa declaração em execução, pois você está dando a garantia de que aquela classe com toda certeza vai existir ao empacotar o projeto.

Em outras palavras, ela será usada em tempo de compilação, mas não em execução.

Vamos testar?

Uma forma de perceber isso na prática seria trocando o `transitive` dos módulos `http` e `nf` por `static`, assim o módulo `domain` estará lá **apenas para efeito de compilação**.

O `br.com.casadocodigo.http/module-info.java` ficará

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

assim:

```
module br.com.casadocodigo.http {
    exports br.com.casadocodigo.http;
    requires jdk.incubator.httpclient;
    requires static br.com.casadocodigo.domain;
}
```

E o `br.com.casadocodigo.nf/module-info.java` ficará:

```
module br.com.casadocodigo.nf {
    exports br.com.casadocodigo.nf.service;
    requires static br.com.casadocodigo.domain;
}
```

Agora experimente compilar todos os módulos novamente.

Todos compilam, com exceção do principal:

```
src/br.com.casadocodigo/br/com/casadocodigo/Main.java:4:
error: package br.com.casadocodigo.model is not visible
import br.com.casadocodigo.model.*;
                          ^
(package br.com.casadocodigo.model is declared in
  module br.com.casadocodigo.domain,
  but module br.com.casadocodigo does not read it)
```

Faz sentido! Antes, enquanto estávamos usando `transitive`, recebíamos o `domain` **implicitamente** para que o código da classe principal compilasse. Agora que estamos usando `static`, precisamos declará-lo explicitamente no módulo principal novamente.

Experimente adicionar o `domain`, mas usando o `static` exatamente como fizemos nos demais módulos do projeto.

O `br/com/casadocodigo/module-info.java` ficará assim:

```
module br.com.casadocodigo {
    requires br.com.casadocodigo.nf;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    requires br.com.casadocodigo.http;
    requires static br.com.casadocodigo.domain;
}
```

E agora vamos compilar. Ok, funciona!

O código compila, mas, ao executar...

```
Exception in thread "main"

java.lang.NoClassDefFoundError:
  br/com/casadocodigo/model/Book
  at br.com.casadocodigo/br.com.casadocodigo.Main.<clinit>
  (Main.java:56)

Caused by: java.lang.ClassNotFoundException:
br.com.casadocodigo.model.Book
```

A classe não existe.

Lembra da regra desse `static`? As dependências estáticas **são usadas apenas em tempo de compilação**, mas não em execução — *runtime*. O código compila, porém, na hora de executar, não funciona. O `domain` estará lá **apenas para efeito de compilação**.

Para usar o `static` dessa forma, o módulo principal precisa sempre declarar da forma natural, sem ser `static`, dando a garantia de que o *bytecode* dessa dependência esteja sempre disponível também na hora de empacotar e executar o projeto.

Experimente mudar:

```
module br.com.casadocodigo {
    requires br.com.casadocodigo.nf;
    requires br.com.casadocodigo.http;
    requires br.com.casadocodigo.domain;
}
```

Agora sim, o código todo compila e funciona. A aplicação

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

executa normalmente.

> Existem situações em que podemos usar o `transitive` e `static` juntos! Isso significaria que a dependência estaria disponível em tempo de compilação, implicitamente, também para os módulos que dependerem desse primeiro.

7.5 O JDK MODULAR

A essa altura, você provavelmente já sabe, ou percebeu, que o sistema modular não é uma ferramenta exclusiva para o código dos nossos projetos como usuários finais. O JDK inteiro foi reestruturado em módulos. Quer fazer um teste?

Até agora, não usamos nada de diferente, fora a `String`, primitivos e listas do `java.util`. Experimente usar alguma classe de fora dos pacotes padrões da plataforma.

Você pode testar com um `import` do `java.sql`, na classe `Main`, por exemplo, e também do `swing`:

```
import java.sql.*;
import javax.swing.*;
```

Compile! Teremos:

```
src/br.com.casadocodigo/br/com/casadocodigo/Main.java:10:
error: package java.sql is not visible
import java.sql.*;
           ^
(package java.sql is declared in module java.sql,
  but module br.com.casadocodigo does not read it)
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
src/br.com.casadocodigo/br/com/casadocodigo/Main.java:11:
error: package javax.swing is not visible
import javax.swing.*;
            ^
(package javax.swing is declared in module java.desktop,
but module br.com.casadocodigo does not read it)
```

O erro diz exatamente o problema: esses pacotes pertencem aos módulos `java.sql` e `java.desktop`, que não estamos declarando como dependência em nosso `module-info`. Por padrão, todo sistema modular já vem com o módulo `java.base`, contendo a String e todo `java.lang`, `java.io`, `java.util` e demais pacotes muitas vezes essenciais para a esmagadora maioria dos projetos. Implementações específicas como *SQL*, *Swing*, *JavaFX* etc. ficam em módulos separados.

O módulo `java.base` está sempre presente. Você pode declará-lo em todos os seus projetos, porém não é obrigatório — assim como o pacote `java.lang`, em suas classes.

Para todos os demais módulos, você precisará declarar explicitamente no `module-info.java` os que precisar, da mesma forma que declaramos as dependências entre módulos do nosso projeto.

Para que o código compile com os `import`s do `java.sql` e `java.swing`, da forma como fizemos, bastaria adicionar na declaração do nosso módulo principal:

```
module br.com.casadocodigo {

    // our modules
    requires br.com.casadocodigo.nf;
    requires br.com.casadocodigo.http;
    requires br.com.casadocodigo.domain;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    // platform modules
    requires java.sql;
    requires java.desktop;
}
```

Para listar e conhecer todos os módulos da plataforma, você pode usar o comando:

```
java --list-modules
```

São 94 ao total! O resultado deverá ser parecido com:

```
java.activation@9
java.base@9
java.compiler@9
java.corba@9
java.datatransfer@9
java.desktop@9
java.instrument@9
java.jnlp@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.se@9
java.se.ee@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9
java.sql@9
java.sql.rowset@9
java.transaction@9
java.xml@9
java.xml.bind@9
java.xml.crypto@9
java.xml.ws@9
java.xml.ws.annotation@9
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
jdk.accessibility@9
jdk.attach@9
jdk.charsets@9
jdk.compiler@9
jdk.crypto.cryptoki@9
jdk.crypto.ec@9
jdk.deploy@9
jdk.deploy.controlpanel@9
jdk.dynalink@9
jdk.editpad@9
jdk.hotspot.agent@9
jdk.httpserver@9
jdk.incubator.httpclient@9
jdk.internal.ed@9
jdk.internal.jvmstat@9
jdk.internal.le@9
jdk.internal.opt@9
jdk.internal.vm.ci@9
jdk.jartool@9
jdk.javadoc@9
jdk.javaws@9
jdk.jcmd@9
jdk.jconsole@9
jdk.jdeps@9
jdk.jdi@9
jdk.jdwp.agent@9
jdk.jfr@9
jdk.jlink@9
jdk.jshell@9
jdk.jsobject@9
jdk.jstatd@9
jdk.localedata@9
jdk.management@9
jdk.management.agent@9
jdk.naming.dns@9
jdk.naming.rmi@9
jdk.net@9
jdk.pack@9
jdk.packager@9
jdk.packager.services@9
jdk.plugin@9
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
jdk.plugin.dom@9
jdk.plugin.server@9
jdk.policytool@9
jdk.rmic@9
jdk.scripting.nashorn@9
jdk.scripting.nashorn.shell@9
jdk.sctp@9
jdk.security.auth@9
jdk.security.jgss@9
jdk.snmp@9
jdk.unsupported@9
jdk.xml.bind@9
jdk.xml.dom@9
jdk.xml.ws@9
jdk.zipfs@9
oracle.desktop@9
oracle.net@9
```

Por padrão, os módulos que implementam a especificação do Java SE possuem o prefixo `java`, como o próprio `java.base`, `java.desktop`, `java.xml`, `java.sql` etc. O *JavaFx* foi dividido em diversos módulos também, que possuem o prefixo `javafx`.

Todas as implementações específicas da Oracle começam com `oracle` e, por fim, as específicas do JDK também seguem esse padrão com o prefixo `jdk`. Um exemplo desse último é o próprio módulo do `httpclient` que, enquanto está em incubação, fica no `jdk.incubator`.

Você consegue ver detalhes de um módulo específico utilizando a opção `-d`, ou `--describe-module`.

Podemos experimentar no módulo `java.base`:

```
java -d java.base

exports java.io
exports java.lang
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
exports java.math
exports java.text
exports java.time
exports java.util
...
```

Também podemos usar essa opção em algum módulo do nosso próprio projeto, mas nesse caso precisamos informar o `module-path`, para que o Java saiba onde deve procurar as definições de seus módulos.

```
java --module-path mods -d br.com.casadocodigo

bookstore
/mods/br.com.casadocodigo/
    requires br.com.casadocodigo.nf
    requires br.com.casadocodigo.http
    requires br.com.casadocodigo.domain
    requires java.base mandated
    contains br.com.casadocodigo
```

Legal, não é? Depois de 20 anos com uma estrutura monolítica, finalmente o JDK foi completamente redesenhado para que nossas aplicações possam usar apenas o pedacinho que precisar.

Em nosso projeto `bookstore`, no lugar dos 94 possíveis módulos da plataforma Java, **estamos usando o único que precisamos**, que é o `java.base`. Isso tem uma série de vantagens de performance e consumo de memória, já que a aplicação fica com um *footprint* inicial muito menor.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Quebra de compatibilidade e o famoso sun.misc.Unsafe

Durante a reestruturação da plataforma, chegou a ser cogitado que implementações internas, como o `sun.misc.Unsafe`, seriam removidas, o que causaria de certa forma uma quebra de compatibilidade em bibliotecas que dependiam dessas implementações — não deveriam, mas dependiam.

Para evitar essa quebra, foi decidido que, antes de apagá-las, essas implementações ficarão em um módulo específico do `jdk` chamado `jdk.unsupported`. O objetivo é que as bibliotecas tenham um tempo maior para migrar seu código, deixando de usar essas implementações, para que possam finalmente ser apagadas.

```
java -d jdk.unsupported

exports com.sun.nio.file
exports sun.misc
exports sun.reflect
requires java.base mandated
opens sun.reflect
opens sun.misc
```

O JDK 9 também introduziu uma fase opcional chamada `link-time`, que acontece entre a compilação e execução dos nossos projetos. Esse é o momento em que é criada uma imagem de execução do JRE customizada para o nosso código, apenas com os módulos que precisamos. Esse processo todo, com uso do `jlink` e outras ferramentas da nova versão, será um dos tópicos que vamos ver no próximo capítulo.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Maven, OSGi e outros

O sistema de módulos foi criado para oferecer uma forma de estruturar melhor aplicações e o próprio código da plataforma Java, resolvendo assim diversos dos problemas que vimos aqui no capítulo e ainda vamos ver durante o livro. Não é objetivo substituir o Maven, por exemplo, que é uma ferramenta de *build*. Os dois são necessários, mas diferentes.

O sistema de módulos vai nos ajudar a agrupar pacotes e recursos, encapsulando melhor as implementações internas, definindo bem suas dependências e, no final, ligando tudo de uma forma que você use apenas o que for necessário do JRE, que também foi completamente modularizado. O Maven vai continuar cuidando do processo de *build*, download e gerenciamento das dependências externas, além do controle de versões, exatamente como antes.

A única diferença é que, muito provavelmente, muitas dessas dependências poderão ser módulos a partir de agora. Em outras palavras, os dois se completam.

Já com *OSGi*, uma solução que já te permite escrever e estruturar um projeto em módulos, a história é completamente diferente. Ele resolve muitos desses problemas de *runtime*, mencionados pela limitação do legado *classpath*, e permite um controle mais fino do encapsulamento do seu projeto com seu conceito de *bundles*. A grande diferença é que, como uma biblioteca, ele não pode fazer muito sobre o controle e estrutura da plataforma e da JVM em si. E esse é o principal papel do sistema de módulos.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Essas e outras ferramentas ainda têm seu espaço no mercado, e agora possuem a vantagem de que podem tirar muito proveito dessa estrutura modular e novas possibilidades da plataforma.

7.6 PARA SABER MAIS

Arranhamos a superfície do modular, mas ainda há muito para ser discutido e praticado! No decorrer do livro vamos explorar novas possibilidades da arquitetura modular do JDK, além de boas práticas e outros benefícios dessa mudança enorme. Caso queira aprofundar mais a leitura, certamente vai se interessar pelo documento oficial da Oracle, *The State of the Module System*:

http://openjdk.java.net/projects/jigsaw/spec/sotms/

Também existe um projeto no repositório do AdoptOpenJDK, com diversos exemplos de código e tutoriais de migração para o modular.

https://github.com/AdoptOpenJDK/jdk9-jigsaw

7.7 DOWNLOAD DO PROJETO COMPLETO

Se quiser, você pode baixar o projeto completo, que está disponível na pasta deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 8

CRIANDO IMAGENS DE EXECUÇÃO CUSTOMIZADAS

Nossa `bookstore` já é modular e, como vimos, isso tem uma série de vantagens. O destaque especial talvez fique para o enorme ganho de encapsulamento das implementações e também na integridade forte que existe entre as fases de compilação e execução do projeto — dessa forma, erros são evitados ou, pelo menos, descobertos muito antes.

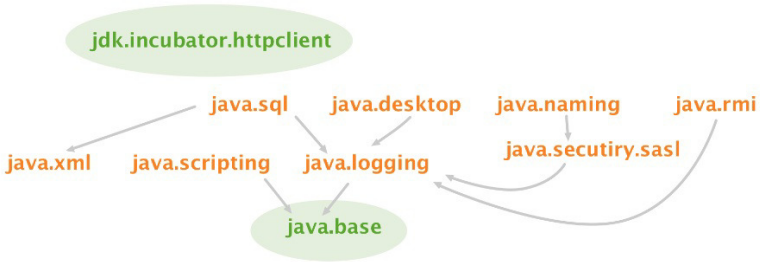
Também vimos que o próprio JDK foi inteiro reestruturado em módulos, abandonando o legado monolítico. O código relacionado a SQL fica no módulo `java.sql`, o de programação desktop com *Swing* fica no `java.desktop`, manipulação de XML em `java.xml` e assim por diante. A seguinte imagem mostra um pedacinho da relação entre os 94 módulos da plataforma.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

O `java.base` está sempre presente, implicitamente. E como vimos, cada um de nossos módulos precisa declarar explicitamente em seu arquivo `module-info` quais as dependências adicionais que precisam da plataforma. Um exemplo seria a definição do módulo de `http`, que declara o `httpclient` para usar a nova API de *HTTP/2*:

```
module br.com.casadocodigo.http {
    ...
    requires jdk.incubator.httpclient;
}
```

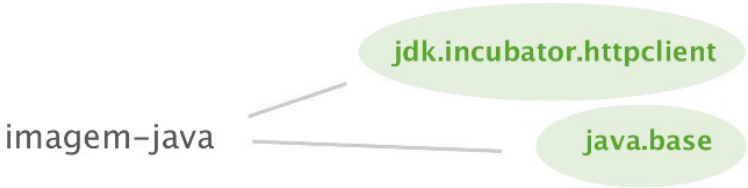
Apesar disso, quando executamos um projeto modular, **ainda estamos usando o JRE completo** assim como era feito antes. Em outras palavras, independente de usarmos apenas 2 dos 94 módulos existentes, por padrão todos são carregados.



E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Afinal, se só dependemos do `java.base` e `jdk.incubator.httpclient`, por que precisamos do JRE completo para executar? A resposta é simples, **não precisamos**.

O JDK 9 introduziu uma nova fase opcional chamada *link-time*, que acontece entre a compilação e a execução de um projeto modular. Nela, com uso da ferramenta `jlink`, é possível criar uma imagem customizada incluindo apenas o pedacinho do JRE que nossa aplicação precisa para executar.



O ganho é enorme, desde o *startup* mais rápido até um *footprint* muito menor em memória. No lugar do monolítico gigante, podemos ter um ambiente de execução com tamanho ótimo para o projeto.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Compact Profiles

O Java 8 já tinha dado um grande passo para a direção de minimizar o tamanho da JRE, com a introdução dos *compact profiles*. De forma resumida, a plataforma foi dividida em 3 profiles e você pode gerar um ambiente de execução específico para o seu projeto escolhendo algum deles, com uso da ferramenta `jrecreate` . Ainda assim, mesmo escolhendo o menor dos compact profiles, nosso projeto estaria usando muito mais do que precisa. O `jlink` muda completamente essa realidade.

Você pode ver mais detalhes em:

http://www.oracle.com/technetwork/java/embedded/resources/tech/compact-profiles-overview-2157132.html

8.1 CRIANDO UMA IMAGEM DE EXECUÇÃO CUSTOMIZADA

Quando li sobre essa possibilidade pela primeira vez, confesso que imaginei ser algo extremamente complexo. Afinal, estamos falando sobre criar uma imagem customizada só com algumas partes do JRE; a frase em si é intimidadora!

A realidade é que o processo é verdadeiramente simples, e o `jlink` tem uma estrutura bastante familiar.

```
jlink <options>
    --module-path <modulepath>
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
--add-modules <module>[,<module>...]
--output <imagename>
```

O `module-path` daqui, diferente de quando estamos compilando ou executando nosso projeto, é a pasta `jmods` do Java. Se você for curioso como eu, certamente já abriu o diretório em que o Java 9 está instalado na sua máquina para ver o que mudou. Se ainda não o fez e quer experimentar agora, vai encontrar algo parecido com:

```
README.html
bin
conf
include
jmods
legal
lib
release
```

Ao listar os arquivos do diretório `jmods`, verá que todos os módulos estão lá em arquivos com a extensão `.jmod` — um formato novo do JDK, otimizado para módulos, que discutiremos adiante.

```
java.activation.jmod
java.base.jmod
java.compiler.jmod
java.corba.jmod
java.datatransfer.jmod
java.desktop.jmod
...
```

Sabendo disso, a primeira etapa para criar a imagem customizada será apontar para esse caminho. Podemos usar a variável `JAVA_HOME` como a seguir:

```
--module-path $JAVA_HOME/jmods
```

O próximo passo será informar os módulos que devem ser

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

adicionados nessa imagem, separando por vírgula. Em nosso caso, só precisaremos do `java.base` e `jdk.incubator.httpclient`, portanto:

```
--add-modules java.base,jdk.incubator.httpclient
```

Cuidado com os espaços

Não podem existir espaços entre o nome de um módulo e outro. Se isso acontecer, você receberá um erro parecido com:

```
Error: invalid argument: --output,
small-JRE,
jdk.incubator.httpclient
```

Por fim, só precisamos informar o nome da imagem que será criada. Esse pode ser qualquer nome de sua preferência, em nosso caso, vamos chamá-la de `small-JRE`:

```
--output small-JRE
```

O comando completo ficará assim:

```
jlink --module-path $JAVA_HOME/jmods \
    --add-modules java.base,jdk.incubator.httpclient \
    --output small-JRE
```

Experimente executá-lo no diretório de seu projeto. Se tudo correu bem, um diretório com o nome da imagem customizada será criado. Para executar o Java a partir dele, basta acessá-lo pela pasta `bin` como a seguir:

```
small-JRE/bin/java
```

Nada complicado, não é? Podemos garantir que ela contém

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

apenas as duas dependências que foram definidas usando o `--list-modules`:

```
small-JRE/bin/java --list-modules

java.base@9
jdk.incubator.httpclient@9
```

Perfeito, temos a nossa pequena JRE.

Executando o projeto

Para executar o projeto utilizando essa nossa imagem de execução customizada, vamos fazer exatamente como antes, mudando apenas o local do Java que será utilizado para o `small-JRE/bin/java`.

```
small-JRE/bin/java \
    --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

Sucesso. Nada muda na aplicação, o comportamento será o mesmo:

```
Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
3 - Introdução e boas práticas em UX Design
4 - Conhecendo o Adobe Photoshop CS6
5 - Edição e organização de fotos com Adobe Lightroom
6 - Métricas Ágeis
7 - Scrum: Gestão ágil para projetos de sucesso
8 - Test-Driven Development
9 - Direto ao Ponto
10 - Guia da Startup
11 - O Mantra da Produtividade
12 - Java SE 7 Programmer I
13 - Java SE 8 Programmer I
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
14 - Zend Certified Engineer

Digite o número do livro que quer comprar:
```

A imagem tem apenas o necessário para executar o nosso projeto. Nem mais, nem menos.

E se algo ficar de fora?

Um detalhe importante aqui é que imagens como essa que acabamos de criar **são usadas apenas em ambiente de execução**. O problema é que, com isso, já não temos uma integridade tão grande entre a etapa de compilação e a de execução como antes, uma vez que poderíamos esquecer de adicionar algum dos módulos essenciais para o funcionamento de nossa `bookstore`.

Vamos tentar forçar esse erro? Basta apagar a `small-JRE` criada e executar o comando novamente, deixando de fora o `httpclient`:

```
jlink --module-path $JAVA_HOME/jmods \
    --add-modules java.base \
    --output small-JRE
```

A imagem será criada como esperado.

Agora experimente executar o projeto a partir dela:

```
small-JRE/bin/java \
    --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

O resultado será um esperado erro de inicialização, informando que o módulo necessário para o `http` não foi encontrado.

```
Error occurred during initialization of boot layer
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    java.lang.module.FindException:
Module jdk.incubator.httpclient not found,
    required by br.com.casadocodigo.http
```

A mensagem é clara e nos ajuda a descobrir facilmente o que ficou de fora.

Apesar desse problema acontecer, perceba que **fomos alertados imediatamente ao tentar subir a aplicação**. Isso é muito melhor do que a abordagem anterior, do classpath, na qual o erro só aconteceria em tempo de execução.

Ainda assim, podemos evitar esse risco. Como? Criando a imagem a partir das definições do projeto, no lugar de adicionar cada um dos módulos manualmente como estamos fazendo.

Sim, isso é possível e bastante indicado, já que em projetos maiores seriam vários os módulos utilizados e a possibilidade de algum ficar de fora seria ainda maior — além do enorme trabalho de precisar adicionar um a um na linha de comando.

8.2 CRIANDO A IMAGEM A PARTIR DAS DEFINIÇÕES DO PROJETO

Cada um de nossos módulos tem o arquivo `module-info.java`, com toda a informação necessária para criarmos uma imagem apenas com o que eles precisam para o projeto. Podemos tirar proveito disso com uma mudança bem pequena no comando do `jlink`, que ficará assim:

```
jlink --module-path $JAVA_HOME/jmods:mods \
    --add-modules br.com.casadocodigo \
    --output JRE-bookstore
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Perceba que o `module-path` está recebendo também a pasta `mods`, que é onde ficam todos os módulos compilados da `bookstore`. Além disso, a opção `--add-modules` passa a receber apenas o nosso módulo principal, `br.com.casadocodigo`, no lugar de cada um dos outros módulos da plataforma.

Execute o comando para testar. A imagem `JRE-bookstore` será criada.

Dessa vez, ao utilizar o `--list-modules`, veremos que as dependências de nosso projeto estão junto com as da plataforma:

```
JRE-bookstore/bin/java --list-modules

br.com.casadocodigo
br.com.casadocodigo.domain
br.com.casadocodigo.http
br.com.casadocodigo.nf
java.base@9
jdk.incubator.httpclient@9
```

Agora temos uma garantia maior de que todas as dependências necessárias para execução do projeto estão presentes na imagem. Além disso, uma vez que os módulos do projeto já estão adicionados nessa imagem, não precisamos mais passar o `--module-path` para executar. O comando ficará assim:

```
JRE-bookstore/bin/java \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

8.3 ANÁLISE DE DEPENDÊNCIAS DOS MÓDULOS

Agora que trouxemos o mundo modular para nosso projeto, estamos criando imagens customizadas, e certamente vamos

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

utilizar outras bibliotecas (JARs, arquivos JMOD etc.). Assim, **entender e analisar as dependências usadas em cada ponto do código será uma necessidade constante**, tanto para saber se nosso projeto está com acoplamento maior do que deveria, como para sabermos o que é necessário ao criar ambientes de execução customizados.

O `jdeps`, uma ferramenta de linha de comando disponível desde o JDK 8, certamente será um grande aliado nesse trabalho. Em sua versão inicial, já era possível mostrar todas as dependências de uma classe como no exemplo a seguir.

```
jdeps mods/br.com.casadocodigo/br/com/casadocodigo/Main.class

Main.class -> java.base

  br.com.casadocodigo -> br.com.casadocodigo.data
  br.com.casadocodigo -> br.com.casadocodigo.model
  br.com.casadocodigo -> br.com.casadocodigo.nf.service
  br.com.casadocodigo -> java.io
  br.com.casadocodigo -> java.lang
  br.com.casadocodigo -> java.lang.invoke
  br.com.casadocodigo -> java.util
  br.com.casadocodigo -> java.util.function
  br.com.casadocodigo -> java.util.stream
```

Perceba que a ferramenta mostrou cada um dos pacotes que usamos explícita ou implicitamente na `Main.class`, da `bookstore`.

No JDK 9, foi adicionada a opção `--list-deps`. Executando o mesmo exemplo de antes, com essa nova opção, você receberá a lista de dependências que seu arquivo compilado possui diretamente.

```
jdeps --list-deps \
    mods/br.com.casadocodigo/br/com/casadocodigo/Main.class
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Para esse caso, o resultado será apenas:

```
java.base
```

Na classe `Books`, por exemplo, o módulo `httpclient` também será listado:

```
jdeps --list-deps \
    mods/br.com.casadocodigo.http/br/com/
casadocodigo/http/Books.class

java.base
jdk.incubator.httpclient
```

Faz sentido, já que é nessa classe específica que estamos usando o novo *HTTP/2 Client*.

Poderíamos fazer o mesmo para um JAR, ou mesmo no diretório `/mods` completo:

```
jdeps --list-deps mods/
```

O resultado será parecido, uma vez que só usamos esses mesmos dois módulos no projeto inteiro. Também deve aparecer um *unnamed module*, dependendo de seu build do JDK 9 instalado, pois o `jdeps` tenta procurar um arquivo `module-info` na raiz do diretório.

```
java.base
jdk.incubator.httpclient
unnamed module: bookstore/mods
```

Interessante, não é? E talvez ainda mais útil em uma representação gráfica, que pode ajudar muito em projetos maiores. O `jdeps` possui suporte para gerar arquivos `.dot`, que é uma linguagem abstrata de descrição de gráficos. Você pode fazer isso com a opção `--dot-output`, como a seguir:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

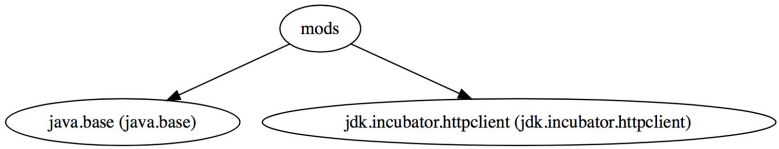
```
jdeps --dot-output . mods/
```

O arquivo `summary.dot` será criado com o seguinte conteúdo:

```
digraph "summary" {
  "mods" -> "java.base (java.base)";
  "mods" -> "jdk.incubator.httpclient (jdk.incubator.httpclient)"
;
}
```

Existem algumas ferramentas que nos ajudam a gerar o gráfico a partir dessa definição criada, como o *Graphviz*. Ele possui uma versão online, em http://www.webgraphviz.com.

Basta colocar o conteúdo desse arquivo `.dot` no campo de texto e clicar em `generate graph`.



E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

ENCONTRANDO JDK.INTERNALS

O `jdeps` também recebeu um suporte especial que nos ajuda a identificar rapidamente se em algum ponto do projeto dependemos direta ou indiretamente de implementações internas da linguagem que, nessa nova versão, estão fortemente encapsuladas no JDK.

Um exemplo? O `sun.misc.Unsafe`, mencionado anteriormente. Você pode executar o seguinte comando para saber se seu código, ou alguma das libs, usa o `Unsafe` ou alguma outra dessas implementações internas:

```
jdeps --jdk-internals mods
```

Se nenhum resultado for exibido, significa que ele não usa. Isso é um ótimo sinal. Algo apareceu? Experimente pesquisar se o componente pode ser atualizado ou migrado, uma vez que essas classes devem desaparecer em futuros releases da linguagem.

8.4 ENCONTRANDO DETALHES DOS MÓDULOS E DEPENDÊNCIAS VIA REFLECTION

Também é possível saber todos os detalhes de módulos e suas definições em tempo de execução, com uso da nova classe `Module` presente no pacote `java.lang.reflect`. Todo objeto agora possui um método `getModule`, que retornará uma

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

instância desse tipo.

```
Module module = book.getClass().getModule();
```

Para imprimir o nome do módulo que ele pertence, por exemplo, faremos:

```
System.out.println(module.getName());
```

O resultado será `br.com.casadocodigo.domain` , já que esse é o nome do módulo em que está a classe `Book` .

O método `getDescriptor` mostra, além do nome, todos os `requires` , `exports` e outras definições que podem existir dentro do módulo.

```
System.out.println(module.getDescriptor());
```

A saída será:

```
module {
  name: br.com.casadocodigo.domain,
    [mandated java.base (@9)],
    exports:
    [br.com.casadocodigo.model]
}
```

Além destes, existem métodos para retornar o *classloader*, *layer* (*profile*), entre outros. Os principais deles estão listados a seguir:

```
public final class Module {
    public String getName();
    public ModuleDescriptor getDescriptor();
    public ModuleLayer getLayer();
    public ClassLoader getClassLoader();
    public boolean canRead(Module target);
    public boolean isExported(String packageName);
}
```

Os tipos `ModuleDescriptor` , `ModuleLayer` e outros estão

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

presentes no pacote `java.lang`. Você pode passear por essas classes diretamente por esse link da documentação:

http://download.java.net/java/jdk9/docs/api/java/lang/Module.html

8.5 DOWNLOAD DAS JRES CRIADAS

Caso queira navegar por sua estrutura, as imagens de execução da JRE que foram criadas aqui no capítulo estão disponíveis na pasta deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

CAPÍTULO 9

EVOLUÇÃO DOS JARS NO JDK 9

Neste ponto, você já sabe como modularizar um projeto e conhece as principais mudanças nos processos de compilação e execução. Mas afinal, como empacotar e distribuir um código modular?

Nosso objetivo agora será entender o que muda nessa etapa de *packaging* do projeto, além da nova estrutura do *JAR* e dos *MRJARs* — que oferecem suporte às múltiplas versões da linguagem.

9.1 EMPACOTANDO PROJETOS MODULARES

Podemos empacotar e compartilhar os módulos de nossos projetos em JARs, de forma muito parecida com a que fazemos hoje em dia. As próprias IDEs certamente vão ajudar com esse trabalho, mas uma alternativa para fazê-lo pela linha de comando seria usando o `jar --create`, que agora possui pequenas diferenças em sua estrutura.

A primeira delas é que, no lugar de passar o *classpath* como antes, vamos usar a opção `-C` para informar o diretório onde está

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

o módulo já compilado. Para o módulo `domain`, por exemplo, faríamos:

```
jar --create \
  -C mods/br.com.casadocodigo.domain
```

Além disso, podemos utilizar a opção `--module-version` para definir uma versão para esse código que está sendo empacotado:

```
--module-version 1.0
```

Novo esquema de versões do Java

Por falar em versões, a linguagem definiu um formato oficial de versionamento que nos ajuda a diferenciar facilmente updates maiores, ou que envolvem segurança, dos demais releases. Se quiser saber mais, você encontra detalhes em:

http://openjdk.java.net/jeps/223

Para completar o comando, também precisamos do `--file` indicando o local onde o JAR deverá ser criado:

```
--file=mlib/br.com.casadocodigo.domain-1.0.jar
```

Vamos testar? Basta criar esse diretório `mlib`, onde vão ficar os JARs da `bookstore`:

```
mkdir mlib
```

E agora executar o comando completo, que criará uma biblioteca com a versão 1.0 do módulo `domain`:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
jar --create \
  --file=mlib/br.com.casadocodigo.domain-1.0.jar \
  --module-version 1.0 \
  -C mods/br.com.casadocodigo.domain .
```

Ao executar, o arquivo `br.com.casadocodigo.domain-1.0.jar` será criado dentro da pasta `mlibs`.

```
└─ mlib
   └── br.com.casadocodigo.domain-1.0.jar
```

Sua estrutura interna ficará exatamente assim:

```
mlib
└──br.com.casadocodigo.domain-1.0
   ├── META-INF
   │   └── MANIFEST.MF
   ├── br
   │   └── com
   │       └── casadocodigo
   │           └── model
   │               ├── Book.class
   │               └── Category.class
   └── module-info.class
```

Perceba que, por motivos de compatibilidade, esse JAR é parecido com o formato atual em todos os aspectos, com exceção do arquivo `module-info.class` que está presente em sua raiz.

MODULAR JAR FILES

Apesar de o tipo de arquivo e a estrutura serem praticamente a mesma, arquivos do tipo JAR que possuem o `module-info.class`, com as definições de um módulo, são conhecidos como *modular JAR files*. Esse termo é muito utilizado nas documentações da nova versão da linguagem.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Podemos fazer o mesmo para empacotar os módulos de nota fiscal e do `http` :

```
jar --create \
  --file=mlib/br.com.casadocodigo.nf-1.0.jar \
  --module-version 1.0 \
  -C mods/br.com.casadocodigo.nf .

jar --create \
  --file=mlib/br.com.casadocodigo.http-1.0.jar \
  --module-version 1.0 \
  -C mods/br.com.casadocodigo.http .
```

Agora, diferente dos demais, vamos usar a opção `--main-class` para empacotar o módulo principal como um executável, já que ele é o ponto de partida de nossa aplicação! A opção deve indicar o caminho para a classe `Main` , como a seguir:

```
--main-class=br.com.casadocodigo.Main
```

O comando completo ficará assim:

```
jar --create \
  --file=mlib/br.com.casadocodigo-1.0.jar \
  --module-version 1.0 \
  --main-class=br.com.casadocodigo.Main \
  -C mods/br.com.casadocodigo .
```

Ao executá-lo, teremos todos os módulos dentro do diretório `mlib` :

```
└─ mlib
   ├── br.com.casadocodigo.domain-1.0.jar
   ├── br.com.casadocodigo.http-1.0.jar
   ├── br.com.casadocodigo.nf-1.0.jar
   └── br.com.casadocodigo-1.0.jar
```

9.2 EXECUTANDO PROJETOS MODULARES

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Não há nada de especial na execução de um JAR modular. A estrutura será muito parecida com a que já estamos usando, com a diferença de que agora a opção `-p` deve indicar o diretório onde estão todos os módulos empacotados.

```
java -p <pasta-com-os-jars> -m <nome-do-modulo>
```

Sendo assim, em nosso projeto, o comando será:

```
java -p mlib -m br.com.casadocodigo
```

Com isso, o projeto deverá ser executado sem problemas.

Se você esqueceu de criar o JAR de algum dos módulos do projeto, certamente recebeu um erro parecido com esse:

```
Error occurred during initialization of boot layer
java.lang.module.FindException:
Module br.com.casadocodigo.http not found,
required by br.com.casadocodigo
```

A mensagem é bem intuitiva, e nesse exemplo indica que o módulo `http` não foi encontrado. Basta empacotar a dependência que faltou e executar o projeto novamente, que tudo deverá funcionar.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

ARQUIVOS .JMOD

O JKD 9 introduziu um novo tipo de arquivo, com extensão `.jmod`, que vimos ao listar todos os módulos do diretório de instalação do Java.

```
java.activation.jmod
java.base.jmod
java.desktop.jmod
...
```

Esse novo tipo é muito parecido com o JAR, mas pode ser usado quando estamos trabalhando com bibliotecas nativas da linguagem, além de compactar de uma forma muito eficiente as definições do módulo com suas configurações e seus recursos adicionais.

Apesar disso, é importante perceber que o JMOD **não é executável**, portanto pode ser utilizado apenas nas fases de compilação e link de um projeto. Para todos os outros casos, ainda devemos continuar usando arquivos do tipo JAR.

Há uma ferramenta chamada `jmod`, que ajuda na criação e definição de arquivos como esses. Se quiser mais detalhes sobre o *jmod* e como trabalhar com arquivos desse tipo, pode se interessar pelo link:

https://docs.oracle.com/javase/9/tools/jmod.htm

9.3 MULTI-RELEASE JAR FILES

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

O JDK 9 introduziu uma possibilidade que eu sinceramente esperei muito para ver na linguagem, chamada *multi-release JAR files*. Agora você pode ter **um único JAR que suporta diferentes versões específicas do Java**.

Perceba que a nossa classe `Main.java`, para exibir a lista de livros com seu *index* no começo da linha, utiliza um `IntStream#range` que só existe a partir do JDK 8 com a introdução da API de *Streams*:

```
IntStream.range(0, books.size())
  .forEach(i -> {
    System.out.println(i + " - " + books.get(i).getName());
});
```

Ao tentar compilar essa classe com o JDK 7, um erro de compilação aconteceria, informando que o pacote de *streams* não existe. Para que funcione, precisaríamos substituir esse `IntStream` por algum recurso compatível da versão anterior, como um `for` tradicional com *index*:

```
for(int i = 0; i < books.size(); i++) {
    System.out.println(i + " - " + books.get(i).getName());
}
```

O problema é que teríamos de implementar as duas versões desse código em diferentes locais do repositório do projeto, além de sempre criar um JAR compilado para cada uma das versões suportadas. Como consequência, muito provavelmente acabaríamos optando por não utilizar as novas APIs e recursos, para evitar o enorme trabalho de manter e distribuir as diferentes versões — ou pior ainda, uma quebra de compatibilidade.

Esse é um problema muito comum para quem mantém bibliotecas e frameworks, que muitas vezes deixam de evoluir

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

porque precisam continuar dando suporte para a esmagadora maioria dos usuários e projetos que usam versões anteriores.

Com o JDK 9 e o novo conceito de JAR para diferentes releases, você finalmente pode fazer isso de uma forma simples. Um mesmo JAR pode sobrescrever implementações para as diferentes versões que você precisa suportar da linguagem.

Um JAR com múltiplas versões (ou *MRJAR*, como é chamado) tem a seguinte linha no arquivo `MANIFEST.MF` :

```
Multi-Release: true
```

Além disso, ele vai possuir uma estrutura um pouco diferente. Vamos ver isso na prática! Lembra como criamos o JAR do módulo principal da bookstore? O comando completo foi esse:

```
jar --create \
  --file=mlib/br.com.casadocodigo-1.0.jar \
  --module-version 1.0 \
  --main-class=br.com.casadocodigo.Main \
  -C mods/br.com.casadocodigo .
```

Para criar um *MRJAR*, que sobrescreve a classe `Main` quando estivermos usando o JDK 8, o comando ficará assim:

```
jar --create \
  --file=mlib/br.com.casadocodigo-1.0.jar \
  --module-version 1.0 \
  --main-class=br.com.casadocodigo.Main \
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  -C diretorio-java-8 . --release 9 \
  -C mods/br.com.casadocodigo .
```

Perceba que a única coisa que mudou foi a linha a seguir:

```
-C diretorio-java-8 . --release 9
```

Ela está dizendo ao programa `jar` que, em um diretório chamado `diretorio-java-8`, estão as classes compiladas com alguma outra versão que sobrescreve o código do Java 9.

Se você tentar executar o comando dessa forma, o resultado seria:

```
diretorio-java-8: no such file or directory
```

Claro, em nenhum momento criamos esse diretório! Precisamos criar.

```
mkdir -p diretorio-java-8/br/com/casadocodigo
```

Perceba que, no comando anterior, eu já criei inclusive o pacote base do projeto. A estrutura desse diretório ficou assim:

```
diretorio-java-8
└── br
    └── com
        └── casadocodigo
```

Agora que temos ele preparado, vamos executar o comando de gerar o JAR novamente.

```
jar --create \
  --file=mlib/br.com.casadocodigo-1.0.jar \
  --module-version 1.0 \
  --main-class=br.com.casadocodigo.Main \
  -C diretorio-java-8 . --release 9 \
  -C mods/br.com.casadocodigo .
```

Dessa vez funciona, e o JAR será criado.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Mas qual classe ele está sobrescrevendo? Nenhuma. Não definimos isso em lugar algum, só dissemos que o diretório poderia ter classes que sobrescrevem alguma implementação atual feita em com o Java 9.

Apesar disso, experimente descompactar o JAR criado e olhar a sua estrutura interna. Ele estará assim:

```
mlib/br.com.casadocodigo-1.0
├── META-INF
│   ├── MANIFEST.MF
│   └── versions
│       └── 9
│           ├── br
│           │   └── com
│           │       └── casadocodigo
│           │           └── Main.class
│           └── module-info.class
└── br
    └── com
        └── casadocodigo
```

Perceba que agora existe uma pasta `versions/9`, dentro do diretório `META-INF`, com todos o código compilado em Java 9. Legal, não é? É aí que vão ficar os arquivos compilados em outras versões da linguagem.

Para ver isso na prática, vamos criar uma classe `Main.java`, dentro de `diretorio-java-8/br/com/casadocodigo`, com um simples `System.out` imprimindo uma mensagem diferente de nossa implementação atual da classe `Main`, que executa o fluxo completo da bookstore.

```
package br.com.casadocodigo;

public class Main {

  public static void main(String ...args) {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    System.out.println("classe main do java 8!");
  }
}
```

Agora vem um detalhe importante, **precisamos compilar essa classe com o JDK 8!** Existem várias formas que você pode fazer isso, talvez a mais simples seja editando o `$JAVA_HOME` direto pela sua janela do terminal.

```
export JAVA_HOME='caminho-para-o-seu-java-8'
```

E em seguida compilando o arquivo com o `javac`, que agora estará usando a versão que você espera.

```
javac diretorio-java-8/br/com/casadocodigo/Main.java
```

Feito isso, o arquivo `Main.class` será criado e o diretório terá o seguinte conteúdo:

```
diretorio-java-8
└── br
    └── com
        └── casadocodigo
            ├── Main.class
            └── Main.java
```

Tudo pronto, não se esqueça de abrir uma nova janela do terminal ou exportar o `$JAVA_HOME` novamente, para voltar ao JDK 9.

Vamos gerar o JAR novamente. O comando é o mesmo:

```
jar --create \
  --file=mlib/br.com.casadocodigo-1.0.jar \
  --module-version 1.0 \
  --main-class=br.com.casadocodigo.Main \
  -C diretorio-java-8 . --release 9 \
  -C mods/br.com.casadocodigo .
```

Dessa vez, ao inspecionar sua estrutura, veremos algo

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

diferente!

```
mlib/br.com.casadocodigo-1.0
├── META-INF
│   ├── MANIFEST.MF
│   └── versions
│       └── 9
│           ├── br
│           │   └── com
│           │       └── casadocodigo
│           │           └── Main.class
│           └── module-info.class
└── br
    └── com
        └── casadocodigo
            ├── Main.class
            └── Main.java
```

Além do diretório de versões, com a implementação do Java 9, a classe `Main` está disponível na raiz do projeto. Ao executar o JAR com o Java 9, o código completo da bookstore será executado como antes. Ao executar esse mesmo JAR, com o Java 8, o `System.out` da classe compilada com essa versão específica será executado.

```
>java -version
java version "1.8.0_141"

>java -jar mlib/br.com.casadocodigo-1.0.jar
classe main do java 8!
```

Incrível, não é? Claro que a primeira vista isso pode parecer assustador, especialmente fazendo tudo manualmente como estamos na linha de comando. Certamente as IDEs e outras ferramentas vão facilitar esse trabalho no futuro.

9.4 PARA SABER MAIS

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Se quiser ver mais a respeito, no seguinte link há um exemplo de como essa implementação funcionaria junto com o Maven:

https://github.com/hboutemy/maven-jep238

A *AdoptOpenJDK* também tem um exemplo de como fica a estrutura de um projeto modular com Maven, para o gerenciamento de dependências e build.

https://github.com/AdoptOpenJDK/jdk9-jigsaw/tree/master/session-3-refactoring-migration/03_ServiceMonitor_migration_to_java_9

Em resumo cada módulo possui um arquivo `pom.xml`, com suas configurações e dependências específicas, e aponta para o `parent-pom` que define as configurações globais de todos eles. Você encontra mais detalhes sobre a especificação das mudanças e o status do suporte ao Jigsaw, que ainda não está completo, em:

https://cwiki.apache.org/confluence/display/MAVEN/Java+9+-+Jigsaw

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 10

LOGGING API

Se você mantém um sistema, independente do tamanho, certamente sabe a importância de ter *logs* pelo código. Foram inúmeras as vezes que eu consegui identificar e resolver problemas rapidamente pela existência de logs que nos passavam as informações necessárias para isso, ou ainda a situação contrária, na qual fiquei muito tempo para encontrar algo pequeno que seria facilmente identificado com uso apropriado deles.

Existem diversos frameworks, como o *Log4j* da Apache, que nos ajudam com esse trabalho. O próprio Java possui sua implementação `java.util.logging`, criada na versão 1.4 do JDK, mas que não ficou muito popular — mesmo sendo uma solução nativa e usada pelos componentes internos da plataforma.

Um problema muito comum é que cada biblioteca podia escolher um logger diferente para usar e, no fim, você acabava com várias implementações no classpath (`java.util.logging`, `log4j`, `commons-logging`, `logback` etc).

Cada uma dessas implementações poderia ter configurações diferentes, e APIs diferentes. Não existia padrão.

Para resolver, muitos projetos usam o SLF4J, *Simple Logging Facade for Java*, que servia como uma abstração para todas elas.

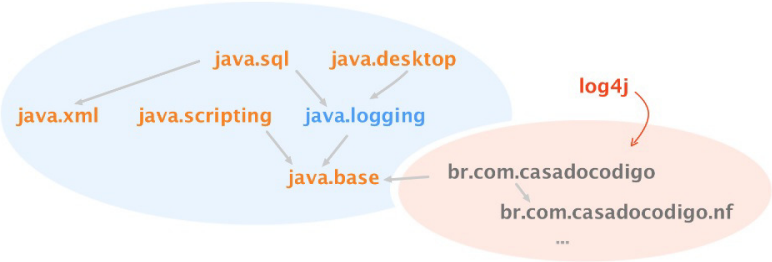
E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

No gerenciador de dependências de seu projeto, como por exemplo o Maven, você poderia excluir cada uma dessas implementações mantendo apenas o SLF4J, que fornecia as interfaces de cada uma dessas APIs no classpath como um truque, mas no final redirecionava para uma implementação única e com configuração única.

Você encontra detalhes sobre essa estratégia na documentação do SLF4J, em *Bridging legacy APIs*:

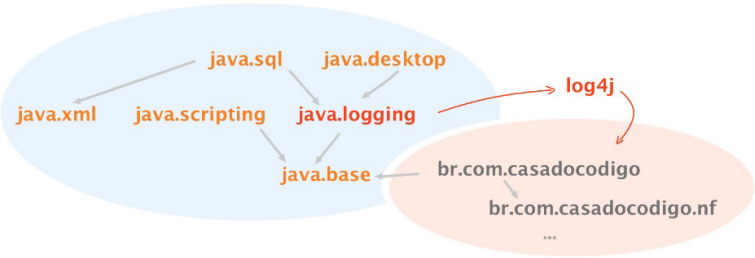
https://www.slf4j.org/legacy.html

Em nosso próprio projeto, poderíamos optar pelo uso do Log4j e, com isso, estaríamos utilizando uma API diferente do que é utilizada pela plataforma, que é o `java.util.logging` .



Essa foi a grande motivação por trás da nova API de *logging* do JDK 9, que lhe possibilita criar um provedor padrão de mensagens que poderá ser usado tanto em seu código como no do próprio JDK. O antigo `java.util.logging` continua existindo, mas evoluiu para que passe a usar a implementação de log de sua escolha, no lugar da solução padrão da plataforma.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com



Além disso, também nasceu uma nova opção de linha de comando que lhe permite investigar em um só lugar todas as informações de log da JVM — incluindo do novo *garbage collector* padrão, que será o assunto de um dos próximos capítulos. Isso é muito legal por uma série de motivos que você vai descobrir na prática, aplicando a solução na `bookstore`.

10.1 CRIANDO UM MÓDULO DE LOGS

Para explorar as possibilidades dessa nova API, vamos criar um novo módulo chamado `br.com.casadocodigo.logging`. O processo é o mesmo que já fizemos em todos os outros, começando pela criação de um diretório com seu nome e toda a estrutura de pastas do pacote base.

```
br.com.casadocodigo.logging
└── br
    └── com
        └── casadocodigo
            └── logging
```

Também precisamos criar o `module-info.java`, com sua definição inicial.

```
module br.com.casadocodigo.logging {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
}
```

Tudo pronto, temos toda a infraestrutura necessária para o módulo de `logging`.

```
br.com.casadocodigo.logging
├── br
│   └── com
│       └── casadocodigo
│           └── logging
└── module-info.java
```

Já podemos começar com a implementação do código.

Implementando System.Logger

Independente de escolher criar uma nova implementação ou usar alguma das bibliotecas existentes, é indicado que você **encapsule todo contato com esse código dentro de um pacote que ficará visível apenas para o próprio módulo**. Para isso, vamos criar a classe `CustomLogger`, dentro do pacote `br.com.casadocodigo.logging.impl`.

```
package br.com.casadocodigo.logging.impl;

public class CustomLogger {

}
```

Como mencionado, o grande segredo da nova API é que, a partir de mecanismo de busca que logo veremos, a própria solução atual do `java.util.logging` será capaz de localizar e utilizar a mesma implementação de logs do nosso projeto. Para que isso aconteça, precisamos usar uma estrutura em comum com a plataforma que foi definida na nova interface `java.lang.System.Logger`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
interface java.lang.System.Logger {

  String getName();

  boolean isLoggable(Level level);

  void log(Level, ResourceBundle, String, Throwable);

  void log(Level, ResourceBundle, String, Object...);

  <<< várias sobrecargas default do método log >>>
}
```

Vamos fazer com que nossa classe `CustomLogger` implemente essa interface e seus métodos abstratos. O esqueleto do código ficará assim:

```
package br.com.casadocodigo.logging.impl;

import java.lang.System.Logger;
import java.util.ResourceBundle;

public class CustomLogger implements Logger {

  @Override
  public String getName() {
    //...
  }

  @Override
  public boolean isLoggable(Level level) {
    //...
  }

  @Override
  public void log(Level level, ResourceBundle bundle,
    String msg, Throwable thrown) {

    //...
  }

  @Override
  public void log(Level level, ResourceBundle bundle,
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    String format, Object... params) {

    //...
  }
}
```

Repare que, apesar de o `Logger` estar no pacote `java.lang`, por ele ser uma **interface interna** da classe `System` foi necessário adicionar o `import`.

> Nesse momento, poderíamos baixar o JAR do log4j, ou algum outro framework, e delegar cada um desses métodos para a implementação específica da biblioteca. Vou evitar isso para não termos de desviar a atenção dos detalhes da nova API com alguma implementação que talvez não seja familiar para você, além de toda a burocracia de download e de suas configurações específicas.

O método `getName` define a forma como vamos nos referir ao log pelas demais partes do projeto. Podemos implementar retornando o texto `CustomLogger`, que é o nome dessa nossa classe de implementação.

```
@Override
public String getName() {
    return "CustomLogger";
}
```

O método `isLoggable` retorna um `boolean`, informando se você suporta ou não o nível recebido como parâmetro.

```
@Override
public boolean isLoggable(Level level) {
    //...
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
}
```

O `enum System.Logger.Level` define os diferentes tipos de log e seus pesos, que podem ser:

Nome	Peso
ALL	Integer.MIN_VALUE
TRACE	400
DEBUG	500
INFO	800
WARNING	900
ERROR	1000
OFF	Integer.MAX_VALUE

Poderíamos, por exemplo, delegar cada um dos possíveis níveis para a versão correspondente do framework de log que usamos no projeto. Com o Log4j, por exemplo, o código ficaria assim:

```
@Override
public boolean isLoggable(Level level) {
 switch (level) {
 case OFF:
   return log4j.isEnabled(org.apache.logging.log4j.Level.OFF);
 case TRACE:
   return log4j.isEnabled(org.apache.logging.log4j.Level.TRACE);
 case DEBUG:
   return log4j.isEnabled(org.apache.logging.log4j.Level.DEBUG);
 case INFO:
   return log4j.isEnabled(org.apache.logging.log4j.Level.INFO);
 case WARNING:
   return log4j.isEnabled(org.apache.logging.log4j.Level.WARN);
 case ERROR:
   return log4j.isEnabled(org.apache.logging.log4j.Level.ERROR);
 case ALL:
   return log4j.isEnabled(org.apache.logging.log4j.Level.ALL);
 default:
  throw new IllegalStateException("unknown level " + level);
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
 }
}
```

Para o nosso exemplo, podemos receber um `Level` no construtor da classe, definindo o nível que será utilizado, além de manter o `Level.INFO` como a opção padrão.

```
private final int severity;

public CustomLogger() {
  this.severity = Level.INFO.getSeverity();
}

public CustomLogger(Level level) {
  this.severity = level.getSeverity();
}
```

E agora, podemos verificar se sua `severity` é menor ou igual à do `Level` recebido. Dessa forma saberemos se a mensagem deve ou não ser exibida.

```
@Override
public boolean isLoggable(Level level) {
  return severity <= level.getSeverity();
}
```

Por fim, se o `level` for suportado, o método `log` fará um simples `System.out` , formatando cada um dos valores recebidos. Vamos também adicionar o nome da classe na mensagem, para ficar fácil diferenciar nossos logs da saída padrão do Java.

```
@Override
public void log(Level level, ResourceBundle
  bundle, String msg, Throwable thrown) {

   if (!isLoggable(level)) {
     return;
   }

   System.out.printf(
     "CustomLogger [%s]: %s - %s%n",
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    level, msg, thrown);
}
```

A outra sobrecarga, que recebe um *varargs* como parâmetro, pode fazer o mesmo e utilizar o `java.text.MessageFormat.format` para formatar o texto da mensagem de log.

```
@Override
public void log(Level level, ResourceBundle
  bundle, String format, Object... params) {

   if (!isLoggable(level)) {
     return;
   }

   System.out.printf(
        "CustomLogger [%s]: %s%n", level,
        java.text.MessageFormat.format(format, params));
}
```

A implementação é extremamente simples, mas mesmo assim você não precisa se preocupar tanto com ela. O ideal é que você apenas delegue toda essa responsabilidade para as bibliotecas existentes, como veremos adiante.

Com isso, a classe `CustomLogger` completa ficará assim:

```
package br.com.casadocodigo.logging.impl;

import java.lang.System.Logger;
import java.util.ResourceBundle;

public class CustomLogger implements Logger {

  private final int severity;

  public CustomLogger() {
      this.severity = Level.INFO.getSeverity();
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  public CustomLogger(Level level) {
      this.severity = level.getSeverity();
  }

  @Override
  public String getName() {
      return "CustomLogger";
  }

  @Override
  public boolean isLoggable(Level level) {
      return severity <= level.getSeverity();
  }

  @Override
  public void log(Level level, ResourceBundle
    bundle, String msg, Throwable thrown) {

     if (!isLoggable(level)) {
          return;
       }

    System.out.printf(
       "CustomLogger [%s]: %s - %s%n",
      level, msg, thrown);
  }

  @Override
  public void log(Level level, ResourceBundle
    bundle, String format, Object... params) {

     if (!isLoggable(level)) {
          return;
     }

     System.out.printf(
          "CustomLogger [%s]: %s%n", level,
          java.text.MessageFormat.format(format, params));
  }
}
```

Não deixe de compilar o novo módulo para garantir que, por enquanto, tudo funciona.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
javac -d mods/br.com.casadocodigo.logging \
  --module-path mods \
  src/br.com.casadocodigo.logging/module-info.java \
  $(find src/br.com.casadocodigo.logging -name "*.java")
```

Aparentemente tudo está ok! Já escrevemos a maior parte da nossa implementação, mas o trabalho ainda não está totalmente pronto. De alguma forma, precisaremos **ensinar à JVM como localizar essa nossa classe**, para que seja usada tanto no código da `bookstore` como no do próprio JDK.

Para fazer isso, vamos herdar de uma nova classe abstrata chamada `System.LoggerFinder`, que usa o padrão de *service loaders* introduzido no JDK 6.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

SERVICELOADER API

Talvez esse seja um termo novo para você, mas provavelmente já usou — ainda que indiretamente — ou ouviu falar dos *service loaders* ao usar abstrações de algum protocolo de banco de dados, como *JDBC*, ou qualquer outro tipo de serviço. O conceito é extremamente amplo, motivado pelas práticas de injeção de dependências e inversão de controle, mas pode ser resumido de uma forma simples: para exibir uma mensagem de log, por exemplo, você não quer saber como e onde é criado um logger, mas apenas saber **usar** a versão disponível no projeto.

No lugar de espalhar a regra, ela pode e deve ficar completamente encapsulada em um ponto específico do código, isolado do restante. A API de `ServiceLoader` tem um conjunto de interfaces (e classes abstratas) para ajudá-lo com esse trabalho:

https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html

Implementando System.LoggerFinder

A classe `System` possui um novo método `getLogger` que, dado o nome da implementação, retorna a instância construída com uso dos *service loaders*. Na prática, para usar o `CustomLogger` que acabamos de implementar na `bookstore`, faremos:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
Logger logger = System.getLogger("CustomLogger");
```

> Eu, particularmente, não gosto muito do uso de Strings dessa forma, por serem delicadas e erros bobos de digitação causarem o não funcionamento — como o texto de uma String não é compilado, erros passam facilmente despercebidos. Por outro lado, isso poderia ser facilmente revertido com uso de constantes ou uma configuração isolada.

A abordagem em si é bastante interessante, evitando expor os pacotes da implementação que ficam muito bem encapsulados e desacoplados do restante do projeto. Para que funcione, vamos criar uma classe `CustomLoggerFinder` que herda do `System.LoggerFinder`, utilizado internamente pelo `System.getLogger` para ensinar como a classe pode encontrar as possíveis implementações.

```
package br.com.casadocodigo.logging.impl;

import java.lang.System.Logger;
import java.lang.System.LoggerFinder;

public class CustomLoggerFinder extends LoggerFinder {

  @Override
  public Logger getLogger(String name, Module module) {
     //...
  }
}
```

Seu único método, `getLogger`, recebe uma String representando o nome da implementação e o módulo que a está

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

solicitando. Poderíamos, por exemplo, retornar uma implementação diferente para módulos específicos de nosso projeto ou do próprio JDK.

Além disso, projetos com várias implementações de log poderiam escolher a implementação que seria usada a partir de seu nome, como no exemplo:

```
if ("log4j".equals(name)) {
    return new Log4j();
}
if ("sl4j".equals(name)) {
    return new Sl4j();
}
```

Esse parâmetro `name` também poderia ser usado para um mesmo log, mas com nomes diferentes indicando o level que deve ser utilizado, ou qualquer outro detalhe de sua implementação.

Em nosso caso, que temos uma única implementação e queremos usá-la globalmente, vamos sempre retornar um `CustomLogger` quando o produtor `CustomLoggerFinder` estiver ativo e, por padrão, o `Level.INFO` do construtor vazio será usado.

```
package br.com.casadocodigo.logging.impl;

import java.lang.System.Logger;
import java.lang.System.LoggerFinder;

public class CustomLoggerFinder extends LoggerFinder {

  @Override
  public Logger getLogger(String name, Module module) {

    System.out.println(
      "CustomLoggerFinder: [name=" + name
      + ", module=" + module.getName() + "]");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    return new CustomLogger();
  }
}
```

Agora só precisamos registrar esse *service* na definição do nosso módulo de *logging*. Podemos fazer isso com uso de um novo modificador, `provides... with` , no arquivo `module-info.java` :

```
module br.com.casadocodigo.logging {

  provides
    java.lang.System.LoggerFinder
  with
    br.com.casadocodigo.logging.impl.CustomLoggerFinder;
}
```

Isso significa que, sempre que o módulo `br.com.casadocodigo.logging` estiver presente, qualquer lugar que precise do `System.LoggerFinder` receberá a nossa implementação `br.com.casadocodigo.logging.impl.CustomLoggerFinder` no lugar. Isso inclui os módulos da plataforma.

Vale reforçar que **em nenhum momento estamos expondo nossa implementação interna**. Não há nenhuma declaração de `exports` no módulo, portanto, a única forma de acessar a implementação é a partir do `System.getLogger` . E qual a vantagem disso?

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Esse baixo acoplamento é excelente para manutenção e evolução da implementação no futuro. Poderíamos mudar o `CustomLogger` do `System.out.printf` para o log4j, por exemplo, sem tocar em nenhum outro ponto do projeto além do próprio módulo de log.

Com isso, o código está pronto. Podemos compilar a versão final.

```
javac -d mods/br.com.casadocodigo.logging \
  --module-path mods \
  src/br.com.casadocodigo.logging/module-info.java \
  $(find src/br.com.casadocodigo.logging -name "*.java")
```

Agora, a estrutura completa do módulo de *logging* fica assim:

```
br.com.casadocodigo.logging
 ├── br
 │   └── com
 │       └── casadocodigo
 │           └── logging
 │               └── impl
 │                   ├── CustomLogger.java
 │                   └── CustomLoggerFinder.java
 └── module-info.java
```

Adicionando logs na bookstore

Agora que implementamos o suporte a logs em nosso projeto, podemos substituir alguns daqueles `System.out` s que utilizamos na classe `Main` por logs, que, diferente deles, tem uma semântica muito maior e diferentes níveis.

Com isso podemos escolher, de acordo com nossa necessidade, se queremos mostrar logs detalhados com informações importantes para quando estamos desenvolvendo o projeto, como o `TRACE` , `DEBUG` e `ERROR` , ou apenas logs de `INFO` e `WARNING`

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

que podem ser relevantes para todos os usuários.

Já vimos que, para fazer isso, podemos utilizar o `System.getLogger` como a seguir:

```
Logger logger = System.getLogger("CustomLogger");
```

Em nosso caso, a String `CustomLogger` passada como parâmetro é meramente informativa, deixando explícito que estamos usando essa implementação, mas seria desnecessária já que estamos sempre retornando o `CustomLogger`.

Vamos adicionar um log do tipo `TRACE`, na primeira e última linha do método, avisando que a aplicação está começando e terminando. Também podemos substituir o `System.err` de dentro do bloco `catch` por um log `ERROR`, e o `System.out` com a informação do livro selecionado por um log `INFO`.

O código da classe `Main.java` poderá ficar assim:

```
package br.com.casadocodigo;

import br.com.casadocodigo.http.*;
//...
import java.lang.System.Logger;
import java.lang.System.Logger.Level;

public class Main {

 public static void main(String ...args) {

   Logger logger = System.getLogger("CustomLogger");

   logger.log(Level.TRACE, "Iniciando a execução da bookstore");

   //...

   System.out.print("\nDigite o número do livro que quer comprar:
 ");
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
   try {
     int number = scanner.nextInt();
     //...

     logger.log(Level.INFO, "O livro escolhido foi:"+ book.getNam
e());

     //...

   } catch(Exception e) {
      logger.log(Level.ERROR, "Ops, aconteceu um erro: "+ e);
   }

   logger.log(Level.TRACE, "Terminando a execução da bookstore");
 }
}
```

Vamos executar? Lembre de compilar o módulo que foi modificado primeiro.

```
javac -d mods/br.com.casadocodigo/ \
    --module-path mods \
    src/br.com.casadocodigo/module-info.java \
    $(find src/br.com.casadocodigo/ -name "*.java")
```

Agora sim, podemos experimentar:

```
java --module-path mods \
    -m br.com.casadocodigo/br.com.casadocodigo.Main
```

O resultado será:

```
CustomLoggerFinder:
[name=CustomLogger, module=br.com.casadocodigo]

Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
...

Digite o número do livro que quer comprar: 2
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
...
CustomLogger [INFO]:
O livro escolhido foi: Java 8 Prático
```

Os `TRACE` s não aparecem, pois criamos o `CustomLogger` com o construtor padrão. Experimente agora mudar a classe `CustomLoggerFinder` , criando o *logger* com `Level.ALL` .

```
return new CustomLogger(Level.ALL);
```

Não esqueça de importar o enum `System.Logger.Level` .

> Certamente o uso de um `.properties` para definir o level, como fazem os frameworks, seria mais adequado. O nosso objetivo aqui não é focar na implementação do código, assim como você não deve fazer em seus projetos — a não ser que você queira criar um novo framework de logs, suprindo alguma necessidade que os atuais não fazem.

Agora compile o módulo de *logging* e execute o projeto novamente:

```
CustomLoggerFinder:
[name=CustomLogger, module=br.com.casadocodigo]

CustomLogger [TRACE]:
Iniciando a execução da bookstore

Lista de livros disponíveis

0 - Desbravando Java e Orientação a Objetos
1 - Explorando APIs e bibliotecas Java
2 - Java 8 Prático
...
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
Digite o número do livro que quer comprar: 2
...
CustomLogger [INFO]:
O livro escolhido foi: Java 8 Prático

...
CustomLogger [TRACE]:
Terminando a execução da bookstore
```

Agora sim, todos os logs estão lá.

Um detalhe interessante é que, em nenhum momento, precisamos informar que nosso projeto usa o `LoggerFinder`, que estamos fabricando nesse módulo de *logging*. Isso acontece porque o `java.base` já declara isso para você, com uma instrução especial do `module-info`:

```
java -d java.base

<<< um monte de coisas >>>
uses java.lang.System$LoggerFinder
```

Em outras palavras, de um lado fazemos o `provides... with` como no módulo de *logging* e, do outro, utilizamos o `uses` com o nome do *service* que deve ser consumido. Poderíamos ter feito explicitamente em nosso `module-info.java` principal, como por exemplo:

```
module br.com.casadocodigo {

    requires br.com.casadocodigo.nf;
    requires br.com.casadocodigo.http;
    requires br.com.casadocodigo.domain;

    uses java.lang.System$LoggerFinder;
}
```

Mas isso seria desnecessário, já que ele está presente implicitamente em todos os tipos de projeto.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

ESTOU USANDO A JRE-BOOKSTORE, E NENHUM LOG APARECEU!

Se você estiver usando a imagem customizada da JRE que criamos especialmente para a `bookstore`, nenhum dos logs realmente aparecerá. Isso acontece porque ela foi compilada antes de o novo módulo existir, portanto, certamente não terá essa definição preparada.

Uma alternativa seria recriar a imagem, usando o `jlink`. Um detalhe importante é que, como não há relação direta entre os módulos de nosso projeto e o de `logging`, ele precisa ser passado como um parâmetro extra da opção `--add-modules`, como a seguir:

```
jlink --module-path $JAVA_HOME/jmods:mods \
  --add-modules br.com.casadocodigo,br.com.casadocodigo.logg
ing \
  --output JRE-bookstore
```

Uma outra opção seria adicionar o `requires br.com.casadocodigo.logging` no `module-info` do módulo principal do projeto, mesmo sem que nenhum pacote e código seja exportado por ele. Ao recriar a imagem, se essa relação estiver explícita, o módulo de `logging` também seria adicionado.

E quanto aos logs do JDK?

Ok, os logs funcionam em nossa `bookstore`, mas lá no início eu tinha dito que **esse mesmo componente seria utilizado para as classes internas da plataforma.** Vamos ver se isso é verdade?

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Podemos forçar um exemplo com aquela *HttpURLConnection API* que falamos no capítulo de HTTP, pois ela tem vários logs do tipo `TRACE` declarados. Experimente adicionar o seguinte trecho de código no início da classe `Main`:

```
try {
    new URL("https://google.com")
        .openConnection().connect();
} catch (final IOException e) {
    logger.log(Level.ERROR, "Não consegui conectar");
}
```

Compile e execute o módulo. Você verá que, além dos nossos, aparecerão diversos logs do `HttpURLConnection`:

```
CustomLoggerFinder:
[name=CustomLogger, module=br.com.casadocodigo]

CustomLogger [TRACE]:
Iniciando a execução da bookstore

CustomLoggerFinder:
[name=HttpURLConnection, module=java.base]

CustomLogger [TRACE]:
ProxySelector Request for https://google.com/

CustomLogger [TRACE]:
Looking for HttpClient
for URL https://google.com
and proxy value of DIRECT

CustomLogger [TRACE]:
Creating new HttpsClient
with url:https://google.com
and proxy:DIRECT with connect timeout:-1

CustomLogger [TRACE]: Proxy used: DIRECT

...
CustomLogger [TRACE]: Terminando a execução da bookstore
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Legal, não é? Assim conseguimos ter controle total de todos os logs da aplicação. Apesar disso, é preciso tomar muito cuidado para o excesso de informações não sobrecarregar seus servidores e dificultar a análise desses dados.

10.2 LOGS DA JVM E GARBAGE COLLECTOR

Uma nova opção `-Xlog` foi adicionada no JDK 9, para que possamos acessar de uma forma única todos os tipos de log da JVM. Sua sintaxe completa é um pouco complicada, mas não é tão assustadora assim. Quer ver?

Experimente executar a aplicação novamente, mas agora com a opção extra `-Xlog` sem nenhum outro parâmetro.

```
java --module-path mods \
  -Xlog \
  -m br.com.casadocodigo/br.com.casadocodigo.Main
```

Umas 80 mil linhas de log devem ter saltado em seu terminal! Perceba que nelas existem todas as informações da JVM e *Garbage Collector*, como qual a implementação que está sendo utilizada:

```
[0.018s][info][gc] Using G1
```

Ou o local da sua instalação do JDK, de onde as classes estão sendo carregadas:

```
[0.009s][info][class,path]
bootstrap loader class
path=../jdk-9.jdk/Contents/Home/lib/modules
```

Diversas outras informações são exibidas, como quantidade de memória e tempo consumido, além de qual classe e de onde ela está sendo carregada:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
[0.021s][info][class,load]
java.lang.Object source: jrt:/java.base

[0.021s][info][class,load]
java.io.Serializable source: jrt:/java.base

[0.021s][info][class,load]
java.lang.Comparable source: jrt:/java.base
```

Você pode usar a opção `-Xlog:help` para entender melhor como filtrar e ajustar o nível de detalhes exibidos por esse log unificado. Entre as principais alternativas, um exemplo seria usar a opção `-Xlog:gc` para mostrar apenas os logs do *Garbage Collector*:

```
-Xlog:gc
```

Também podemos definir que queremos apenas os logs de *debug*, por exemplo:

```
-Xlog:gc=debug
```

E que o conteúdo deve ser impresso em um arquivo separado, no lugar do output padrão no terminal:

```
-Xlog:gc=debug:file=gc.txt
```

É muito conveniente ter acesso a todas essas informações, de uma forma simples e unificada.

Você encontra mais detalhes sobre os diferentes parâmetros desses logs unificados da JVM no link:

http://openjdk.java.net/jeps/158

10.3 DOWNLOAD DO PROJETO COMPLETO

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Se quiser, você pode baixar ou passear pelo código completo do módulo de `logging` , que está disponível na pasta deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 11

STACK-WALKING API

Alguns anos atrás, durante o desenvolvimento de um trecho delicado de um framework web, esbarrei em um problema grande de *memory leak*. Em algum ponto, meu código não liberava memória e, após algum tempo executando, as aplicações consumiam muito mais recursos do que precisavam e apresentavam uma performance precária. Como todo grande problema, o motivo não era nada evidente.

Foi então que me lembrei de uma sugestão que o Victor Harada, um amigo desenvolvedor, tinha dado algum tempo atrás quando estávamos tentando descobrir exatamente por quais pontos do código um controller do sistema em que trabalhávamos passava. A sugestão era dar um `new Exception().printStackTrace()`, assim toda stack seria exibida.

Como uma variação disso, o que eu fiz foi subir duas máquinas em um loop de requisições, uma com a versão antiga e estável, e outra com a nova implementação problemática. Usei então o `getStackTrace` para agrupar e somar a quantidade de referências que aparecia para cada classe.

Poucos segundos depois, a diferença na Stack das duas

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

aplicações era gritante, e o motivo do problema bastante evidente. O código, escrito no Java 7 da época, fazia algo parecido com:

```
Map<String, Integer> map = new HashMap<>();

StackTraceElement[] stacks =
      new Throwable().getStackTrace();

for(StackTraceElement element: stacks){

  String className = element.getClassName();

  int count = 1;

  if (map.containsKey(className)) {
    count = map.get(className);
    count++;
  }
  map.put(className, count);
}
```

E acredite quando eu digo que essa é uma versão muito mais simples do que a original, em que eu fazia vários *groupings* pelos métodos e linhas específicas de cada classe.

Hoje consigo pensar em um milhão de formas diferentes de investigar e resolver esse mesmo problema, utilizando diversas ferramentas de análise como o *New Relic*, *JProfiler*, *JMeter* ou a própria *Visual VM* do JDK. Na época, apesar de frustrado pelas várias limitações que encontrei, me senti um verdadeiro hacker.

O maior problema que encontrei nessa forma de passear e investigar as Stacks foi que a API era extremamente anêmica; você precisa resolver tudo de forma manual e próxima ao procedural. Além disso, não é nada eficiente. Mesmo precisando de apenas uma ou duas das últimas stacks, esse método `getStackTrace` ou algumas de suas variações fazem com que a máquina virtual

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

capture de forma gulosa toda a informação que representa a stack completa, o que é longe de ser o ideal.

Você também deve perceber que o `StackTraceElement` do retorno possui apenas os nomes da classe e método, como um texto, e não instâncias do tipo `Class` com as demais informações que podem ser necessárias. Para contornar isso, você precisaria usar o `SecurityManager`, que tem um método `protected getClassContext` para carregar essa informação.

Essas são algumas das muitas falhas desse modelo, que transforma uma necessidade simples em ciência espacial. Neste capítulo, vamos experimentar a nova API de *Stack-Walking*, que resolve todos esses problemas de forma performática e elegante, tirando proveito dos poderosos recursos da API de Streams.

11.1 NOVO MÓDULO DE TRACKING

Para que a diferença e vantagens da nova API fiquem claras na prática, vamos criar um novo módulo na `bookstore`, chamado `br.com.casadocodigo.tracking`, com a responsabilidade de encapsular a implementação de um logger com informações de rastreamento da stack. A estrutura será assim:

```
br.com.casadocodigo.tracking
 ├── br
 │   └── com
 │       └── casadocodigo
 │           └── tracking
 │
 └── module-info.java
```

E o arquivo `br.com.casadocodigo.tracking.module-info.java`:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
module br.com.casadocodigo.tracking {

}
```

O objetivo é prover uma interface simples, permitindo exibir as informações da stack, a partir de um ponto específico do código. Ao usar, queremos fazer algo parecido com:

```
Stack.logDebugTrace();
```

Para isso, podemos criar a classe `Stack`, com esse único método estático:

```
package br.com.casadocodigo.tracking;

public class Stack {

  public static void logDebugTrace() {
    //..
  }
}
```

O método `logDebugTrace` pode tirar proveito de nossa implementação global de log, portanto, vamos usar o `System#getLogger` assim como fizemos na classe `Main`. Como vimos, ele cuidará de buscar o *service* que o módulo de `logging` está provendo, sem exigir nenhum acoplamento do nosso lado.

Sua implementação deve mostrar a linha, o método e a classe de cada elemento da stack, por exemplo:

```
import java.lang.System.Logger;
import java.lang.System.Logger.Level;
...

public static void logDebugTrace() {

  Logger logger = System.getLogger("CustomLogger");

  logger.log(
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
      Level.TRACE,
      "linha X do método Y na classe Z"
  );
}
```

> Vale lembrar de que, nesse ponto, não precisamos declarar nenhuma relação entre os módulos de *tracking* e *logging*. Em outras palavras, você não precisa (e nem deve) declarar o `requires br.com.casadocodigo.logging` no `module-info` de nenhum dos módulos. **Isso acontece de forma implícita e completamente desacoplada.**

Para conseguir essas informações, vamos começar usando o legado `Throwable#getStackTrace`, iterando e exibindo os dados de cada novo elemento:

```
for(StackTraceElement element: stacks) {

  String className = element.getClassName();
  String methodName = element.getMethodName();
  int lineNumber = element.getLineNumber();

  // log
}
```

Ao final, nossa primeira versão da classe `Stack` deve ficar assim:

```
package br.com.casadocodigo.tracking;

import java.lang.System.Logger;
import java.lang.System.Logger.Level;

public class Stack {

 public static void logDebugTrace() {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
    Logger logger = System.getLogger("CustomLogger");

    StackTraceElement[] stacks =
         new Throwable().getStackTrace();

    for(StackTraceElement element: stacks) {

       String className = element.getClassName();
       String methodName = element.getMethodName();
       int lineNumber = element.getLineNumber();

       logger.log(Level.TRACE,
             " linha "
           + element.getLineNumber()
           + " do método "
           +  element.getMethodName()
           + " na classe "
           + element.getClassName());
    }
   }
}
```

Vamos testar? Um bom ponto do projeto para fazer isso é no módulo de notas fiscais, que envolve diferentes classes e threads daquele fluxo reativo. Podemos colocar o `Stack#logDebugTrace` dentro do bloco *catch* do método `emit`, no `WSPrefeitura`, e forçar uma exception:

```
...
import br.com.casadocodigo.tracking.*;

public class WSPrefeitura {

  public static void emit(NF nf) {
    try {
      //...
      throw new RuntimeException();
    } catch(Exception e) {
      System.out.println("falha ao emitir a nf");
      Stack.logDebugTrace();
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
}
```

Agora experimente compilar os dois módulos.

O de *tracking* funciona:

```
javac -d mods/br.com.casadocodigo.tracking \
  --module-path mods \
  src/br.com.casadocodigo.tracking/module-info.java \
  $(find src/br.com.casadocodigo.tracking/ -name "*.java")
```

Mas o de notas não!

```
javac -d mods/br.com.casadocodigo.nf \
  --module-path mods \
  src/br.com.casadocodigo.nf/module-info.java \
  $(find src/br.com.casadocodigo.nf/ -name "*.java")
```

A mensagem de erro nos lembra do que faltou de forma clara:

```
error: package br.com.casadocodigo.tracking
is not visible
import br.com.casadocodigo.tracking.*;
                            ^
(package br.com.casadocodigo.tracking
is declared in module br.com.casadocodigo.tracking,
but module br.com.casadocodigo.nf does not read it)
```

Vale lembrar: a partir do JDK 9, **não basta ser público para ser acessível**. Para que o pacote de *tracking* fique acessível, o módulo `br.com.casadocodigo.tracking` precisa declarar isso explicitamente com uso do `exports`:

```
module br.com.casadocodigo.tracking {
  exports br.com.casadocodigo.tracking;
}
```

E ainda assim, o módulo de notas fiscais também precisa declarar essa dependência com o `requires`:

```
module br.com.casadocodigo.nf {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  exports br.com.casadocodigo.nf.service;
  requires static br.com.casadocodigo.domain;
  requires br.com.casadocodigo.tracking;
}
```

Agora sim, podemos compilar os dois novamente e tudo deve funcionar. Vamos executar?

Basta usar a `bookstore` normalmente, escolhendo um livro e informando o nome para emissão da nota fiscal. Um exemplo simplificado de iteração com a `bookstore` seria:

```
CustomLogger [TRACE]:
Iniciando a execução da bookstore

Lista de livros disponíveis
...

Digite o número do livro que quer comprar: 2

CustomLogger [INFO]:
O livro escolhido foi: Java 8 Prático

Informe seu nome, para que possamos emitir a nota fiscal:
Turini

Obrigado!

<<< depois de alguns segundos >>>

falha ao emitir a nf

[CustomLoggerFinder]
module: br.com.casadocodigo.tracking
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
CustomLogger [TRACE]:
linha 14 do método logDebugTrace na classe
br.com.casadocodigo.tracking.Stack

CustomLogger [TRACE]:
linha 16 do método emit na classe
br.com.casadocodigo.nf.service.WSPrefeitura

CustomLogger [TRACE]:
linha 19 do método onNext na classe
br.com.casadocodigo.nf.service.NFSubscriber

CustomLogger [TRACE]:
linha 7 do método onNext na classe
br.com.casadocodigo.nf.service.NFSubscriber

CustomLogger [TRACE]:
linha 1447 do método consume na classe
java.util.concurrent
  .SubmissionPublisher$BufferedSubscription

CustomLogger [TRACE]:
linha 923 do método exec na classe
java.util.concurrent
  .SubmissionPublisher$ConsumerTask

CustomLogger [TRACE]:
linha 283 do método doExec na classe
java.util.concurrent.ForkJoinTask

CustomLogger [TRACE]:
linha 1603 do método runWorker na classe
java.util.concurrent.ForkJoinPool

CustomLogger [TRACE]:
linha 175 do método run na classe
java.util.concurrent.ForkJoinWorkerThread
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Os logs não apareceram!?

Se você não fez a última modificação sugerida no capítulo da API de *logging*, para que a classe `CustomLoggerFinder` retornasse uma instância de *logger* com `Level.ALL`, nenhum desses logs do tipo `TRACE` devem ter aparecido.

Como corrigir? Basta modificar a linha do retorno que tudo deverá funcionar:

```
return new CustomLogger(Level.ALL);
```

Para evitar problemas como esse em um código de produção, muitas vezes seria interessante adicionar uma mensagem de `WARNING`, alertando que nenhuma mensagem de *trace* seria exibida com o `Level` atual, caso ele fosse menor que o esperado.

Funciona! Todos os detalhes da `Stack` foram exibidos.

Legal, não é? A API antiga nos atende bem mesmo para um exemplo simples. Mas os problemas aparecem quando queremos evoluir esse código. Um exemplo seria filtrando a primeira linha, que mostra as informações de `TRACE` do método `logDebugTrace`, da própria classe `Stack`.

Esse detalhe é irrelevante, e não deveria ficar repetindo o tempo todo.

```
CustomLogger [TRACE]:
linha 14 do método logDebugTrace na classe
br.com.casadocodigo.tracking.Stack
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Como resolver, adicionando um `if` ? E se forem 10, faremos 10 `if` s?

Ok, poderíamos sim transformar o *array* em um *Stream* do JDK 8 para usar seus métodos `filter` , `skip` etc.:

```
StackTraceElement[] stacks =
    new Throwable().getStackTrace();

Arrays.stream(stacks)
    .skip(1)
    .forEach(...);
```

Isso ajuda muito no problema de design do código, trocando o formato imperativo da API pelo design declarativo dos *Streams*, mas nem um pouco com a parte de gargalo de performance. O *array* de `StackTraceElement` s já foi criado com toda stack, independente de precisarmos usar só uma ou duas das potenciais centenas de informações — além disso, adicionamos uma etapa a mais, movendo todos os dados para uma outra estrutura.

Precisa do `Class` que está sendo executado? Boa sorte implementando um monte de código para conseguir isso. Isso fora os *hidden frames* ou implementações de *reflection*, como `Method#invoke` e `Constructor#newInstance` , que não são mantidos pela máquina virtual.

11.2 STACK-WALKING API

Com a nova API, conseguimos recriar o mesmo código, mas de uma forma muito mais declarativa e eficiente. O *factory method* `getInstance` retorna um objeto do tipo `StackWalker` , que é sua principal classe:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
StackWalker walker = StackWalker.getInstance();
```

Essa instância tem acesso a um *Stream* de *frames* da stack. Esses frames são representados pela `StackWalker.StackFrame`, que tem uma interface pública muito parecida com o antigo `StackTraceElement`. A grande diferença está no acesso ao `Class` dos elementos, além de outras informações que veremos adiante.

```
interface StackWalker.StackFrame {

  int getByteCodeIndex();
  String getClassName();
  Class<?> getDeclaringClass();
  String getFileName();
  int getLineNumber();
  String getMethodName();
  boolean isNativeMethod();
  StackTraceElement toStackTraceElement();
}
```

Podemos usar o método `forEach`, passeando por esses *frames*, para chegar no mesmo resultado de antes.

```
walker.forEach(element ->
  logger.log(Level.TRACE,
      " linha "
    + element.getLineNumber()
    + " do método "
    +  element.getMethodName()
    + " na classe "
    + element.getClassName()
));
```

Experimente mudar sua implementação da classe `Stack` dessa forma:

```
package br.com.casadocodigo.tracking;

import java.lang.System.Logger;
import java.lang.System.Logger.Level;
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
public class Stack {

  public static void logDebugTrace() {

    Logger logger = System.getLogger("CustomLogger");

    StackWalker.getInstance().forEach(element ->
      logger.log(Level.TRACE,
          " linha "
        + element.getLineNumber()
        + " do método "
        +  element.getMethodName()
        + " na classe "
        + element.getClassName()
    ));
  }
}
```

O código ficou mais enxuto, claro, mas está longe de ser o único ganho. Antes de entrar em mais detalhes, não deixe de compilar e executar o projeto novamente para ver que o output será exatamente o mesmo.

Andando na Stack

Você pode utilizar o método `walk` para fazer transformações no *Stream* com cada um dos *frames* de sua *stack*.

```
walker.walk(stream -> {
 //...
});
```

Um exemplo de uso seria para fazer o próprio filtro que foi mencionado antes, para que as informações da classe `Stack` não apareçam sempre no log `TRACE` dessa nossa implementação. Isso poderia ser feito com as várias possibilidades do *Stream*, como por exemplo, o `filter`:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
StackWalker.getInstance()
  .walk(stream ->
  stream.filter(frame ->
    !Stack.class.getCanonicalName()
    .equals(frame.getClassName())
  )
...
```

Ou ainda, usando um `skip(1)`, já que sabemos que a classe será sempre o primeiro frame.

```
StackWalker.getInstance()
  .walk(stream -> stream.skip(1)
...
```

Operações de `reduce` como `(drop|keep)While` também são alternativas que poderiam ser utilizadas. Em resumo, é um *Stream* normal; você pode e deve usar qualquer um dos poderosos recursos de sua API.

Feito isso, podemos coletar o resultado em uma lista de *frames* e fazer o `forEach`, ou usar um `peek` para imprimir cada um dos elementos e coletar em uma lista no final. A vantagem da segunda opção é que você pode continuar encadeando operações, caso seja necessário, ou guardar o resultado em uma variável que pode ser usada em algum outro ponto do código.

```
StackWalker.getInstance()
  .walk(stream -> stream.skip(1)
  .peek(element ->
    logger.log(Level.TRACE,
        " linha "
      + element.getLineNumber()
      + " do método "
      +  element.getMethodName()
      + " na classe "
      + element.getClassName()
    ))
  .collect(Collectors.toList()));
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Com essas mudanças, a classe completa ficará assim:

```
package br.com.casadocodigo.tracking;

import java.lang.System.Logger;
import java.lang.System.Logger.Level;

import java.util.stream.*;

public class Stack {

  public static void logDebugTrace() {

    Logger logger = System.getLogger("CustomLogger");

    StackWalker.getInstance()
    .walk(stream -> stream.skip(1)
    .peek(element ->
      logger.log(Level.TRACE,
          " linha "
        + element.getLineNumber()
        + " do método "
        +  element.getMethodName()
        + " na classe "
        + element.getClassName()
      ))
      .collect(Collectors.toList()));
  }
}
```

Experimente compilar e executar o código com essa nova versão. Você verá que, dessa vez, a primeira linha de `TRACE` do nosso componente será da classe `WSPrefeitura`, já que a `Stack` está sendo filtrada.

```
[CustomLoggerFinder]
module: br.com.casadocodigo.tracking

CustomLogger [TRACE]:
linha 16 do método emit na classe
br.com.casadocodigo.nf.service.WSPrefeitura
...
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

11.3 CRIANDO STACKS COM MAIS INFORMAÇÕES

No lugar de comparar a `String` com nome da classe, como fizemos no exemplo de filtro anterior, poderíamos ter comparado o `Class` do objeto diretamente com o `getDeclaringClass`, que vimos ser um dos diferenciais dessa nova API.

Eu evitei isso por um motivo, que pode facilmente ser percebido ao usar o método:

```
StackWalker.getInstance()
  .walk(stream ->
  stream.filter(frame ->
    !Stack.class.equals(frame.getDeclaringClass()))
  .peek(...)
```

Ao compilar, o resultado será uma `UnsupportedOperationException`:

```
java.lang.UnsupportedOperationException:
No access to RETAIN_CLASS_REFERENCE: []
  StackWalker.ensureAccessEnabled(StackWalker.java:581)
  StackFrameInfo.getDeclaringClass(StackFrameInfo.java:72)
  at br.com.casadocodigo.tracking/
  br.com.casadocodigo.tracking.Stack.lambda
  $logDebugTrace$0(Stack.java:15)
```

Isso acontece pois, para evitar carregar mais informações do que necessário, por default o `StackWalker` não retém essa informação. Como resolver?

Você pode indicar isso ao método `getInstance`, que possui uma sobrecarga recebendo um `StackWalker.Option` como parâmetro:

```
enum StackWalker.Option {
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  RETAIN_CLASS_REFERENCE,
  SHOW_HIDDEN_FRAMES,
  SHOW_REFLECT_FRAMES
}
```

Podemos usar seu valor `RETAIN_CLASS_REFERENCE` para que ele passe a reter o objeto do tipo `Class` de cada um dos `StackFrame` s. Nosso código ficaria assim:

```
StackWalker.getInstance(
  Option.RETAIN_CLASS_REFERENCE);
```

A máquina virtual também não exibe alguns frames da *stack*, incluindo os de implementações de *reflection* — como o `Method#invoke` e o `Constructor#newInstance` . Diferente da antiga versão, do `Throwable#getStackTrace` , com a nova API você também pode opcionalmente exibir essas informações, utilizando as opções `SHOW_HIDDEN_FRAMES` e `SHOW_REFLECT_FRAMES` , respectivamente.

Em nosso caso, o resultado seria o mesmo, mas APIs que usam *reflection* diretamente ou invocações dinâmicas, com em expressões *lambda* e *method references*, teriam esses detalhes exibidos.

11.4 DOWNLOAD DO PROJETO COMPLETO

Se quiser, você pode baixar ou passear pelo código completo do módulo de `tracking` , que está disponível na pasta deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Capítulo 12

MAIS JAVA 9, APIS E OUTRAS MUDANÇAS

Se você acompanhou as novidades das versões anteriores da linguagem, talvez tenha ouvido falar do *Project Coin*. Essa foi uma das propostas mais populares do JDK 7, trazendo uma série de melhorias para a linguagem Java. Uma de suas mudanças mais conhecidas aconteceu na inferência de tipos, permitindo que declarações genéricas tivessem o tipo definido apenas do lado esquerdo:

```
// pre-JDK7
Map<Integer, String> map = new HashMap<Integer, String>();

// pos-JDK7
Map<Integer, String> map = new HashMap<>();
```

Além do famoso *operador diamante*, como essa possibilidade foi conhecida, também surgiram os *multi-catch blocks*, transformando o código que antigamente era assim:

```
catch (IOException ex) {
  //...
}
catch (SQLException ex) {
  //...
}
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Em uma única declaração que separa as diferentes exceptions com um pipe:

```
catch (IOException|SQLException ex) {
  //...
}
```

E também o `Try-With-Resources`, que substitui todo *boilerplate* de chamar o método `close` em implementações do tipo `AutoCloseable`, sem o risco das evidentes *I/O exceptions*.

O código que era assim:

```
BufferedReader reader = null;

try {
  //...
}
finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

A partir de então, pode ser escrito como a seguir:

```
try (BufferedReader reader = new BufferedReader(...)) {
  //...
}
```

O `reader.close()` é chamado implicitamente, sem você precisar se preocupar com nada.

Foram muitos os ganhos, especialmente na sintaxe simplificada que ajuda muito na legibilidade desse tipo de código, mas ainda assim existiam algumas pontas soltas ou limitações nesses recursos.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

No JDK 9, uma nova proposta chamada *Milling Project Coin* evolui ainda mais esses recursos e outros que vão além da proposta original.

Try-With-Resources

Apesar do grande avanço com a introdução do `try-with-resources` e a interface `AutoCloseable` em si, com a versão anterior da linguagem você sempre precisava criar uma nova variável dentro do `try`, como no exemplo que vimos:

```
try (BufferedReader reader = new BufferedReader(...)) {
  //...
}
```

Em outras palavras, se você já tivesse esse `BufferedReader` construído — recebendo como parâmetro de um método, por exemplo —, ele não poderia ser usado. Era sempre necessária a criação de uma nova instância, reatribuindo seu valor.

```
public void read(BufferedReader reader) {

  try (BufferedReader reader = reader) {
    //...
  }
}
```

No JDK 9, toda variável **efetivamente final** pode ser utilizada diretamente dentro do `try-with-resources`, que ela será finalizada exatamente como quando criamos um novo tipo.

O código anterior finalmente pode ser escrito assim:

```
public void read(BufferedReader reader) {

  try (reader) {
    //...
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
}
```

Além de o código ficar mais enxuto, você pode utilizar o mesmo objeto que já foi construído anteriormente, evitando a reatribuição.

> **Variáveis efetivamente finais**
>
> Talvez o termo soe estranho para você, mas o JDK 8 introduziu o conceito de variáveis efetivamente finais. Toda variável, mesmo sem o modificador `final` explicitamente declarado, é considerada efetivamente final se você não a modifica em nenhum ponto do código depois de instanciá-la.
>
> Para mais, acesse: http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

Operador diamante em classes anônimas

A inferência de tipos do operador diamante também recebeu uma melhoria. Até então, o recurso poderia ser usado em tipos genéricos, como ao instanciar mapas e listas, mas não funcionava em classes anônimas, como no exemplo:

```
Callable<String> callable = new Callable<>() {
  @Override
  public String call() {
    return "finalmente podemos fazer isso";
  }
};
```

Esse mesmo código, compilado em Java 8, resultaria em erro

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

de *identifier expected*.

```
cannot infer type arguments for
java.util.concurrent.Callable<V>
reason:
cannot use '<>' with anonymous inner classes
```

Ele não só funciona com a nova versão da linguagem, como também pode ser usado diretamente no retorno de um método:

```
public Callable<String> ultraImportantMethod() {
  return new Callable<>() {
    @Override
    public String call() {
      return "finalmente podemos fazer isso";
    }
};
```

Afinal, o tipo do retorno do método já está explícito. A inferência pode ser feita e seu código ficará mais enxuto e legível.

Métodos privados em interface

No JDK 8, interfaces podiam ter métodos `static` e também com implementações, chamados `default methods`. Essa possibilidade, junto com recursos como as expressões *lambda*, abriu caminho para a evolução estratégica da linguagem que finalmente pode aderir um estilo mais próximo à programação funcional.

A possibilidade foi bastante polêmica na época, sugerindo uma possível variação de herança múltipla ou mixins no Java. Vale lembrar de 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.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Uma nova possibilidade é que agora **interfaces também podem ter métodos privados!** Sim, você leu certo. Mas não se preocupe, nada mudou.

A grande motivação por trás deles é te permitir encapsular código dos métodos *default* ou estáticos, evitando duplicações. Com o passar do tempo e dessas novas possibilidades, virou uma realidade comum termos sobrecargas de métodos que possuem códigos muito parecidos, ou mesmo repetidos. Quer um exemplo?

O código a seguir é de uma das interfaces do sistema da Alura, que é a plataforma de cursos online do grupo Caelum.

```
public interface Choiceable {

  List<Alternative> getAlternatives();

  default boolean hasOpinion() {
    return getAlternatives()
      .stream().anyMatch(Alternative::hasOpinion);
  }

  default boolean hasHint() {
    return getAlternatives()
      .stream().anyMatch(Alternative::hasHint);
  }
}
```

Um `Choiceable` representa um tipo de atividade que tem uma ou mais escolhas — sim, uma escolha de nome questionável, mas esse assunto fica para um outro livro. Perceba que os métodos `hasOpinion` e `hasHint` fazem exatamente a mesma coisa, mudando apenas a condição que é passada para o `anyMatch` da API de `Stream`.

Em um caso tão simples e pequeno, estaria tudo bem, mas o ideal é que não exista repetições em nossos códigos. Para evitar

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

isso, poderíamos extrair um método que recebe uma condição e nos diz se ela é ou não atendida.

```
public interface Choiceable {

  List<Alternative> getAlternatives();

  default boolean hasOpinion() {
    return matches(Alternative::hasOpinion);
  }

  default boolean hasHint() {
    return matches(Alternative::hasHint);
  }

  default boolean matches(Predicate<Alternative> condition) {
    return getAlternatives()
      .stream().anyMatch(condition);
  }
}
```

Perceba que, por limitação da linguagem, o método extraído precisou ser `default` e, como todo método de interface até hoje, **público**! Isso significa que essa implementação interna fica exposta e poderia ser usada por qualquer ponto do sistema (ou outros sistemas, caso faça parte de uma API pública).

Para evitar isso, com o JDK 9, podemos deixá-lo privado.

```
public interface Choiceable {

  List<Alternative> getAlternatives();

  default boolean hasOpinion() {
    return matches(Alternative::hasOpinion);
  }

  default boolean hasHint() {
    return matches(Alternative::hasHint);
  }
```


E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

```
  private boolean matches(Predicate<Alternative> condition) {
    return getAlternatives()
      .stream().anyMatch(condition);
  }
}
```

Apesar de soar estranho no primeiro momento, com essa nova possibilidade podemos manter o encapsulamento das implementações default da interface. Todas essas melhorias estão definidas na proposta de *Milling Project Coin*, que refina muitas das possibilidades anteriormente introduzidas na linguagem.

Se quiser, você pode ver mais detalhes em:

http://openjdk.java.net/jeps/213

Outra melhoria extremamente significante que aconteceu nesse novo release foi relacionada à anotação de *deprecated* e implementações internas que foram melhor encapsuladas. Por muito se discutiu se **isso significaria a quebra de compatibilidade da linguagem**, impossibilitando a migração de algumas bibliotecas e projetos que há muito usam e dependem desses comportamentos legados.

12.1 QUEBRA DE COMPATIBILIDADE?

O Java 9 não quebrou o ciclo de retrocompatibilidade dos mais de 20 anos de existência da linguagem. Apesar disso, muitas das APIs internas foram fortemente encapsuladas, como já foi comentado aqui no livro sobre o exemplo do `sun.misc.Unsafe` e o módulo `jdk.unsuported`.

O objetivo é que, no futuro, as bibliotecas não usem mais essas implementações internas, para que aí sim possam ser removidas.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Outro grande passo para esse lado foi a evolução da anotação `@Deprecated`, que ganhou duas novas opções:

- `forRemoval`, que pode ser `true` ou `false`, indicando se o elemento anotado será removido em um próximo release da API, ou se está descontinuado, mas não existe a intenção direta de remoção em um futuro próximo.
- `since`, que indica desde quando o elemento passou a ser *deprecated*. Seu valor é uma `String`, podendo indicar uma versão específica de release, como *beta*, *alpha*, *release candidate* etc.

Com isso, podemos passar mais informações sobre o status da API descontinuada, e indicar para quem está usando a implementação específica que ele precisará se preocupar em substituí-la em uma futura atualização. A comunicação fica mais clara entre o lado dos usuários e dos *mantainers* das APIs!

Por falar nisso, algumas das APIs internas da linguagem entraram para essa lista! Um exemplo são os construtores explícitos dos *wrappers* primitivos (como `Integer`, `Long`, `Boolean` etc.).

No lugar de `new Integer(3)`, por exemplo, é recomendado que você use a versão estática do `Integer.valueOf(3)`, que lida melhor com espaço e tem uma performance melhor. O mesmo para a versão que recebe uma `String`, `new Integer("5")`, onde você poderia usar diretamente o `Integer.parseInt("5")`. Ambos construtores receberam `@Deprecated(since="9")`, mas nenhum deles foi marcado com `forRemoval`.

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

O mesmo vale para as interfaces `Observer` e `Observable`, do JDK 1.0, o *CORBA*, certificados *SHA-1* e a legada *Applet API*.

Underscore não é mais um nome válido

Se você nunca viu esse código e se assustou por compilar, tem a oportunidade de fazer isso agora:

```
String _ = "sim, isso funciona";
```

É muito raro ver um desses por aí, claro. Mas um *underscore* sozinho foi aceito até agora como um nome válido de variável, método ou — pasmem — mesmo de classes!

Desde o JDK 8, um *warning* começou a ser exibido, em tempo de compilação, desencorajando o uso do underscore como nome e também avisando que ele **poderia ser removido em uma versão futura da linguagem**. Pois bem, ele foi. A partir do Java 9, esse código não compila mais.

Você ainda pode usar em nomes com mais de um caractere, claro. Um uso bastante comum, e que continuará compilando, é como divisor de constantes que naturalmente são escritas em *uppercase*.

```
final int NUMERO_MAGICO = 23;
```

Além desses, existem comportamentos que estão diferentes nessa nova versão, mas que não vão necessariamente gerar erros de compilação em seu código — ou mesmo problemas de quebra de compatibilidade. As mudanças do JDK vão muito além de código e melhorias nas APIs, muitos detalhes de performance, algoritmos da máquina virtual e detalhes internos mudaram. Um detalhe bastante interessante foi na adoção de uma forma muito mais

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

eficiente de representar Strings em memória.

12.2 STRINGS MAIS LEVES E EFICIENTES

Se você conferir na implementação das versões anteriores da linguagem, perceberá que uma String guarda seu valor interno em um *array* de caracteres, que mantém dois bytes para cada um deles.

```
private final char value[];
```

O problema da abordagem atual é que, na maior parte dos casos, as Strings usam valores em formato *ISO-8859-1/Latin-1* que precisam de apenas um byte por caractere. Em outras palavras, poderíamos usar metade do espaço!

Bem, é o que estamos fazendo agora. Experimente abrir o JShell e ver como esse campo de valor é declarado hoje:

```
jshell> String.class.getDeclaredField("value")
```

```
private final byte[] value;
```

Um array de bytes!

Com isso, usamos apenas um byte, que **será o suficiente na esmagadora maioria dos casos**. Para os demais, foi adicionado um campo adicional de um byte para indicar qual o *encoding* que está sendo usado. Isso será feito automaticamente, baseado no conteúdo da String.

Essa proposta ficou conhecida como *Compact Strings*!

http://openjdk.java.net/jeps/254

O próprio Java 8 já tinha introduzido um recurso bem

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

interessante de reaproveitamento, ou *deduplicação*, de Strings, criado com objetivo de reduzir espaço ocupado em memória. As pesquisas da proposta mostraram que, dos quase 25% de objetos do tipo `String` , uma média de 13.5% era duplicação.

A ideia é que, durante o processo de compactação dos dados no *heap* de memória, trechos repetidos de Strings fossem reaproveitados. O problema desse recurso é que existia apenas em uma implementação opcional de *Garbage Collector*, conhecida como *Garbage-First, G1*. Para utilizá-lo em tempo de execução, você precisa especificar explicitamente com a opção:

```
-XX:+UseG1GC flag.
```

No JDK 9, como já era esperado por muitos, essa implementação alternativa virou padrão.

12.3 GARBAGE COLLECTORS E O G1

Todas as plataformas modernas possuem gerenciamento de memória automático através de algoritmos de coleta de lixo, que são os famosos *Garbage Collectors (GCs)*. Você pode pensar nele de forma simplificada como threads que são executadas de tempos em tempos, limpando objetos que já não estão sendo referenciados em nenhuma parte do código. Você não sabe ou tem controle algum de quando exatamente essa coleta vai acontecer.

Além do reaproveitamento de Strings, o G1 traz diversos outros benefícios interessantes. Entre os principais deles, um destaque é o fato de ele ser muito mais previsível que a atual implementação, chamada *Concurrent Mark-Sweep Collector* (CMS).

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

Sua execução acontece com diferentes threads e em uma frequência muito maior, compactando o heap de memória durante a execução, no lugar de fazer *full GCs* como os tradicionais algoritmos que varrem toda memória de uma vez — muitas vezes travando, ou deixando as aplicações lentas. Com isso, conseguimos ter um algoritmo de GC mais eficiente e mais previsível, com paradas mais curtas e constantes.

Para conhecer mais detalhes avançados sobre performance e particularidades do algoritmo, você pode se interessar pelo tutorial oficial da Oracle:

http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

12.4 DOWNLOAD DOS EXEMPLOS DESTE CAPÍTULO

Todos os exemplos de código vistos aqui estão disponíveis no diretório deste capítulo no repositório:

https://github.com/Turini/livro-java-9

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

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/Turini/livro-java-9

Para criar uma intimidade maior com as novidades em APIs e na própria estrutura modular da linguagem, recomendo que você exercite bastante os exemplos daqui e vá além, fazendo novos testes e códigos além dos sugeridos.

13.1 COMO TIRAR SUAS DÚVIDAS

Encaminhe suas dúvidas ou crie tópicos para discussão no fórum da Casa do Código, criado especialmente para isso:

http://forum.casadocodigo.com.br

Além de perguntar, você também pode contribuir com sugestões, críticas e melhorias para nosso conteúdo. Sentiu falta de algum assunto? Não deixe de me enviar para que ele seja adicionado em futuras atualizações do livro. É só acessar:

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com

http://erratas.casadocodigo.com.br

13.2 E AGORA?

Esse é o momento em que você coloca tudo que viu aqui no livro em prática, seja criando novos ou migrando seus projetos existentes.

Como vimos, apesar de ser um recurso extremamente interessante e poderoso, você não precisa migrar de imediato todos os seus projetos para o modular. A compatibilidade é mantida para que você possa atualizar a versão do Java, tirando proveito de todos os recursos e APIs, sem precisar migrar.

Espero que o livro tenha ajudado-o nessa nova jornada, e que você se sinta seguro e preparado para usar o novo Java — interativo, reativo e modular.

É hora de programar!

E-book gerado especialmente para Luan Sousa Santana - luanss4@gmail.com