

Scala

Como escalar sua produtividade



Casa do
Código

PAULO SIQUEIRA

ISBN

Impresso e PDF: 978-85-5519-234-0

EPUB: 978-85-5519-235-7

MOBI: 978-85-5519-236-4

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>.

AGRADECIMENTOS

Escrever um livro não é uma tarefa simples, mesmo um livro relativamente curto como este. Leva tempo e é fácil perder o foco. Nesse sentido, agradeço a grande paciência dos editores, que me incentivaram e apoiaram, muitas vezes simplesmente me lembrando que o livro estava "quase" pronto. Se não fosse isso, talvez o livro nunca fosse finalizado.

Também foi indispensável o apoio da minha esposa, Keli, sem o qual não seria possível *trabalhar fora do trabalho*. E o maior apoio de todos que, sem dúvida, veio de minha filha. Mesmo sem saber, ela me manteve firme e de bom humor, como apenas o sorriso de uma criança pode fazer. Obrigado, Victoria.

Por fim, um muito obrigado a todos os colegas de trabalho, antigos e atuais, professores e alunos, e colegas de eventos. Foi o contato com grandes profissionais ao longo do tempo que deram forma ao conhecimento passado neste livro.

SOBRE O AUTOR

Paulo "JCranky" Siqueira é desenvolvedor de software especializado na máquina virtual Java (JVM). Trabalhou com a linguagem *Java* desde 2001, já tendo atuado com diversos tipos de frameworks e tecnologias relacionadas, desde *applets*, passando por aplicações desktop com *AWT* e *Swing*, até aplicações web. Isso inclui algumas ferramentas amadas e outras odiadas hoje em dia, como *Servlets* e *JSPs*, *Struts*, *Spring*, *Hibernate*, *JPA*, *Java EE*, entre muitas outras.

Depois de anos trabalhando com a linguagem *Java*, uma sede por melhores opções tomou conta do autor, que encontrou na linguagem *Scala* e na programação funcional uma alternativa com grande potencial. Algum tempo depois, trabalhar com essa linguagem se tornou a única alternativa aceitável — foi um caminho sem volta.

Depois de mais alguns anos trabalhando e ministrando treinamentos com *Scala* no Brasil, e depois de alguns anos à frente dos *Scaladores* — grupo de usuários da linguagem *Scala* de SP (<http://scaladores.com.br> —, Paulo se mudou para Berlin, onde hoje é engenheiro de Software na *Zalando SE* (<http://zalando.de>, o maior site de comércio de moda da Europa, atuando 100% com *Scala*.

Paulo também é o criador e responsável por dois projetos de código aberto em *Scala* que podem ser interessantes para o leitor: a *Lojinha* (<https://github.com/jcranky/lojinha>, uma aplicação web simples desenvolvida com o *Play Framework* para leilão de usados; e o *EasyForger* (<http://easyforger.com/>, uma *API* que oferece uma *DSL* para desenvolvimento de *mods para Minecraft* em *Scala*.

PREFÁCIO

Uma leitura focada em iniciantes, este livro é um bom ponto de partida para o leitor que sabe pouco ou nada sobre a linguagem *Scala* e está curioso sobre como ela funciona. Um pouco de experiência com programação ajudará o leitor a seguir o conteúdo do livro, mas não é necessário nenhum conhecimento mais avançado.

Serão apresentadas as características e recursos principais da linguagem, bem como conceitos indispensáveis para seu uso efetivo, em especial conceitos sobre Programação Funcional. Os recursos apresentados são o ponto de partida para conceitos mais avançados da linguagem, portanto, neste livro focamos em construir o conhecimento mínimo necessário para se trabalhar com *Scala*.

Além de conceitos, o livro também aborda elementos da *API* que todo programador *Scala* precisa conhecer, incluindo (mas não se limitando a) classes e hierarquia das classes fundamentais da linguagem, *API* de coleções e parseamento de arquivos *XML*.

Para quem já tem algum conhecimento em *Scala*, o livro também pode ser útil. O leitor talvez queira apenas pular até o capítulo 6 ou 7, onde começamos a abordar a *API* de coleções e elementos de Programação Funcional, seguido de recursos mais avançados da linguagem.

Boa leitura!

SOBRE O LIVRO

Se você já sabe programar e agora quer aprender um pouco sobre a linguagem *Scala*, este livro é para você. O objetivo do livro não é tornar o leitor em um especialista na linguagem, mas sim mostrar o caminho das pedras para quem está começando a estudá-la. Mostrar onde estão os elementos fundamentais para que o leitor possa continuar sua jornada no futuro, sabendo o que investigar quando tiver dificuldades.

Ao final do livro, o leitor conseguirá entender e participar de projetos *Scala*, bem como terá adquirido conhecimentos em *Programação Funcional*, um paradigma indispensável para quem realmente quer aprender a linguagem, e também cada vez mais importante em aplicações que precisam ser altamente escaláveis.

Este livro também vai ajudar o leitor que está considerando adicionar *Scala* em um projeto *Java* e não sabe por onde começar. Como mencionaremos no decorrer do livro, *Java* e *Scala* se integram muito bem, portanto esse também é um ponto de partida recomendado.

Por fim, nem todo mundo gosta de ler código-fonte em um livro. Logo, para facilitar a vida de quem quiser acessar todos os exemplos do livro de uma forma mais simples, criamos um repositório no GitHub: <https://github.com/jcranky/scalando>. Eventuais correções necessárias nos códigos serão aplicadas lá também.

Sumário

1 Introdução a Scala	1
1.1 O mínimo que você precisa saber sobre Scala	1
1.2 Instalação	5
1.3 Nosso primeiro programa	9
1.4 Scaladoc	11
1.5 REPL	13
1.6 Inferência de tipos	17
1.7 Um pouco de história	19
2 Acessando fotos do Flickr	21
2.1 Conhecendo nosso problema	21
2.2 Modelos e funções	25
2.3 Como seria em Java?	27
2.4 Como seria no Java 8?	29
3 Classes e objetos	31
3.1 Nossa primeira classe	31
3.2 val vs. var	32
3.3 Métodos e funções	34
3.4 Construtores	39
3.5 Parâmetros default e nomeados	41

3.6 objects	43
3.7 Classes abstratas	45
4 Case classes e pattern matching	46
4.1 Case classes	46
4.2 Case objects	50
4.3 Pattern matching com case classes e object	51
4.4 Método unapply e pattern matching com qualquer classe	56
4.5 Método de fábrica apply	59
5 Hierarquia das classes básicas da linguagem	62
5.1 Option, Some e None	62
5.2 Any-o que?	65
5.3 Null, Nothing, Unit e ???	67
5.4 Exceptions	70
5.5 Value Classes	73
6 Coleções	77
6.1 Elemento básico: TraversableLike	77
6.2 Sets	79
6.3 Lists	81
6.4 Tuplas	85
6.5 Maps	88
6.6 Arrays	90
6.7 Coleções imutáveis versus coleções mutáveis	92
7 Programação funcional	95
7.1 O que é Programação Funcional?	95
7.2 Recebendo funções com dois ou mais parâmetros	99
7.3 Encontrando elementos: filter e find	100
7.4 Transformando elementos: map	102

7.5 Mapeando resultados com coleções aninhadas	106
7.6 Agregando resultados: fold e reduce	107
8 Tipagem avançada	113
8.1 Tipos parametrizados	114
8.2 Limites de tipos: Type Bounds	116
8.3 Tipos invariantes, covariantes e contravariantes	120
9 Um pouco de açúcar: for comprehensions	128
9.1 Percorrendo múltiplas coleções de forma legível	128
9.2 Mantendo a imutabilidade	130
9.3 O segredo do for: Monads	132
10 Classes abstratas e traits	135
10.1 Classes abstratas	135
10.2 Traits	137
10.3 Classes seladas	141
10.4 Herança múltipla e o problema do diamante	144
11 Parseando XML	152
11.1 O básico de XML em Scala	152
11.2 Parseando a resposta XML do Flickr	154
12 Implicits	159
12.1 Adicionando funcionalidade a tipos existentes: conversões implícitas	159
12.2 Conversões implícitas ambíguas	164
12.3 Passando parâmetros sem passar nada: parâmetros implícitos	166
12.4 Como o sum soma valores "somáveis"? ¹⁶⁹	166
13 Colocando tudo junto	173
13.1 Bibliotecas e ferramentas	173
13.2 Componentes da API	175

13.3 Considerações finais	178
14 O fim, e o começo	182

INTRODUÇÃO A SCALA

Antes de começarmos, um lembrete: os exemplos de código do livro estão também disponíveis no GitHub (<http://github.com/jcranky/scalando>), para facilitar a consulta do leitor. Dito isso, vamos começar a *Scalar*!

1.1 O MÍNIMO QUE VOCÊ PRECISA SABER SOBRE SCALA

Scala é uma linguagem que une *Orientação a Objetos* e *Programação Funcional*. Para desenvolvedores com conhecimento em linguagens orientadas a objetos, como Java ou C#, Scala é um passo natural, oferecendo o lado funcional como um desafio em termos de aprendizagem, mas com muitos recursos já familiares. Um desafio com muitos benefícios, como veremos no decorrer do livro.

Além disso, a sintaxe dessa linguagem é muito enxuta — lembrando linguagens dinâmicas como Python ou Ruby. Porém, diferente dessas linguagens, Scala é estaticamente tipada. Ou seja, os tipos de variáveis, retornos de métodos etc. são verificados em tempo de compilação, evitando assim que erros grosseiros passem despercebidos.

As características mencionadas são muito interessantes, mas podemos encontrar recursos parecidos em outras linguagens. Um

grande trunfo da linguagem Scala é trazer tudo isso para a JVM (*Java Virtual Machine*). Sim, aquele ambiente de execução utilizado pela linguagem Java — e hoje em dia por várias outras também. E a integração entre Java e Scala é quase transparente na grande maioria dos casos.

LINGUAGENS DA JVM

Hoje em dia, existem muitas linguagens desenvolvidas para serem executadas na *Máquina Virtual Java* — ou adaptadas para tal. Há muitos anos a *linguagem Java* deixou de ser o único foco da JVM. Entre as mais conhecidas estão, além das próprias Java e Scala, linguagens como *JRuby*, *Jython*, *Closure*, *Groovy*, entre várias outras.

Quando compilamos um arquivo `.scala`, este é transformado em *bytecode* — o mesmo conjunto de *bytecodes* utilizado quando compilamos um arquivo escrito na linguagem Java. Isso torna a integração entre as linguagens muito simples, como mencionado anteriormente. E isso também faz com que possamos tirar proveito dos vários anos de desenvolvimento já investidos na JVM, especialmente em melhorias de performance, segurança e correções de bugs, "de graça".

A única diferença significativa aqui é que, em alguns casos, o compilador Scala precisa gerar mais classes, interfaces etc. do que nós escrevemos de fato no nosso código. Já um código escrito na linguagem Java pode muitas vezes ser traduzido quase diretamente para *bytecode*. Isso acontece porque alguns recursos, como *traits* e funções anônimas, simplesmente não existem na JVM, e o compilador Scala precisa lidar com isso de alguma forma.

LAMBDA NO JAVA 8

Alguns leitores já familiarizados com Java 8 podem ficar surpresos com a afirmação de que *Java não suporta funções anônimas*, já que na versão 8 da JVM foi introduzido suporte a *lambdas*. O fato é que, até a versão 2.11 da linguagem Scala, o compilador ainda não utiliza esse recurso — cujo suporte está planejado para a versão 2.12 da linguagem.

Vejamos isso na prática. Vamos criar uma classe Java bem simples, com apenas alguns atributos e métodos, compilá-la e analisar a estrutura do arquivo `.class` gerado:

```
public class Foto {  
    private final String id;  
    private final String owner;  
  
    public Foto(String id, String owner) {  
        this.id = id;  
        this.owner = owner;  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public String getOwner() {  
        return owner;  
    }  
}
```

Para compilar o código em Java, usaremos o comando `javac` no terminal e, em seguida, o comando `javap` para exibir o *bytecode* da classe:

```
javac Foto.java  
javap -p Foto
```

A estrutura gerada para a classe anterior será a seguinte:

```
Compiled from "Foto.java"
public class Foto {
    private final java.lang.String id;
    private final java.lang.String owner;
    public Foto(java.lang.String, java.lang.String);
    public java.lang.String getId();
    public java.lang.String getOwner();
}
```

Agora vamos fazer o equivalente para uma classe Scala. Primeiro, a criação da classe em si:

```
class Foto(val id: String, val owner: String)
```

Em seguida, a compilação e visualização dos elementos resultantes. Compilar e visualizar o *bytecode* desse código será muito parecido com o que fizemos em Java:

```
scalac Foto.scala
javap -p Foto
```

O *bytecode* resultante é muito similar ao que obtemos na versão Java, mesmo sendo necessário escrever apenas uma linha de código em Scala:

```
Compiled from "Foto.scala"
public class Foto {
    private final java.lang.String id;
    private final java.lang.String owner;
    public java.lang.String id();
    public java.lang.String owner();
    public Foto(java.lang.String, java.lang.String);
}
```

A principal diferença é que os métodos *getters* não possuem a palavra *get* como parte de seus nomes — discutiremos esse assunto com detalhes em um outro momento.

Antes de instalarmos o Scala, vamos abordar rapidamente um último ponto: a *CLR* (*Common Language Runtime*) do .Net. Scala foi planejado, em um primeiro momento, para ser compatível com a

CLR do mundo .Net também, além da plataforma Java. A ideia era podermos executar uma aplicação Scala tanto na JVM quanto na CLR, sem muita dificuldade.

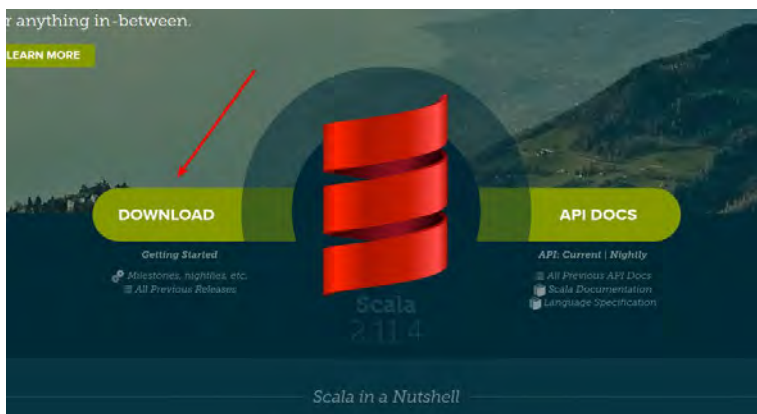
Na prática, esse ideal nunca foi realmente alcançado. Existe um compilador Scala para .Net, porém ele está muito atrasado e praticamente abandonado. Além disso, por causa do uso de bibliotecas específicas de uma determinada plataforma, migrar para a outra provavelmente não seria um cenário muito realista de qualquer forma.

1.2 INSTALAÇÃO

Instalar o ambiente para desenvolver aplicações Scala não é muito diferente do que o que fazemos com a maioria das linguagens modernas: basta baixar um ZIP, descompactá-lo e fazer algumas configurações simples. Vamos ao passo a passo!

Antes de mais nada, como já mencionamos antes, Scala é executado em cima da *Máquina Virtual Java*. Isso quer dizer que, antes de instalar o Scala em si, precisamos ter a JVM instalada. Não vamos detalhar como fazer isso, mas as instruções relevantes podem ser encontradas em <http://java.oracle.com>.

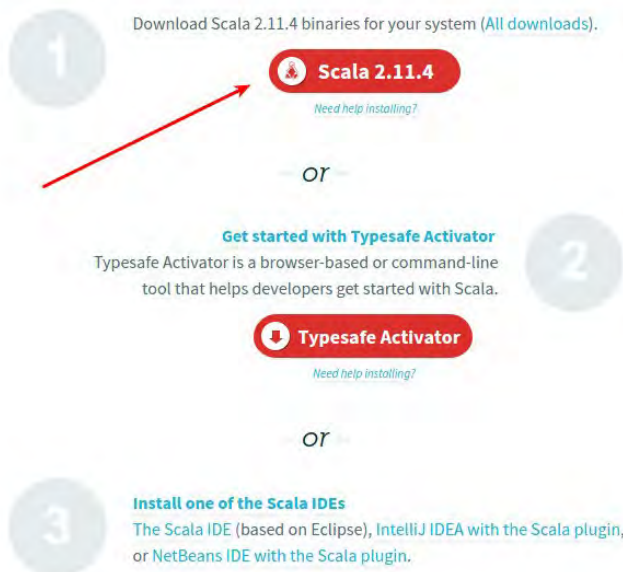
Agora, vamos ao site oficial do Scala (<http://scala-lang.org>) para baixar o ZIP da versão mais nova, que no momento da escrita desse texto é a 2.11.4. Como podemos ver na figura a seguir, a página inicial do site nos apresenta duas opções: *Download* e *API Docs*. Vamos falar da segunda opção mais para a frente. Por enquanto, clique em *Download*.



A próxima página vai novamente nos apresentar três opções. Clique na primeira, com o nome Scala e a versão atual disponível. Isso iniciará o download do arquivo compactado que mencionamos, que será um arquivo `.tgz`. A segunda opção é a de usar o *Typesafe Activator*, e a terceira possui links para quem quiser utilizar *IDEs* — o que não é necessário para seguir o conteúdo do livro.

DOWNLOAD

Choose one of three ways to get started with Scala!



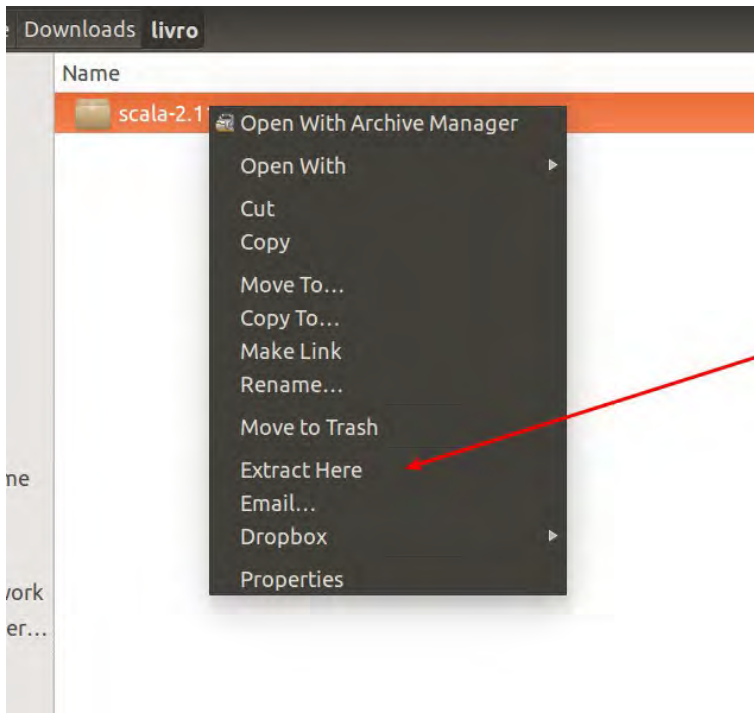
TYPESAFE ACTIVATOR

O *Typesafe Activator* é uma ferramenta que oferece tutoriais e templates de projetos em Scala e Java, utilizando a ferramenta de build *sbt* — *scala build tool* (<http://www.scala-sbt.org/>). Para o leitor curioso, vale a pena baixar e investigar a ferramenta.

Com o arquivo baixado, vamos descompactá-lo e configurar o ambiente para encontrar o Scala. Podemos descompactar o arquivo com o seguinte comando, no Linux (trocando a versão no nome do arquivo para a versão que o leitor tiver baixado):

```
tar zxvf scala-2.11.x.tgz
```

Ou podemos simplesmente clicar com o botão direito no arquivo e, em seguida, em *extrair aqui* — opção que deverá estar disponível na maioria das plataformas. Algo como o que podemos ver na figura a seguir.



Feito isso, teremos um diretório com o compilador Scala e todas as suas bibliotecas. Dependendo do ambiente que o leitor for utilizar no dia a dia, isso pode ser o suficiente. Se for usar uma *IDE*, por exemplo, a própria *IDE* poderá requisitar a localização deste diretório. Em alguns casos, nem isso será necessário, e a *IDE* ou localizará o Scala automaticamente ou fará o download ela mesma.

No nosso caso, como vamos utilizar o console (e o REPL — mais sobre ele adiante) constantemente, ainda precisamos fazer a

configuração do `PATH` do sistema. Por questão de organização, vamos também criar uma variável de ambiente chamada `SCALA_HOME`. O código a seguir, inserido no arquivo `.bashrc` do Linux faz essa configuração:

```
SCALA_HOME=/home/jcranky/java/scala-2.11.x
PATH=$PATH:$SCALA_HOME/bin

export SCALA_HOME PATH
```

Para o Windows ou Mac, os passos são parecidos, bastando o leitor procurar como configurar variáveis de ambiente para o seu caso em particular. Só é importante lembrar de colocar o caminho correto para o diretório do Scala que descompactamos anteriormente, no `SCALA_HOME`.

E pronto. Vamos agora conferir se a instalação funcionou corretamente. Para isso, digite o seguinte em um terminal de comando:

```
scalac -version
```

Se aparecer a versão do Scala que foi baixada, tudo certo. Se não, confira as configurações anteriores — qualquer letra errada fará com que o processo não funcione. Também pode ser necessário fechar e abrir o terminal de comandos após a configuração do ambiente para que o `scalac` seja encontrando corretamente.

1.3 NOSSO PRIMEIRO PROGRAMA

Já vimos um pouco de código Scala no começo do capítulo, para ilustrar o funcionamento básico da linguagem. Antes de entrar em mais detalhes de recursos da linguagem, vamos ver mais um exemplo de código Scala, agora um pouco mais prático.

Escreveremos um pequeno *script* que vai percorrer uma lista de arquivos e apagar todos os que tiverem determinadas extensões —

no caso, extensões que representam arquivos de imagens.

```
import java.io._

val arquivos = new File(".").listFiles
val extensoesImgs = List(".jpg", ".jpeg", ".gif", ".png")

def ehImagem(nomeArq: String) = extensoesImgs.exists(nomeArq.endsWith(_))

arquivos.filter(arq => ehImagem(arq.getName)).foreach(_.delete)
```

De cara, note que estamos usando um pouco da *API Java* para manipular arquivos: a classe `java.io.File`. Essa é uma das grandes vantagens da linguagem Scala em um primeiro momento. A integração com a linguagem e a plataforma Java permite que adotemos Scala facilmente em qualquer ambiente que esteja utilizando Java, e podemos usar qualquer biblioteca Java em nossos programas escritos em Scala.

Isso significa que, apesar de geralmente preferirmos usar APIs Scala para resolver nossos problemas, podemos começar com o que já sabemos do mundo Java. Assim, conseguimos alavancar o uso da linguagem Scala mesmo quando estamos apenas começando a estudá-la.

Outro ponto que deve chamar atenção é o quão concisa a linguagem é. Não precisamos ficar declarando tipos de variáveis diversas vezes, tanto na declaração quanto na criação de variáveis. Isso faz com que a linguagem seja bem enxuta, muitas vezes lembrando linguagens dinâmicas. E operações como filtragem de listas, como no exemplo anterior, ficam muito simples de executar com o suporte à Programação Funcional da linguagem.

Algo que provavelmente ainda está pouco confuso no código anterior é o *underline* (`_`). Esse símbolo é um *açúcar sintático* do Scala. Ele permite escrevermos código ainda mais enxuto, e seu funcionamento ficará mais claro no decorrer do livro. Mas tenha em

mente que seu uso é na maioria das vezes opcional e, se o código que o usa ficar mais difícil de ler em vez de mais fácil, o leitor é encorajado a evitá-lo.

Como um exercício para o leitor, tente escrever o código equivalente ao que fizemos anteriormente na sua linguagem de preferência. O que fica melhor? E o que fica pior?

1.4 SCALADOC

O *Scaladoc* é a documentação oficial das APIs do Scala. Além das APIs padrão, a grande maioria das bibliotecas de terceiros escritas em Scala também oferece uma documentação de API na forma de Scaladoc. Para quem vem do mundo Java, a situação é bem próxima ao que temos com o *Javadoc* das APIs nessa plataforma. A seguir, uma captura de tela mostra como o Scaladoc está organizado.



De novo, para quem trabalha com Java, esse tipo de documentação é algo comum de se encontrar — no caso, é chamado de Javadoc, como mencionamos anteriormente. A semelhança no

nome não é mera coincidência e a inspiração é clara. Porém, o Scaladoc resolve muitas das limitações do Javadoc, a principal delas sendo a facilidade com que se pode encontrar determinadas classes e outros elementos. Na captura de tela a seguir, veja como é fácil procurar a classe `List` :



A versão mais atual do Scaladoc pode sempre ser encontrada em <http://www.scala-lang.org/api/current/>. E versões específicas podem ser encontradas substituindo o *current* pela versão desejada. Se quisermos por exemplo o Scaladoc da versão *2.11.x*, a mais recente no momento da escrita deste livro, acessamos a URL <http://www.scala-lang.org/api/2.11.x/>, substituindo o *x* pela subversão em questão.

PREDEF

Muitas vezes alguns elementos podem parecer aparecer "do nada" no nosso código, como o método `???` que iremos explorar no *capítulo 5*. Quando isso acontecer, o primeiro lugar que o leitor deve procurar é `object Predef`. Muitos elementos padrão da linguagem são definidos lá, incluindo o método `???`.

1.5 REPL

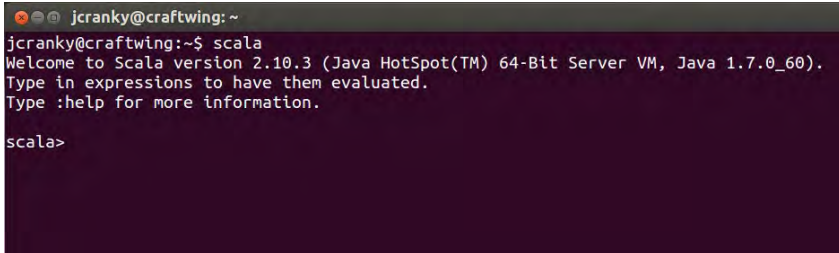
Quando começamos a trabalhar com Scala, alguns recursos e ferramentas trazem um ganho de produtividade praticamente imediato, com muito pouco investimento de tempo necessário. Um desses recursos é a *API de coleções* que, como vamos ver em um capítulo específico, simplifica muitas das tarefas de programação do dia a dia.

No mundo das ferramentas de apoio ao desenvolvimento, o REPL (*Read, Eval, Print, Loop*) também entra nessa categoria. O aprendizado necessário para utilizá-lo é bem pequeno, e o ganho é significativo, principalmente para quem está acostumado com o mundo Java, em que este tipo de ferramenta não é comum. Linguagens dinâmicas como Ruby e Python oferecem algo similar.

Mas o que é o REPL? Como diz o nome, o REPL é um interpretador que *lê* uma entrada, *avalia* e *executa* essa entrada, e *imprime* o resultado, para então começar tudo de novo — o *loop*. Explicando assim talvez pareça um pouco complicado, mas não é. Vejamos um pequeno exemplo.

Primeiro, para acessar o REPL, precisamos ter o Scala instalado.

Basta então digitar `scala` em um terminal de comandos. Isso iniciará o interpretador e seremos apresentados a um prompt de comandos similar ao que vemos na figura a seguir:

A screenshot of a terminal window with a dark background. The prompt is 'jcranky@craftwing: ~'. The user has entered 'scala', and the terminal displays the Scala version 2.10.3, the Java HotSpot(TM) 64-Bit Server VM version 1.7.0_60, and instructions to type expressions for evaluation or ':help' for more information. The prompt 'scala>' is shown at the bottom.

```
jcranky@craftwing: ~  
jcranky@craftwing:~$ scala  
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_60).  
Type in expressions to have them evaluated.  
Type :help for more information.  
scala>
```

De dentro do REPL, podemos executar qualquer instrução da linguagem Scala. A listagem a seguir ilustra uma sessão no REPL onde somamos dois números e imediatamente vemos o resultado:

```
scala> 1 + 2  
res0: Int = 3
```

Na primeira linha, digitamos `1 + 2` e pressionamos *Enter* para executar a instrução. A segunda linha é a mais importante de se entender no momento. O que está acontecendo é que, como não dissemos ao REPL onde queremos que o resultado da nossa conta seja armazenado, ele mesmo criou uma nova variável e deu um nome para ela — no caso, `res0`.

Repare que a variável recebeu o tipo `Int`, e o resultado da nossa conta. Falaremos mais sobre como esse tipo foi definido na próxima seção.

Não somos obrigados a aceitar a variável que o REPL cria. Podemos criar as nossas variáveis e atribuir o resultado da nossa instrução diretamente a ela. A seguir, executamos uma outra instrução e, dessa vez, decidimos explicitamente onde o resultado deverá ser colocado.

```
scala> val texto = "Numero: " + res0
```



```
texto: String = Numero: 3
```

Dessa vez, o resultado está na nossa variável `texto`. De quebra, estamos usando o valor gerado pela instrução anterior. Lembrando de que as instruções são sempre imutáveis, ou seja, cada vez que executamos algo no REPL, um novo resultado é gerado, e as variáveis originais não são alteradas. A não ser que façamos isso explicitamente, mas vamos discutir essa questão mais à frente.

Por enquanto, é aconselhado ao leitor se acostumar com esse tipo de "transformação", pois é o tipo de operação que fazemos o tempo todo em Programação Funcional, com muitos benefícios. Que benefícios? Chegaremos lá.

Só pelo o que apresentamos já percebemos que o REPL é extremamente útil. Ele é ainda mais útil do que parece. Ferramentas como o *sbt* ou o *maven*, por exemplo, permitem que iniciemos o REPL com nosso projeto no `classpath`. Ou seja, podemos testar qualquer código que estamos escrevendo diretamente dentro do REPL.

Também podemos iniciar o REPL com qualquer `jar` arbitrário no `classpath`, e acessar as APIs desse `jar` facilmente — no caso, sem escrever uma única linha de código Scala, já podemos usar qualquer *API Java* no REPL. No código a seguir, iniciamos o REPL com o `jar` do `joda-time` e o `do joda-convert` no `classpath`:

```
scala -classpath joda-time-2.3.jar:joda-convert-1.6.jar
```

DEPENDÊNCIAS OPCIONAIS NA COMPILAÇÃO

Para o suporte a anotações, o `joda-time` depende do `joda-convert`. Na linguagem Java, esse tipo de dependência sempre foi opcional em tempo de compilação. Ou seja, só precisamos adicionar essa dependência ao *classpath* se realmente a estivermos usando. No caso do `joda-time`, isso significa que se não usamos anotações, não precisamos do `joda-convert`.

Já o compilador Scala é mais rigoroso e, até a versão 2.9.x, ele exigia a presença dessa dependência opcional, mesmo que não a usemos na prática. A partir da versão 2.10.x, isso não é mais necessário. A compilação vai funcionar e o compilador emitirá um *warning*, algo parecido com o seguinte do caso do `joda-time`:

```
warning: Class org.joda.convert.FromString not found - continuing with a stub.
warning: Class org.joda.convert.FromString not found - continuing with a stub.
warning: Class org.joda.convert.ToString not found - continuing with a stub.
warning: Class org.joda.convert.ToString not found - continuing with a stub.
warning: Class org.joda.convert.ToString not found - continuing with a stub.
```

Feito isso, podemos interagir com toda a API do `joda-time`. A seguinte iteração no REPL ilustra como podemos obter a data atual:

```
scala> import org.joda.time._
import org.joda.time._

scala> val hoje = new LocalDate()
hoje: org.joda.time.LocalDate = 2014-07-18
```

Ou seja, simplesmente usamos a API da forma que já sabemos

fazer, inclusive fazendo o tradicional `import` . Só precisamos conhecer um pouco da sintaxe da linguagem Scala.

1.6 INFERÊNCIA DE TIPOS

Antes de mergulhar de forma mais profunda nos recursos da linguagem Scala, vamos analisar brevemente um recurso que estaremos usando de uma forma ou de outra o tempo todo, talvez mesmo sem perceber: a *inferência de tipos*.

Scala é uma linguagem estaticamente tipada, ou seja, os tipos das nossas constantes, variáveis etc. são definidos em tempo de compilação e, a partir daí, não podem mais ser alterados. Mesmo assim, nos exemplos anteriores, talvez o leitor tenha ficado com a impressão de Scala ser dinâmico. Isso porque, em momento algum, nós definimos explicitamente os tipos. E aí está o ponto-chave: *explicitamente* em vez de *implicitamente*.

Como algumas outras linguagens modernas, o que Scala faz é inferir os tipos das variáveis baseando-se nos valores que tentamos colocar nelas — ou seja, baseado no contexto no qual elas são definidas. Voltando ao nosso exemplo da soma de dois números no REPL:

```
scala> 1 + 2  
res0: Int = 3
```

```
scala>
```

Como mencionamos antes, aqui Scala declara uma nova variável para o resultado da soma. Repare que o REPL está nos dizendo o tipo dessa variável: `Int` . Ou seja, `res0` é do *tipo inteiro* e não poderia receber nenhum outro tipo de valor. Para entender isso melhor, vamos armazenar o resultado da soma em uma variável pré-definida por nós, e tentar colocar outro valor nela. Vamos ter de usar a palavra-chave `var` para isso.

VAL VERSUS VAR

Resumidamente, `val` indica uma variável que não pode mudar, ou seja, uma *constante*, enquanto `var` seria uma variável tradicional.

```
scala> var soma = 1 + 2
soma: Int = 3

scala> soma = "3"
<console>:8: error: type mismatch;
 found   : String("3")
 required: Int
    soma = "3"
           ^

scala>
```

O erro anterior, *type mismatch*, é bem claro: tentamos colocar uma `String` em um inteiro e a variável `soma` é um inteiro — e não pode mais mudar.

A inferência de tipos nos ajuda a evitar códigos desnecessariamente burocráticos, pois não precisamos dizer o óbvio para o compilador. Mesmo assim, em alguns casos, como quando lidamos com *classes e heranças* ou para parâmetros de funções, ou então simplesmente para *melhorar a legibilidade* (sim, em alguns casos adicionar o tipo torna o código mais legível), podemos querer ou até mesmo precisar definir o tipo *explicitamente*. Isso é bastante simples de se fazer:

```
scala> var soma: Int = 1 + 2
soma: Int = 3

scala>
```

É apenas questão de se acostumar com o formato `nome: tipo`, que é o oposto em relação ao que fazemos em Java, por exemplo,

onde primeiro definimos o tipo da variável e só então o nome dela.

1.7 UM POUCO DE HISTÓRIA

A linguagem Scala foi criada por Martin Odersky, e significa *Scalable Language*. Odersky trabalhou em diversas linguagens e compiladores, entre eles o próprio *javac* atualmente em uso, e no *Java Generics*.

Além das experiências anteriores, dois outros projetos liderados por Odersky influenciaram a criação do Scala: a linguagem *Funnel* (<http://lampwww.epfl.ch/funnel/>), uma linguagem funcional altamente acadêmica e implementada em Java; e *Pizza* (<http://pizzacompiler.sourceforge.net/>), uma extensão para a linguagem Java que adicionava, entre outras coisas, Programação Funcional.

Os projetos de pesquisa de *Martin Odersky* foram criados na EPFL (*École Polytechnique Fédérale de Lausanne*), na Suíça, incluindo *Funnel* e o próprio Scala. Como muitos projetos de pesquisa, Scala começou sendo pouco adequado para ambientes de produção. Mas isso mudou muito desde a versão 2 da linguagem e, em especial, desde que o *Twitter* começou a usar a linguagem, em 2007. Uma entrevista com alguns desenvolvedores mostra um pouco como o *Twitter* usa Scala: http://www.artima.com/scalazine/articles/twitter_on_scala.html.

Ser desenvolvida por uma equipe de pesquisa poderia ser uma desvantagem em alguns cenários. Por exemplo, se uma grande empresa resolver adotar a linguagem e precisar de ajuda, mesmo que comercialmente, quem pode oferecer tal suporte? Existem empresas que oferecem tais serviços, mas são confiáveis?

Para resolver esse problema, e melhorar o ciclo de evolução da

linguagem Scala, *Martin Odersky* e *Jonas Bonér* criaram a *Typesafe*. Ela é uma empresa que recebeu investimentos do mercado financeiro e hoje é responsável por garantir o futuro da linguagem Scala e alguns frameworks importantes, como o *Akka* e o *Play Framework*.

Mencionamos que o nome Scala significa *Scalable Language*. O detalhe é que o *escalável* no nome significa que a linguagem em si é escalável, em termos de recursos — e não que sistemas escritos em Scala são automaticamente escaláveis. Por exemplo, uma das formas de escrever sistemas altamente concorrentes, que vem ganhando muita atenção nos últimos anos, é usar o *Modelo de Atores* — o que o framework *Akka* oferece.

Scala adicionou suporte ao modelo de atores como uma simples API, provando ser possível adicionar novos recursos à linguagem, mesmo que complexos, sem ter de inventar novos operadores, palavras-chave etc. Foram usados apenas recursos que programadores normais (e não hackers de compiladores) poderiam usar.

Também não estamos dizendo que programas escritos em Scala não são escaláveis. Na verdade, Scala nos dá todo o poder que precisamos para desenvolver tais sistemas, e veremos muito disso no decorrer do livro.

ACESSANDO FOTOS DO FLICKR

2.1 CONHECENDO NOSSO PROBLEMA

Novo objetivo no decorrer do livro será consumir web services RESTful do Flickr, para acessar diversas informações sobre fotos armazenadas nesse serviço. Como veremos adiante, acessar esses serviços é extremamente simples; mesmo assim é um problema interessante de ser explorado devido à grande quantidade de informações disponíveis para serem consultadas.

Podemos acessar desde a lista de fotos de um usuário até fazer pesquisas de fotos por tags entre todas as fotos publicadas. E ao mesmo tempo, também poderemos ver diversos recursos da linguagem Scala em ação.

Mas antes mesmo de acessar qualquer serviço, vamos preparar o projeto para lidar com as informações que vamos receber. Caso contrário, estaríamos basicamente jogando um monte de texto no usuário, e nosso sistema seria pouco útil.

Para dar um gostinho do que é possível fazer, vejamos um pequeno exemplo que vai simplesmente listar na tela a resposta que recebemos do Flickr. Para isso, vamos aprender dois recursos interessantes do Scala.

Primeiro, para escrever uma aplicação em Scala e definir o

ponto de início. Isso seria feito em Java usando o famoso `public static void main`, e nós usamos a `trait App`:

```
object ClienteFlickr extends App
```

Veremos com detalhes mais para a frente o que exatamente o `object` e o `App` significam. O importante agora é que, para implementar nossa aplicação, vamos simplesmente adicionar código do corpo do `ClienteFlickr`:

```
object ClienteFlickr extends App {  
  val apiKey = "sua-api-key"  
  val method = "flickr.photos.search"  
  val tags = "scala"  
}
```

Começamos declarando algumas variáveis com informações que vamos precisar para acessar o serviço:

- `apiKey`: sua API de desenvolvedor Flickr. Você pode solicitar uma em <http://www.flickr.com/services/api/keys/apply/>.
- `method`: serviço que vai ser acessado. Neste caso, vamos acessar o serviço de busca de fotos; uma lista completa com os métodos suportados pode ser encontrada em <https://www.flickr.com/services/api/>.
- `tags`: Vamos filtrar nossa busca por tags e, por enquanto, vamos definir uma tag nessa constante.

O próximo recurso que vamos utilizar é a interpolação de `Strings`. Esse é um recurso muito poderoso que pode fazer muito mais do que interpolar `Strings`. Entretanto, nesse caso a simples interpolação é mais do que suficiente para o que precisamos:

```
val url = s"https://api.flickr.com/services/rest/?method=$method&api_key=$apiKey&tags=$tags"
```

Usamos o `s` antes das aspas duplas para indicar que vamos usar interpolação de `Strings`. Em seguida, dentro das aspas,

indicamos o uso de cada variável que queremos interpolar na `String` com cifrão (`$`).

O mais interessante aqui é que o compilador valida o acesso às variáveis. Se digitarmos o nome de alguma delas errado, o código não vai nem mesmo compilar. E podemos também utilizar expressões completas, bastando para isso usar `${expressao}` — ou seja, precisamos apenas usar as chaves junto com o cifrão.

Agora que já temos todas as informações que precisamos, vamos acessar o serviço usando o objeto `scala.io.Source` e imprimir tudo o que recebermos como resposta:

```
scala.io.Source.fromURL(url).getLines().foreach(println)
```

O `scala.io.Source` nos oferece métodos para ler informações de diversas origens diferentes, como arquivos, URLs, `InputStream`s etc. Nosso programa completo ficaria então:

```
object ClienteFlickr extends App {  
  val apiKey = "sua-api-key"  
  val method = "flickr.photos.search"  
  val tags = "scala"  
  
  val url = s"https://api.flickr.com/services/rest/?method=$method  
&api_key=$apiKey&tags=$tags"  
  
  scala.io.Source.fromURL(url).getLines().foreach(println)  
}
```

Para compilá-lo e em seguida executá-lo, basta usar os comandos a seguir:

```
scalac ClienteFlickr.scala  
scala ClienteFlickr
```

O resultado será uma lista de fotos do Flickr em formato XML. Só não esqueça de trocar o valor da `apiKey` por um válido. Caso contrário, em vez de uma lista de fotos, o resultado será algo como:

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<rsp stat="fail">
  <err code="100" msg="Invalid API Key (Key has invalid format)"
/>
</rsp>
```

Com a `apiKey` correta, a resposta recebida será algo como a seguir (resumido por questão de espaço):

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page="1" pages="666" perpage="100" total="66585">
  <photo id="25046413926" owner="119192561@N02" secret="d5764b727f"
server="1535"
    farm="2" title="Scala" ispublic="1" isfriend="0" isfamily=
"0"/>
  <photo id="24702428649" owner="130046925@N04" secret="53c63a4475"
server="1467"
    farm="2" title="NY" ispublic="1" isfriend="0" isfamily="0'
/>
  <photo id="24305307114" owner="76999786@N02" secret="af03afc09d"
server="1613"
    farm="2" title="Future starts slow" ispublic="1" isfriend=
"0" isfamily="0"/>
  <photo id="25029759906" owner="124408581@N06" secret="fea9e2a565"
server="1607"
    farm="2" title="Elizabeth Dress & Shoes_003" ispublic=
"1" isfriend="0"
    isfamily="0"/>
  <photo id="24938009862" owner="124408581@N06" secret="9a55753645"
server="1554"
    farm="2" title="Elizabeth Dress & Shoes_004" ispublic=
"1" isfriend="0"
    isfamily="0"/>
  <photo id="24328322599" owner="79920649@N07" secret="1407ca8449"
server="1453"
    farm="2" title="Lonely Robot London Scala 201215" ispublic
="1" isfriend="0"
    isfamily="0"/>
</photos>
</rsp>
```

Para finalizar esse exemplo, vamos ler a `apiKey` de um arquivo de configuração. No código a seguir, estamos assumindo que exista um arquivo chamado `config.properties` no classpath do projeto:

```
val props = new Properties()
props.load(getClass.getClassLoader.getResourceAsStream("cap02/config.properties"))

val apiKey = props.getProperty("flickr.api.key")
```

No código anterior, estamos usando a classe Java `Properties` para fazer a leitura da configuração, ilustrando novamente a integração entre Java e Scala. Em aplicações Scala reais, no entanto, seria mais comum utilizar a biblioteca *Typesafe Config* para isso.

TYPESAFE CONFIG

A biblioteca *Typesafe Config* é uma biblioteca para configuração de software desenvolvida para a JVM. Apesar de ser mais frequentemente vista em aplicações Scala, ela foi desenvolvida apenas com Java e pode facilmente ser utilizada por qualquer outra linguagem que seja executada na JVM, incluindo, é claro, a própria linguagem Java.

Recursos como suporte à sintaxe JSON, suporte a valores padrão para chaves de configuração e facilidade para acesso a arquivos de configuração em diversos locais diferentes (como classpath, arquivos em disco e URLs) fazem com que valha a pena pelo menos considerar usá-la. Mais informações podem ser encontradas no GitHub: <https://github.com/typesafehub/config>.

2.2 MODELOS E FUNÇÕES

Conforme fomos estudando os principais recursos da linguagem, vamos evoluir um pequeno projeto que acessará e nos permitirá realizar algumas operações interessantes com o serviço do

Flickr. Além de simplesmente acessar listagens de fotos, por exemplo, vamos com facilidade parsear, agregar, filtrar etc. as informações de diversas formas.

Um exemplo bem simples. As fotos do Flickr são hospedadas em *farms* (fazendas), e essa informação está disponível na resposta que vimos no exemplo anterior. Vamos agora descobrir quantas fotos, entre as que foram retornadas na busca, estão hospedadas na *farm* 2.

Primeiro, criaremos uma forma de representar cada foto:

```
class Foto(val id: Long, val owner: String, val title: String, val farm: Int)
```

Esse código é bem óbvio, mas não se preocupe se não foi possível entendê-lo completamente: vamos estudar classes em detalhes no próximo capítulo. Também não se incomode com o fato de não estarmos representando todas as informações disponíveis, pois esse modelo ainda será evoluído no decorrer do livro.

Mesmo sendo muito simples, o modelo anterior já vai nos permitir trabalhar com as informações de determinadas fotos. O ponto mais complexo deste exemplo é transformar o resultado do serviço em classes, e isso veremos apenas mais para o final do livro. Portanto, vamos resumir essa funcionalidade em uma linha apenas, por enquanto:

```
val fotos = parseiaResultado(resultadoServico)
```

Agora vem a parte mais interessante, que é responder à nossa pergunta: quantas fotos estão hospedadas na *farm* 2?

```
val countFotosFarm7 = fotos.count(foto => foto.farm == 2)
```

Pronto! Código extremamente simples e legível. Mesmo que o leitor ainda não tenha muito conhecimento de *Programação Funcional* ou da linguagem Scala, deve ser possível entender o que está acontecendo.

Por fim, nosso exemplo completo:

```
class Foto(val id: Long, val owner: String, val title: String, val
farm: Int)

val fotos = parseiaResultado(resultadoServico)
val countFotosFarm7 = fotos.count(foto => foto.farm == 2)
```

2.3 COMO SERIA EM JAVA?

Como uma pequena comparação, vejamos como seria fazer a filtragem que fizemos anteriormente com Java 7. Existem obviamente muitas formas de se fazer isso e, em muitos casos, escolher a melhor forma vai depender do cenário sendo trabalhado.

No nosso caso, vamos espelhar o comportamento do `filter` que observamos: vamos encontrar todos os elementos de uma coleção, que se encaixem em um determinado critério. Primeiro, vamos criar a classe `Foto`, lembrando de que ela será imutável, como fizemos em Scala:

```
public class Foto {
    private final Integer id;
    private final String owner;
    private final String title;
    private final Integer farm;

    public Foto(Integer id, String owner, String title, Integer farm) {
        this.id = id;
        this.owner = owner;
        this.title = title;
        this.farm = farm;
    }

    public Integer getId() {
        return id;
    }

    public String getOwner() {
        return owner;
    }
}
```

```

public String getTitle() {
    return title;
}

public Integer getFarm() {
    return farm;
}
}

```

Notamos claramente que Scala nos permite economizar código "óbvio", o famoso *boiler-plate*. Assim como fizemos no exemplo Scala, vamos supor que temos uma coleção de fotos pronta. Vamos então encontrar todas as fotos da *farm número 7*:

```

Collection<Foto> fotosFarm7 = new LinkedList<>();
for (Foto foto : fotos) {
    if (foto.getFarm() == 7) {
        fotosFarm7.add(foto);
    }
}

```

Temos diversos problemas nessa solução. Primeiro, a lógica que determina o critério do filtro está enterrada no *loop*, e dificilmente poderá ser reusada. Mesmo eventualmente extraíndo métodos ou usando técnicas similares, o código não ficaria tão claro quanto o que vimos na versão Scala.

Outro problema é que nós mesmos tivemos de declarar a coleção resultante, e esta precisou ser mutável. Conseguimos não alterar a coleção original, mas o fato de estarmos manipulando a coleção resultante torna a lógica anterior monolítica, inseparável.

Na versão Scala, só temos acesso à coleção resultante quando ela já está pronta. O processo utilizado para criá-la e preenchê-la é detalhe de implementação. Isso permite que a API seja otimizada internamente sem causar problemas para o nosso código. Como poderíamos aplicar a lógica de filtragem em questão em paralelo, por exemplo?

Em resumo, na versão Scala, nós estamos dizendo ao

compilador *o que* queremos que seja feito, enquanto que, na versão Java, além de dizer o que queremos, também temos de dizer *como* queremos que isso seja feito.

O mundo Java hoje já não está tão ruim como no exemplo anterior, como veremos na seção a seguir. Não seria exagero dizer que Scala tem também influenciado a linguagem Java a melhorar.

2.4 COMO SERIA NO JAVA 8?

No Java 8, finalmente temos estruturas funcionais na linguagem. Se traduzirmos o último exemplo da seção anterior de Java 7 para 8, teríamos o seguinte código:

```
fotos.stream().filter((foto) -> (foto.getFarm() == 7)).forEach((foto) -> {  
    fotosFarm7.add(foto);  
});
```

Sem entender nada de *Programação Funcional*, esse código deve parecer um pouco estranho em um primeiro momento. Leia-o com calma e perceba que agora a lógica mudou um pouco, com muito mais foco em *o que* precisa ser feito em vez de *como* fazer.

Não é tão limpo quanto em Scala, porém já é muito interessante para quem não tiver opção de trocar de linguagem, mas puder utilizar Java 8. E assim como *Orientação a Objetos*, *Programação Funcional* é um conceito que não está amarrado à linguagem: uma vez que o conceito seja entendido, é fácil aplicá-lo a qualquer outra linguagem funcional.

O exemplo anterior ainda não é o melhor que podemos fazer com Java 8, pois ainda estamos manualmente adicionando o resultado do filtro na coleção final. O ideal é recebermos a coleção resultante do filtro diretamente, algo que fazemos frequentemente em Scala. Implementar isso em Java 8, entretanto, fica como um

exercício para o leitor curioso.

Com isso tudo em mente, o leitor já pode ir se habituando com alguns termos apresentados nos exemplos deste capítulo, como `filter` e `forEach`, e outros como `map`, `fold` e `reduce`, que são elementos comuns e muito importantes em diversas *linguagens de programação funcionais* que exploraremos nos próximos capítulos.

CLASSES E OBJETOS

3.1 NOSSA PRIMEIRA CLASSE

Neste capítulo, vamos estudar com detalhes classes e objetos na linguagem Scala. Para começar, vamos definir uma classe simples, que encapsulará o acesso à API do Flickr.

```
class FlickrCaller
```

Este código é tudo o que precisamos para definir uma classe simples. Se a classe não tiver corpo, não precisamos nem das chaves que vamos usar a seguir. Isso é algo para sempre pensarmos quando estamos escrevemos código Scala: chaves são muitas vezes opcionais e, se o código resultante for claro, é melhor omiti-las.

É claro que uma classe sem nenhum corpo é geralmente pouco útil, então vamos adicionar alguns atributos e métodos. Primeiro um atributo, declarado com a palavra-chave `val`.

```
class FlickrCaller {  
    val apiKey = "sua-api-key"  
}
```

Temos diversas coisas acontecendo aqui. Primeiro, `apiKey` é um atributo do tipo `String`. Como mencionamos rapidamente no capítulo anterior, Scala é uma linguagem estaticamente tipada, então nossas variáveis e atributos não podem ficar sem um tipo definido. Mesmo assim, não tivemos de declarar esse tipo: a linguagem o inferiu. A declaração anterior, na prática, é exatamente

igual ao exemplo a seguir:

```
class FlickrCaller {  
    val apiKey: String = "sua-api-key"  
}
```

Neste caso, estamos usando `:` (dois pontos) para separar o nome do atributo do seu tipo. Estamos adicionando um pouco de verbosidade para tornar o tipo mais explícito. Na grande maioria das vezes, isso não é necessário, mas se você quiser tornar o tipo fixo e *documentado*, a notação explícita pode ser útil, especialmente quando estamos falando de APIs públicas — ou seja, que serão usadas por outros desenvolvedores além do autor do código em questão.

3.2 VAL VS. VAR

O segundo ponto que precisamos prestar atenção no exemplo anterior é o uso da palavra-chave `val`. Ela na verdade não está simplesmente declarando uma *variável*: ela está declarando uma *constante*! Isso quer dizer que, uma vez definido, o valor da nossa `apiKey` não poderá mais mudar. Se iniciarmos uma sessão no REPL e digitarmos o código a seguir, veremos isso claramente:

```
val apiKey = "api-key"  
apiKey = "nova-key"
```

Nesse caso, o REPL nos apresentará o seguinte erro:

```
error: reassignment to val
```

O erro deixa bem claro que de fato não podemos alterar o valor do `val`. Se realmente precisarmos de um atributo, ou *variável*, que possa ser alterado, precisamos usar a palavra-chave `var`. Com `var`, poderíamos fazer o exemplo anterior funcionar:

```
var apiKey = "api-key"  
apiKey = "nova-key"
```

Dessa vez, o código funciona normalmente. A única diferença é que `var` s podem ter seus valores alterados. A inferência de tipos, por exemplo, continua funcionando normalmente e nossa `apiKey` continua sendo entendida (*inferida*) como uma `String`.

Quando declaramos atributos ou variáveis, portanto, a primeira coisa que Scala nos obriga a considerar é: queremos uma *variável* de verdade, ou uma "*variável que não varia*", ou seja, uma *constante*? Essa não é uma questão tão simples ou pequena quanto pode parecer a princípio.

Usar `val` pode parecer uma limitação e o nosso primeiro impulso, principalmente quando começamos a estudar Scala, pode ser usar `var` em quase todos os casos. Mas isso é exatamente o pensamento oposto à boa prática que deveria ser aplicada a qualquer linguagem funcional — tanto que algumas dessas linguagens, como *Haskell*, possuem por padrão o comportamento do `val` — e obrigam o desenvolvedor a explicitamente mudar o comportamento padrão, se for o que realmente querem fazer.

Portanto, a recomendação é dar preferência para `val` s sempre que possível. Como em Java, onde usar `final` é uma boa prática, mesmo que poucos desenvolvedores o façam, usar `val` traz alguns benefícios importantes. Entre eles:

- Podemos compartilhar o atributo entre várias *threads*, sem medo de que alguma delas altere o seu valor e cause erros extremamente difíceis de se encontrar;
- Facilita o "pensamento funcional".

O segundo item fará mais sentido quando falarmos de programação funcional mais a fundo. Porém o ponto é que, com `val` s, pensamos menos em fazer "primeiro isso, depois aquilo, agora o valor está correto" (pensamento imperativo) e pensamos mais em simplesmente usar o valor que, se existir, deverá estar

correto

3.3 MÉTODOS E FUNÇÕES

Vamos agora adicionar um pouco de funcionalidade, ou comportamento, à nossa classe. Para declarar um método ou função, usamos a palavra-chave `def`, seguida do nome do método. Em seguida, colocamos a lista de parâmetros, entre parênteses, e o tipo de retorno desejado. Vejamos um pequeno exemplo a seguir:

```
def buscaFotos(tag: String): Seq[Foto] = ???
```

Os elementos na lista de parâmetros são separados por vírgulas, na mesma forma da declaração de variáveis: primeiro o nome, depois o tipo, separados por ponto e vírgula. E no final da declaração, temos `:` (dois pontos) e o tipo de valor retornado pelo método.

???

O `???` visto em alguns exemplos a partir deste capítulo não é um artifício especial do livro, nem da linguagem Scala, nem mesmo um erro de digitação. O `???` pode ser usado sempre que não sabemos ainda o que vamos colocar em alguma variável ou na implementação de um método. > Falaremos mais disso no capítulo 5.. Por enquanto, basta saber que o `???` é um método que está sempre disponível para nosso código e pode substituir qualquer outro código. Quando invocado, ele simplesmente lança uma exceção.

Vamos fugir da nossa aplicação de exemplo do Flickr por um momento, para ilustrar um outro ponto importante: tipos de

retorno são opcionais. Ou seja, podem ser omitidos e, portanto, terem seus valores inferidos pelo compilador, assim como acontece com variáveis. Veja o seguinte exemplo:

```
def soma(x: Int, y: Int) = x + y
```

O exemplo anterior é equivalente ao código a seguir:

```
def soma(x: Int, y: Int): Int = x + y
```

Convém mencionar aqui um pouco de boas práticas: tome cuidado ao omitir o tipo de retorno de métodos. Se o método fizer parte de uma API pública, por exemplo, prefira definir esses tipos explicitamente. Isso é importante por dois motivos principais:

- Documenta o método para o leitor do código;
- Torna mais difícil alterar o tipo de retorno por acidente.

Documentar um método é sempre interessante, como já mencionamos. E tornar alterações acidentais mais difíceis é especialmente importante quando o método em questão é acessado por outros projetos. Isso porque tal alteração pode fazer com que esses projetos parem de funcionar corretamente, pois estaríamos quebrando a API da qual esses projetos dependem.

O código a seguir muda o tipo de retorno do exemplo anterior de `Int` para `Double`, de forma quase imperceptível:

```
def soma(x: Int, y: Int) = x + y + 1.0
```

Agora preste atenção no uso das chaves no código anterior — ou melhor, na ausência delas. Quando o corpo de um método possui apenas uma linha, podemos omitir as chaves. Mas se esse não for o caso, elas são obrigatórias.

Buscar uma foto no serviço do Flickr provavelmente precisará de mais do que uma linha de código, então vamos revisar aquele

método:

```
def buscaFotos(tag: String): Seq[Foto] = {  
  // algoritmo (complexo?) de acesso ao Flickr aqui  
  ???  
}
```

Agora temos duas linhas de código, um comentário representando a lógica de acesso, que vamos implementar em um capítulo futuro, e o `???` representando o retorno do método. Isso nos leva à próxima questão: o que o método retorna?

Quando tínhamos apenas uma linha de implementação, era fácil imaginar que o retorno era simplesmente o resultado dessa linha ou expressão. Com múltiplas linhas, é a mesma coisa, com foco na última linha: o retorno do método será sempre a última linha (ou expressão) executada na sua implementação.

Entretanto, mesmo assim Scala possui uma palavra-chave `return`. Uma nova versão do código anterior seria:

```
def buscaFotos(tag: String): Seq[Foto] = {  
  // algoritmo (complexo?) de acesso ao Flickr aqui  
  return ???  
}
```

Essa versão é muito parecida com a anterior, com algumas diferenças importantes. Primeiro, o tipo de retorno passa a ser obrigatório. O código a seguir não vai compilar:

```
def buscaFotos(tag: String) = {  
  // algoritmo (complexo?) de acesso ao Flickr aqui  
  return ???  
}
```

Outro ponto é a legibilidade, principalmente quando começarmos a lidar e pensar mais em funções em vez de métodos, como pensamento funcional. Pensamos nessas funções como expressões que geram um resultado.

O uso do `return` torna a implementação mais imperativa e difícil de compor. Acabamos mudando o pensamento de *o que* queremos fazer para *como vamos fazer*. A própria palavra `return` nos induz a pensar em ordens, ou seja, de forma imperativa.

Em programação funcional, no entanto, é comum usarmos resultados de outras expressões como o resultado de uma função. Ou o resultado de outra função, que por sua vez é calculado com base em outra expressão ou função, e assim por diante. Um exemplo simples a seguir, fugindo um pouco da API Flickr novamente:

```
def usuarioAtual() = usuarioOpt match {  
  case Some(usuario) => usuario  
  case None => "anônimo"  
}
```

Estamos implementando um método para determinar quem é o usuário logado. Para isso, usamos *pattern matching* — em um primeiro momento, parecido com o `switch` do Java e outras linguagens *C-like*, mas bem mais poderoso —, como veremos em outro capítulo. O *pattern matching* é uma expressão por si só, e o que estamos fazendo então é atribuir o resultado dessa expressão ao resultado da função.

Voltando à questão da declaração da função. Se as chaves são opcionais, por que os parênteses também não são? Veja:

```
def usuarioAtual = usuarioOpt match {  
  case Some(usuario) => usuario  
  case None => "anônimo"  
}
```

Porém, nesse caso, a ausência de parênteses traz uma consequência: funções declaradas com parênteses podem ser invocadas com ou sem eles.

```
def usuarioAtual() = usuarioOpt match {  
  case Some(usuario) => usuario  
  case None => "anônimo"  
}
```

```
val usuario1 = usuarioAtual()  
val usuario2 = usuarioAtual
```

Já funções declaradas sem parênteses só podem ser invocadas sem eles. O código a seguir não compila:

```
def usuarioAtual = usuarioOpt match {  
  case Some(usuario) => usuario  
  case None => "anônimo"  
}  
  
val usuario1 = usuarioAtual()
```

O erro será algo como:

```
Usuario.scala:18: not enough arguments for method apply: (index: I  
nt)Char in class StringOps.  
Unspecified value parameter index.  
usuarioAtual()  
      ^
```

Um dos principais usos desse recurso é permitir criar APIs fluentes nas quais, para acessar determinada informação, é irrelevante se o que estamos acessando é um método ou função (ou seja, um valor calculado), ou alguma variável ou constante. Isso fica transparente; passa a ser detalhe de implementação.

Esse acesso transparente a membros de uma classe é chamado de *Princípio do acesso uniforme*. Em um contexto levemente diferente, esse recurso também é útil na escrita de DSLs — em que saber como determinados valores são gerados é ainda menos importante.

PRINCÍPIO DO ACESSO UNIFORME

A expressão *Princípio do acesso uniforme* foi definido por Bertrand Meyer em seu livro *Object-Oriented Software Construction*. Em suma, esse princípio diz que, quando acessamos um membro qualquer de uma classe ou objeto, não precisamos (nem devemos) saber de onde vem a informação — ou seja, se é por exemplo uma variável ou um valor calculado.

Com isso em mente, poderíamos transformar um membro que era calculado em uma variável, e um membro que era uma variável em um valor calculado, sem que o código cliente seja afetado. Em outras palavras, a notação (sintaxe) utilizada não vaza detalhes de implementação.

Para conseguir esse efeito, algumas linguagens precisam usar de criatividade — quebrando em parte o princípio descrito, mas atingindo a flexibilidade desejada. Em Java, por exemplo, é comum trabalhar com todos os acessos a membros de uma classe sendo feitos através de métodos *getter*. Assim, garantido a flexibilidade mencionada, pois o *getter* pode fazer cálculos ou simplesmente retornar o valor de uma variável privada.

Já em Scala, as coisas são mais simples. Quando lemos um código como `foto.title`, não sabemos se `title` é um atributo ou um método. E temos a liberdade de alterar esse campo para que ele seja o que precisarmos, sem afetar o código cliente. Isso é possível pois métodos sem parâmetros em Scala podem ser invocados sem parênteses.

3.4 CONSTRUTORES

Voltando ao nosso `FlickrCaller`, vamos melhorá-lo para que, em vez de usar uma `apiKey` definida diretamente no código, essa classe receba a `apiKey` em um construtor. Assim, a classe será um pouco mais flexível. Criar esse construtor é tão simples quanto:

```
class FlickrCaller(api: String)
```

Ou seja, a declaração do construtor é mesclada à declaração da classe. Chamamos esse construtor de construtor principal, mas podemos criar construtores auxiliares também.

Suponha que, além da opção de especificar a `apiKey`, também queremos permitir criar objetos dessa classe sem especificar uma `apiKey`:

```
class FlickrCaller(api: String) {  
    def this() = this("")  
}
```

Repare que, na implementação do construtor auxiliar, nós invocamos o construtor principal. Isso é importante. Na verdade, é uma regra da sintaxe da linguagem: quando criamos construtores auxiliares, esses novos construtores devem invocar, ou algum outro construtor auxiliar declarado antes dele, ou o construtor principal.

Desde que sigamos essa regra, podemos ter quantos construtores auxiliares quisermos. Esse novo construtor não fará sentido no nosso exemplo, pois o uso da `apiKey` é obrigatório. Vamos então deixá-lo de fora dos nossos próximos exemplos.

Podemos também transformar o parâmetro em uma variável ou constante. Na forma como a classe `FlickrCaller` está definida no momento, o seguinte código não compilaria:

```
val caller = new FlickrCaller("apiKey")  
println(caller.apiKey)
```

O problema é que o campo `apiKey` é, por padrão, privado. Podemos mudar isso com o seguinte código:

```
class FlickrCaller(val apiKey: String)

val caller = new FlickrCaller("apikey")
println(caller.apiKey)
```

Poderíamos ter também usado `var` em vez de `val`, e as regras de `val` *versus* `var` que discutimos antes continuam valendo. Ou seja, o campo, além de simplesmente visível, seria mutável.

Se lembrarmos dos exemplos do primeiro capítulo, veremos que, quando analisamos o *bytecode* gerado pelo compilador Scala, `val` s fazem com que métodos de acesso ao campo sejam gerados, e `var` s incluem *setters* a esse resultado. Portanto, se analisarmos o *bytecode* do exemplo anterior, o resultado seria algo como:

```
public class FlickrCaller {
    private final java.lang.String apiKey;
    public java.lang.String apiKey();
    public FlickrCaller(java.lang.String);
}
```

No caso, a linha que representa o *getter* ao qual estamos nos referindo é `public java.lang.String apiKey();`.

3.5 PARÂMETROS DEFAULT E NOMEADOS

Um recurso muito interessante da linguagem Scala é o suporte a parâmetros com valores padrão e parâmetros nomeados. Os dois recursos funcionam muito bem para criar APIs mais elegantes e fáceis de serem usadas, e ajudam a diminuir em muito a necessidade de criarmos sobrecargas de métodos ou construtores. Vamos expandir nosso método para busca de fotos:

```
def buscaFotos(tag: String, userId: String) = ???
```

Temos agora dois parâmetros para a busca: `tags` e o *id do usuário*, cujas fotos queremos procurar. Porém, nem sempre queremos especificar os dois valores. Podemos querer todas as fotos

de um usuário, independente das tags, e podemos querer fotos de todos os usuários com determinadas tags (como o que fizemos até agora).

Uma solução típica para esse problema seria criar sobrecargas do método. Entretanto, como já mencionado, Scala oferece uma alternativa mais interessante:

```
def buscaFotos(tag: String = "", userId: String = "") = ???
```

Isso nos permitirá especificar apenas os parâmetros que nos interessam, deixando o valor padrão nos outros casos.

TIPOS OPCIONAIS

O correto seria usar algo como `Option[String]` para `String` s opcionais, e usar `None` como o valor padrão. Vamos discutir isso em detalhes no capítulo 5.

Com a assinatura definida anteriormente, podemos agora invocar o método de diversas formas diferentes, entre elas:

```
buscaFotos("scala")  
buscaFotos(userId = "userid")
```

Repare que, no segundo caso, tivemos de especificar o nome do parâmetro que estamos usando. Se não fosse assim, não seria possível para a linguagem saber que queremos especificar o `userId`, e não a `tag`. Especificar o nome do parâmetro para a `tag` seria válido, mas desnecessário.

Na verdade, independente do tamanho da lista de parâmetros, enquanto estivermos especificando os valores em ordem, não precisamos especificar o nome do parâmetro, mesmo se não especificarmos todos os parâmetros. Podemos também passar parte

dos parâmetros em ordem e sem nomes, e outra parte com nomes. Veja:

```
buscaFotos("scala", userId = "userid")
```

A mesma coisa vale para construtores. Portanto, o código a seguir é válido:

```
class FlickrCaller(api: String = "")  
val caller = new FlickrCaller()
```

3.6 OBJECTS

Muitas vezes precisamos de um local único onde serão definidos alguns métodos ou variáveis — aqueles elementos que serão exatamente os mesmos para todos os objetos de todas as classes — na prática, elementos globais.

Muitos projetos acabam tendo classes `Util`, `Helper` ou algo do tipo, em que tais elementos comuns são acumulados. Na linguagem Java, temos duas formas para implementar isso: *métodos estáticos*, ou *objetos singleton*. Cada um tem suas vantagens e desvantagens, e discuti-las foge do escopo do livro.

Para o momento, só é importante entender que essa distinção não existe em Scala, por dois motivos complementares:

- Não existem *elementos estáticos* na linguagem Scala;
- Scala oferece suporte nativo para criação de singletons.

Por tanto, em Scala, sempre que queremos um local único e global para definições de constantes e funções comuns, criamos um *singleton*. Implementar *singletons* é uma tarefa relativamente simples, porém tem suas armadilhas — e é fácil ter uma implementação falha. Esse risco é inexistente em Scala pois, como mencionado anteriormente, ela suporta *singletons* nativamente.

Vamos supor que, em vez de trabalhar apenas com fotos, vamos suportar outros tipos de mídia. Poderíamos criar um `object` com as definições dos tipos de mídias suportadas da seguinte forma:

```
object Media {  
  val fotos = "fotos"  
  val videos = "videos"  
  val todas = "all"  
}
```

Usamos a palavra-chave `object` para declarar o *singleton* — em terminologia Scala, dizemos apenas que estamos declarando um objeto. Como esse objeto é um singleton, não podemos criar novos objetos a partir dele — ele não é uma classe. Os elementos definidos em seu corpo devem ser acessados diretamente. O seguinte código não compila:

```
val m = new Media()
```

Para acessar as informações do objeto, fazemos exatamente como faríamos com membros estáticos em Java:

```
println(Media.fotos)  
println(Media.videos)  
println(Media.todas)
```

O resultado será a impressão dos valores dos `val`s em questão, na tela. Internamente, o compilador Scala acaba tendo de gerar membros estáticos. Isso porque, como mencionamos no primeiro capítulo, Scala é executado na JVM, onde *singletons* não existem nativamente.

Quais membros são realmente gerados depende de outros detalhes, mas de uma forma ou de outra eles vão acabar existindo. O que o Scala está fazendo na prática é tirar a responsabilidade de implementar um novo *singleton* a todo momento das nossas mãos. Podemos nos concentrar em fazer um design mais correto.

3.7 CLASSES ABSTRATAS

Um outro tipo de classe que as linguagens de programação orientadas a objetos costumam oferecer são as *classes abstratas*. Classes que são incompletas, usadas basicamente como blocos para nos ajudar na criação de classes mais complexas.

Vamos usar isso a nosso favor para criar um modelo que represente os tipos de mídia de forma mais robusta e flexível. Primeiro, vamos criar a classe abstrata `Media`, que será a base para todos os tipos de mídia suportados:

```
abstract class Media(val value: String)
```

É exatamente a mesma estrutura que vimos antes, com o acréscimo da palavra chave `abstract`. Não podemos criar novos objetos a partir dessa classe, mas podemos estendê-la para criar os tipos de mídia que queremos suportar.

```
object Fotos extends Media("fotos")
object Videos extends Media("videos")
object Todas extends Media("all")
```

Criamos um `object` para cada formato de mídia suportado, e agora temos uma forma limpa de lidar com essa informação. E faz todo o sentido que esses formatos sejam representados por `objects`, pois não precisamos de mais de um objeto para cada formato.

Esse tipo de uso de classes abstratas e `object` `s` é bem parecido com o que faríamos com `enum s` em Java. Mas ainda temos uma decisão importante a tomar: queremos permitir que o usuário da API crie novos tipos de mídia? Provavelmente não. Veremos como resolver isso, além de outros detalhes, no capítulo 10. *Classes abstratas e traits*.

CASE CLASSES E PATTERN MATCHING

Vimos no capítulo anterior como trabalhar com classes e um pouco sobre `objects`. Na prática, vimos como mapear conceitos básicos de Orientação a Objetos na linguagem Scala.

Vamos agora abordar um recurso mais específico, *case classes*, e de quebra também vamos investigar um pouco o suporte a *pattern matching* — um recurso muito usado no dia a dia de qualquer programador Scala.

4.1 CASE CLASSES

Case classes são classes que trazem diversos recursos extras em relação às classes normais. O termo `case` pode lembrar um pouco de blocos `switch / case` da linguagem Java, por exemplo, e esse realmente é um dos usos de `case class`: facilitar o uso de *pattern matching*.

É muito comum precisarmos criar classes com diversos atributos, fazer comparações entre objetos dessas classes, manipulá-los dentro de listas ou `hashmaps`, copiá-los etc. Há diversos usos comuns e recorrentes para classes simples, que geralmente fazem parte do nosso modelo. Se estiver familiarizado com DDD (*Domain Driven Development*), pense nas *Value Classes*. Nesse tipo de cenário, a `case class` se encaixa muito bem.

Mas, afinal, o que essas classes nos oferecem? Primeiro, implementações confiáveis dos clássicos métodos `equals`, `hashCode` e `toString`. Se o leitor é programador Java, certamente já teve de sobrescrever estes métodos alguma vez, e muito provavelmente mais do que uma única vez. Em Scala, podemos usar case classes e evitar esse trabalho tedioso.

COMPARANDO IGUALDADE

Java tem um problema que causa muitos transtornos: comparar dois objetos com o operador `==` compara as referências dos objetos, e não seus valores ou atributos. Em Java, para termos uma comparação correta entre dois objetos, precisamos sobrescrever o método `equals` e invocá-lo explicitamente.

Já o operador de igualdade (`==`) do Scala invoca o método `equals` do objeto em vez de fazer como Java faria e comparar as referências, mesmo existindo uma forma conhecida de fazer a comparação correta. Ainda precisamos sobrescrever o método `equals` para definir a forma correta de se comparar dois objetos, mas Scala se encarrega de invocar esse método se usarmos o `==`.

Isso é possível pois Scala trata o `==` (e vários outros operadores) como um método qualquer, e o sobrescreve na classe `AnyRef`, a base para todas as classes em Scala.

Voltando ao nosso exemplo, o Flickr na verdade também suporta vídeos, além de fotos. A API nos permite especificar que tipo de mídia estamos buscando: fotos, vídeos ou ambos. Vamos fazer uma modelagem inicial para representar essa informação, e

melhorá-la mais para a frente. Voltando ao nosso exemplo, o Flickr na verdade também suporta vídeos, além de fotos. A API nos permite especificar que tipo de mídia estamos buscando: fotos, vídeos ou ambos. Vamos fazer uma modelagem inicial para representar essa informação, e melhorá-la mais para a frente.

No código a seguir, declaramos uma `case class` para nossos tipos de mídias e, em seguida, criamos objetos a partir dessa classe:

```
case class Media(value: String)

val fotos = Media("fotos")
val videos = Media("videos")
val all = Media("all")

fotos == new Media("fotos") // true
videos == new Media("videos") // true
fotos == videos // false

println(videos)
```

O resultado do `println` do exemplo anterior será `Media(videos)`. Note que, na declaração das constantes anteriores, não usamos o operador `new` para criar os objetos. Vamos entender melhor como isso funciona depois, mas é importante saber que isso só foi possível por estarmos usando `case classes` que, por padrão, possuem um método especial chamado `apply` (classes normais nem compilariam).

MÉTODO APPLY

O código anterior funciona porque `case classes` possuem um método chamado `apply`, gerado pelo compilador. Podemos também criar esse método manualmente em qualquer `class` ou `object` para obter um efeito similar e fazer com que o código compile corretamente.

Outro momento no qual *case classes* facilitam nossa vida é quando queremos copiar objetos. A JVM oferece o método `clone`, mas seu uso é complexo e pouco recomendado na prática. Se quiser entender melhor o porquê, o livro *Effective Java*, escrito por Joshua Bloch, explica muito bem.

Como solução para o problema da cópia, duas técnicas são comuns: criar métodos de cópia, ou criar construtores de cópia. No caso, *case classes* já nos oferecem métodos de cópia prontos. Esses métodos, chamados convenientemente de `copy`, unidos com o uso de parâmetros nomeados, são extremamente úteis e flexíveis. Vejamos um exemplo a seguir:

```
val fotos = Media("fotos")
val videos = fotos.copy(value = "videos")

fotos == videos
```

O resultado da comparação no exemplo anterior será `false`. O método `copy` cria um novo objeto, com o novo `value`. Portanto, `fotos` e `videos` são dois objetos completamente diferentes e independentes.

Neste caso, o método `copy` pode não parecer muito útil, mas imagine usá-lo com *case classes* com vários atributos — fica bem mais interessante e útil. Você pode especificar apenas os parâmetros que quer alterar e os demais ficam iguais aos valores no objeto original.

Um detalhe muito importante: os objetos retornados são sempre objetos novos — os originais não são alterados de forma alguma. Esse é um pequeno exemplo do poder da imutabilidade, algo muito valorizado em programação funcional. Não importa de onde o objeto veio, não corremos o risco de introduzir bugs no sistema enquanto manipulamos tais objetos.

4.2 CASE OBJECTS

Além dos `object`s tradicionais que vimos no capítulo anterior, temos os `case objects`. Como `case classes`, eles são basicamente `object`s com recursos extras, como a implementação mais legível do `toString`. Porém, existe uma limitação importante: eles não podem ter parâmetros, ou seja, não pode ter um construtor — o que faz sentido, afinal, nunca criamos instâncias de `object`s. O código a seguir não compila:

```
case object Media(value: String)
```

Isso na verdade serve para qualquer tipo de `object`, seja `case` ou não: não seria possível passar parâmetros para o construtor pois, como mencionamos, nunca construímos esses objetos diretamente. Logo, Scala não nos permite nem tentar fazer tal declaração.

Usando o exemplo anterior de tipos de fotos, poderíamos refatorá-lo para o seguinte:

```
class Media(value: String)

case object Fotos extends Media("fotos")
case object Videos extends Media("videos")
case object Todas extends Media("all")
```

Agora sim temos uma declaração de *case objects* válida. Não podemos definir parâmetros para o construtor do objeto, mas podemos utilizar normalmente os construtores de eventuais superclasses. E agora, como estamos definindo `case object`s, o `toString` seria bem mais agradável, por exemplo. Em vez de imprimir a referência, teríamos algo como a seguinte iteração no *REPL*:

```
scala> println(Videos)
Videos
```

Ou seja, o nome do `object` é impresso.

4.3 PATTERN MATCHING COM CASE CLASSES E OBJECT

Na lista de recursos que nos trazem benefícios imediatos logo que começamos a usar Scala está *pattern matching*. À primeira vista, *pattern matching* não parece ser muito mais do que um `switch` do Java um pouco mais flexível.

Se tivéssemos, por exemplo, criado constantes para identificar os tipos de mídia no exemplo anterior, poderíamos ter usado *pattern matching* para verificar o tipo da mídia sendo manipulado. Vamos então declarar as constantes:

```
val FOTOS = 1
val VIDEOS = 2
val TODAS = 3
```

Essa é uma técnica até que comum, muito usada principalmente antes de Java suportar `enums` — que no fim das contas acabam sendo um pouco mais do que açúcar sintático para obter o mesmo resultado (estamos obviamente ignorando aqui outras vantagens do uso de enumerations). Vejamos agora como ficaria o *pattern matching*:

```
val midia = 1
midia match {
  case FOTOS => println("processando fotos")
  case VIDEOS => println("processando videos")
  case _ => println("processando qualquer outra coisa")
}
```

Para quem está acostumado com Java, estes recursos não apresentam nenhuma novidade. Vamos então aproveitar para entender a sintaxe.

Usamos a palavra-chave `match` para indicar que estamos entrando em uma operação de *pattern matching*. Em seguida, temos os `case`s, que representam cada tentativa de encontrar um padrão

no valor sendo analisado — a mídia, neste caso.

É no *case* do *pattern matching* que temos a primeira diferença em relação à linguagem Java: não existe aqui a necessidade (nem possibilidade) de um `break`. O primeiro *case* que bater com o valor sendo analisado será executado. E só.

A melhor forma de entender isso é imaginar a estrutura do *pattern matching* como uma forma de mapear (ou transformar) um valor para um outro valor qualquer. Isso fica mais evidente quando percebermos que a instrução como um *todo* tem valor, e esse valor pode ser armazenado em uma variável. Ou seja, o *pattern matching* *todo* é basicamente uma expressão.

Se quisermos, por exemplo, armazenar o texto em vez de imprimi-lo, e deixar para imprimir o resultado depois que o *pattern matching* finalizar, podemos alterar o código anterior para algo como o seguinte:

```
val midia = 1
val texto = midia match {
  case FOTOS => "processando fotos"
  case VIDEOS => "processando videos"
  case _ => "processando qualquer outra coisa"
}

println(texto)
```

Ou seja, seja o que for que o *pattern matching* encontrar, será armazenado na variável `texto`. De forma similar a métodos e funções, a última linha de um *case* será o valor retornado. E a inferência de tipos também funciona aqui, portanto, o tipo da variável `texto` neste caso será `String`, já que todos os *case* s retornam `String` s.

Se esse não fosse o caso, Scala procuraria um tipo pai comum entre os tipos dos *case* s. No exemplo a seguir, o tipo do resultado será `Any`, pois ele é o único tipo comum entre

`String` e `Int`. Vamos investigar a hierarquia de classes do Scala mais a fundo no próximo capítulo.

```
val midia = 1
val resultado = midia match {
  case FOTOS => "processando fotos"
  case VIDEOS => "processando videos"
  case _ => -1
}

println(resultado)
```

Agora, em vez de usar constantes para verificar que elementos temos em mãos, vamos usar objetos de verdade. Refatorando nosso exemplo anterior para usar os `case object`s criados anteriormente teríamos o seguinte:

```
val midia = Fotos
val texto = midia match {
  case Fotos => "processando fotos"
  case Videos => "processando videos"
  case _ => "processando qualquer outra coisa"
}

println(texto)
```

Em um primeiro momento, não parece mudar muita coisa. E teríamos de nos preocupar em transformar o valor lido do usuário em um objeto — ou seja, mapear o `Int` para o `case object` equivalente.

Porém, se a mídia for atributo de alguma classe, esse atributo pode ser de um tipo mais específico, que faça sentido para a regra de negócio, e não um valor sintético feito para agradar a linguagem de programação. Mais do que isso: se usarmos *case classes* em vez de *case objects*, podemos inclusive analisar os atributos dessa classe para determinar se encontramos o que queremos ou não.

Por exemplo, o código a seguir armazena `true` na variável `teste` se uma determinada foto for do `jcranky`, e `false` caso

contrário:

```
val teste = foto match {  
  case Foto(_, "jcranky", _, _) => true  
  case _ => false  
}
```

Isto é, para um determinado objeto da classe `Foto`, queremos que ele seja `true` apenas se o seu atributo `owner` for igual a `jcranky`. O valor dos demais atributos não importa. Para que isso funcione, precisamos fazer uma pequena alteração na declaração da nossa classe `Foto`:

```
case class Foto(id: Int, owner: String, title: String, farm: Int)
```

Agora a classe `Foto` é uma *case class*, logo, amigável a *pattern matching*. O mais interessante é que isso não é uma característica especial da linguagem. *Case classes* são apenas açúcar sintático para o que realmente precisamos fazer para o código anterior funcionar: criar os métodos `apply` e `unapply`.

Falaremos mais desses métodos a seguir. O ponto agora é que, além de `equals`, `hashCode` e outros, os métodos `apply` e `unapply` também são gerados para nossas *case classes*.

O que vimos é a base do funcionamento de *pattern matching*, mas existe muito mais, e vale a pena o leitor investigar mais a fundo. Por exemplo, podemos usar condições de guarda para adicionar novas condições para um `case` ser válido. Vamos expandir o exemplo anterior:

```
val teste = foto match {  
  case Foto(_, "jcranky", _, farm) if farm == 7 => true  
  case _ => false  
}
```

Aqui estamos verificando também se a `farm` é a de número 7, e só então retornamos `true`. Poderíamos ter feito da mesma forma que fizemos com o `owner`, mas em alguns casos a condição pode

ser mais complexa e usar guardas fará mais sentido.

Fora isso, repare também que não usamos parênteses no `if`. Diferente de `if`s convencionais, os parênteses nos `if`s das guardas são opcionais.

Em alguns exemplos, nós usamos o `_` (underline) para indicar que um determinado `case` pode aceitar qualquer coisa. Uma outra forma interessante de se fazer a mesma coisa seria dando um nome a esse *qualquer coisa*. Assim, poderíamos, por exemplo, imprimir o que recebemos. Vamos fazer isso no exemplo a seguir:

```
foto.copy(owner = "vic") match {  
  case Foto(_, "jcranky", _, _) => println("foto do jcranky")  
  case f => println(s"Uma outra foto: $f")  
}
```

Esse último recurso é interessante, mas precisamos tomar um cuidado especial com ele. A primeira letra do identificador em questão *deve* ser escrita em letra minúscula. Se não fizermos isso, corremos o risco de adicionar um bug no nosso código que será muito difícil de ser encontrado depois. O exemplo a seguir, com letra maiúscula, não compila:

```
foto.copy(owner = "vic") match {  
  case Foto(_, "jcranky", _, _) => println("foto do jcranky")  
  case F => println(s"Uma outra foto: $F")  
}
```

E o erro que receberemos será algo como:

```
PatternMatching.scala:61: not found: value F
```

Ou seja, o compilador Scala tentou encontrar um valor `F` para verificar contra a variável do *pattern matching*, mas esse valor não existe. Isso porque, quando o identificador começa com uma letra maiúscula, em vez de declarar uma variável, ele procura por uma com aquele nome. Podemos fazer o código compilar da seguinte forma:

```
val F = foto.copy(id = 9999)
foto match {
  case Foto(_, "jcranky", _, _) => println("foto do jcranky")
  case F => println(s"Uma outra foto: $F")
}
```

Agora o código compila, porém teremos um erro em tempo de execução:

```
scala.MatchError: Foto(1,vic,Scala Rulez,1) (of class Foto)
```

Isso acontece pois, quando o *pattern matching* encontrou o `F`, ele tentou comparar com a `foto`, e a comparação falhou. E em seguida, não havia mais nenhum `case` para tentar bater com a `foto`.

No exemplo, o problema é até fácil de entender, pois o `F` é declarado logo acima do *pattern matching*. Mas no dia a dia, o que pode acabar acontecendo é uma variável dessas estar disponível em qualquer outro lugar no escopo do *pattern matching* — o que torna bem difícil encontrar esse problema.

Portanto, lembre-se dessa regra: `cases` que capturam qualquer coisa em uma variável devem sempre ter essa variável com um nome iniciado com letra minúscula.

4.4 MÉTODO UNAPPLY E PATTERN MATCHING COM QUALQUER CLASSE

Tornar a classe `Foto` uma `case class` é a maneira mais simples de suportar o uso de *pattern matching* com ela, além de trazer outros benefícios. Entretanto, não é a única. Na verdade, uma `case class` nada mais é do que um açúcar sintático que gera diversos métodos utilitários, alguns dos quais já mencionamos.

Além dos elementos que mencionamos anteriormente, `case classes` ganham mais dois métodos muito importantes e que são os

responsáveis por permitir o uso de *pattern matching* da forma como fizemos: `apply` e `unapply`. Esses métodos podem ser entendidos como *construtores* e *extratores*: o primeiro é normalmente utilizado para criar objetos, como um pequeno *factory method*; e o último é focado em extrair os atributos de uma classe, ou em outras palavras, *desconstruí-la*.

FACTORY METHOD

O padrão *Factory Method* foi documentado no famoso livro *Design Patterns: Elements of Reusable Object-Oriented Software*, por um grupo de autores conhecido hoje como GoF (*Gang of Four*). Apesar de ser focado em programação orientada a objetos, possui diversos padrões interessantes mesmo no mundo da programação funcional, entre eles o próprio *Factory Method*.

De maneira bem resumida, o que esse padrão define é que podemos usar métodos auxiliares para a criação de objetos de uma determinada classe e, dessa forma, não precisamos saber o tipo exato do objeto sendo criado. Se esse padrão for algo novo para o leitor, vale a pena pesquisar um pouco a respeito. Embora não seja conhecimento obrigatório, tal conhecimento ajudará a entender melhor o que está por trás do método `apply` em muitos casos.

Para definir estes métodos, vamos precisar criar um *companion object* para a foto. Vamos então redefinir nossa classe `Foto` e criar um *companion* para ela:

```
class Foto(id: Int, owner: String, title: String, farm: Int)
object Foto
```

Tendo declarado esses elementos, se tentarmos executar o último *pattern matching* da seção anterior, teremos o seguinte erro:

```
error: object Foto is not a case class constructor,
                                nor does it have an unapply/unapplySeq method
    case Foto(_, "jcranky", _, _) => true
```

A mensagem é bem clara: nossa classe não é uma `case class` e não tem um método `unapply`. Esse método precisa ser criado dentro do `object Foto`, pois ele não é executado em nenhuma instância em especial. Seu papel é trabalhar em conjunto com instâncias de fotos já existentes, como um método auxiliar.

COMPANION OBJECTS NO REPL

Caso queira criar um `companion object` no REPL, será necessário utilizar o `:paste`. Este comando permite digitar várias linhas de uma vez, e fazer com que o REPL avalie e execute todas elas juntas. Isso porque, se declararmos a classe e o `object` separados, o REPL vai interpretá-los como dois elementos completamente independentes, mesmo se tiverem o mesmo nome, ou seja, a relação de *Companion Object* não vai existir.

Vamos agora implementar o `unapply`:

```
class Foto(val id: Int, val owner: String, val title: String, val farm: Int)
object Foto {
  def unapply(foto: Foto): Option[(Int, String, String, Int)] =
    Some((foto.id, foto.owner, foto.title, foto.farm))
}
```

A primeira coisa que tivemos de fazer foi transformar os parâmetros da classe `Foto` em atributos, para que o *companion*

object possa acessá-los. A segunda coisa que temos de fazer é criar o próprio `unapply`, seguindo duas regras: receber um objeto do tipo que vamos extrair, e retornar um `Option` com os elementos extraídos. Repare que agrupamos os elementos retornados em uma tupla.

O retorno é um `Option` para que possamos não retornar nada, ou seja, podemos dizer que o *pattern matching* não casa em determinados cenários. Isso fará com que o *pattern matching* falhe para aquele elemento (a foto, no nosso exemplo) em questão. Para não retornar nada, devemos retornar `None` em vez de um `Some` com o valor extraído.

Podemos retornar qualquer quantidade de elementos, mas essa quantidade retornada é o que deve ser usada no *pattern matching*, independente do construtor que tenhamos declarado na classe. Poderíamos, por exemplo, declarar uma classe `Nome` com um atributo como parâmetro para o *nome completo*, e criar um extrator que nos devolverá *nome* e *sobrenome* separados.

Para o leitor curioso, fica como exercício investigar o código-fonte da classe `scala.util.matching.Regex`. Ele permite utilizar *pattern matching* para extrair padrões de expressões regulares.

4.5 MÉTODO DE FÁBRICA APPLY

Relacionado de forma indireta com *pattern matching*, por ser o oposto do `unapply`, o método `apply` é muito útil e muito usado. A facilidade de construir objetos sem precisar do operador `new` tem pelo menos dois grandes usos: facilitar a criação de *DSLs* internas; e esconder a complexidade da construção de alguns tipos de objetos. Este último está diretamente ligado ao *design pattern Factory Method*, mencionado anteriormente. E em muitos casos, usamos o `apply` por pura conveniência.

Como mencionamos, o `apply` é o oposto do `unapply` : em vez de extrair as partes ou atributos de um objeto, ele constrói um objeto novo a partir dos parâmetros especificados. No exemplo da `Foto` , caso queiramos criar novas fotos sem utilizar o operador `new` , podemos criar o seguinte método `apply` :

```
class Foto(val id: Int, val owner: String, val title: String, val
farm: Int)
object Foto {
  def apply(id: Int, owner: String, title: String, farm: Int) =
    new Foto(id, owner, title, farm)
}
```

Novamente precisamos criar o método dentro do `object` , que precisa ser um *companion object*, da mesma forma que discutimos na seção anterior. Feito isso, podemos criar objetos da classe `Foto` da seguinte forma:

```
val foto = Foto(1, "jcranky", "foto do jcranky", 7)
```

Como o leitor deve ter percebido pelos dois últimos exemplos, na prática o operador `new` não desaparece, ele apenas muda de lugar. Porém essa conveniência é muito poderosa quando, por exemplo, a lógica de construção do objeto em questão é mais complexa do que no exemplo anterior.

Existe ainda mais um ponto muito importante a ser entendido sobre esse método. O código a seguir é exatamente igual ao exemplo anterior:

```
val foto = Foto.apply(1, "jcranky", "foto do jcranky", 7)
```

Ou seja, o método `apply` tem uma característica especial: ele pode ser invocado sem ter seu nome especificado. Sempre que temos um método chamado `apply` , podemos invocá-lo simplesmente colocando a lista de parâmetros após o nome do objeto, entre parênteses.

Isso nos leva a um uso bastante comum de `apply` nas próprias

APIs da linguagem Scala. Para entender esse uso, vamos primeiro criar uma lista de fotos, considerando que os objetos `foto1` , `foto2` e `foto3` já existem:

```
val fotos = List(foto1, foto2, foto3)
```

Neste caso, o *companion object* da classe `List` possui um método `apply` , que recebe um *varargs* de elementos. A forma exata de se criar a lista fica completamente encapsulada neste método. Na verdade, não sabemos nem mesmo qual é a classe exata usada para criar o objeto.

Em seguida, para acessar os elementos dessa lista, podemos usar o índice do elemento da seguinte forma:

```
val fotoUm = fotos(0)
```

À primeira vista, esse código parece simplesmente ser uma versão diferente da sintaxe com colchetes (`[e]`) que muitas linguagens usam para acessar elementos de *arrays*. Porém, na verdade, o que está acontecendo é novamente o uso do método `apply` .

A classe `List` (dessa vez estamos falando da *classe*, não do *object*) possui um método `apply` que é invocado e retorna o elemento da lista com o índice especificado. Portanto, o código a seguir tem o mesmo resultado:

```
val fotoUm = fotos.apply(0)
```

Vamos agora partir para o próximo capítulo e conhecer a hierarquia básica de classes da linguagem Scala!

HIERARQUIA DAS CLASSES BÁSICAS DA LINGUAGEM

Neste capítulo, vamos explorar algumas classes importantes da API do Scala. São classes que, direta ou indiretamente, usamos o tempo todo quando estamos escrevendo código na linguagem Scala e, portanto, todo desenvolvedor deve conhecer.

5.1 OPTION, SOME E NONE

Frequentemente, quando estamos escrevendo código em Java ou outra linguagem similar, enfrentamos a famigerada exceção `NullPointerException` — ou erro equivalente, quando estamos acessando alguma variável, provavelmente recebida de um contexto externo ao qual estamos trabalhando no momento.

Isso pode acontecer por vários motivos, ou porque declaramos variáveis e as inicializamos com `null` e cometemos algum erro no código de inicialização, ou porque estamos recebendo um parâmetro de uma fonte não confiável, ou simplesmente nos esquecemos de inicializar a variável. Ou então quando vamos invocar aquele método com vários parâmetros para os quais ainda não temos todos os valores, passamos `null`, e esquecemos de corrigir isso depois.

Note que parte dos problemas seria resolvida simplesmente evitando usar variáveis e focar em constantes. Mas no caso de

parâmetros recebidos de contextos externos, por exemplo, isso fica bem complicado — para não dizer inevitável.

Os casos anteriores são apenas exemplos de más práticas das quais às vezes é difícil fugir, mas a raiz do problema é a simples existência do `null`. Se escrevermos código como se o `null` não existisse, praticamente eliminamos erros desse tipo, e ainda escrevemos um código mais correto e fácil de entender, sem "atalhos". E é isso que tentamos fazer em Scala.

Na verdade, o `null` existe em Scala apenas para compatibilidade com a linguagem Java. Essa compatibilidade é um dos pontos fortes do Scala e, provavelmente, não faria sentido complicar essa funcionalidade em nome do purismo.

Porém, se estamos apenas escrevendo código Scala, o ideal é esquecermos completamente que o tipo `null` existe. Para que isso seja possível, Scala oferece três elementos que vão trabalhar em conjunto: `Option`, `Some` e `None`.

`Option` é a classe base de `Some` e `None` e indica, como diz o nome, um elemento *opcional*. Esse é um dos segredos dessas classes: elas tornam o fato de um determinado elemento ou parâmetro ser opcional *explícito* em vez de algo que pode acontecer sem percebermos. Vamos revisar nosso método de busca de fotos, agora usando `Option`:

```
def buscaFotos(tag: Option[String]) = ???
```

Veja como ficou óbvio: o parâmetro `tag` do método é opcional. Isso também quer dizer que eventuais outros parâmetros não `Option` são requisitos obrigatórios. Logo, passar `null` para qualquer um deles é uma péssima ideia.

Para invocar o método, temos duas opções:

```
buscaFotos(Some("scala"))
```

Neste caso, estamos especificando quais tags queremos usar na busca. Ou então podemos dizer que não queremos especificar nenhuma tag:

```
buscaFotos(None)
```

A definição do parâmetro ocorre sempre de forma *explícita*, sempre indicando exatamente a nossa intenção.

Olhando a implementação dessas classes, veremos que `None` é um `object`. Não existe razão para termos mais do que um único `None`, já que ele simplesmente indica a ausência de valor. Já `Some` é uma classe que encapsula o valor com o qual vamos trabalhar.

Pense nessa tríade como uma pequena coleção, que pode ter apenas exatamente *0* ou *1* elemento. Esse pensamento é importante, pois a maioria das operações que podemos fazer com coleções (que veremos em outros capítulos) também pode ser aplicada a `Options`.

Temos muitas formas para acessar o conteúdo de um `Option`, em boa parte por causa da equivalência com coleções mencionadas anteriormente. Vamos ver apenas duas por enquanto; as demais podemos inferir do que vamos ver em coleções.

```
val tagBusca = tag.getOrElse("sem tag")
```

Neste caso, estamos indicando explicitamente um valor que queremos usar caso o usuário do método não passe nada, ou seja, passe `None` para o método. Também podemos obter o valor diretamente e deixar um erro acontecer, ou perguntar ao parâmetro se ele existe:

```
val tagBusca = tag.get
val tagExiste = tag.isDefined
```

O método `get` lança uma `NoSuchElementException` se a tag for do tipo `None`, e o método `isDefined` retorna `true` se o

parâmetro for `Some`, e `false` se for `None`. O `isDefined` pode ser muito útil dependendo do algoritmo que estamos implementando.

Já o `get` é um método que devemos evitar usar sempre que possível, pois corremos o risco de voltar a ter os mesmos problemas que estamos tentando evitar. Ou seja, podemos estar apenas trocando uma `NullPointerException` por uma `NoSuchElementException`, sendo que o que queremos é uma código mais seguro e robusto.

5.2 ANY-O QUE?

Temos três classes que são a base de todas as outras em Scala. Uma é a `Any`, que como diz o nome, é qualquer coisa: é a classe mãe de todas as outras classes, similar à classe `Object` em Java. É aqui, por exemplo, que está definido o método `==` (sim, `==` é um método). Em Scala, `==` compara os objetos de verdade, e não referências, e é aqui que isso está definido.

`Any` tem dois filhos muito importantes: `AnyVal` e `AnyRef`. `AnyVal` é a base de todos os valores "primitivos" — entre aspas, porque Scala não possui tipos primitivos de verdade, já que tudo é definido como classes e objetos. Procure por `Int` ou `Double` no *Scaladoc*. Você verá a definição de duas classes, não tipos de variáveis.

Isso pode parecer ruim em termos de performance, pois seria como adicionar uma camada extra de objetos em todo o manuseio de primitivos — em Java isso ocorreria com boxing e unboxing de `int` para `Integer` e vice-versa, por exemplo. Mas o que acontece na prática é que essas classes somem depois da compilação, ou seja, o compilador Scala otimiza esses elementos. Vamos ver um exemplo:

```
class Foto(id: String, owner: String, server: Int, title: String)
```

É uma classe simples, do tipo que já vimos antes. Vamos compilar e usar o `javap` para analisar como essa classe é compilada:

```
scalac Foto.scala
javap Foto.class
```

Vejamos o resultado disso:

```
Compiled from "Foto.scala"
public class Foto {
    public Foto(java.lang.String, java.lang.String, int, java.lang.String);
}
```

Veja que o `Int` sumiu e, no seu lugar, está o `int` — tipo primitivo para valores inteiros suportado pela JVM. Isso vai acontecer para todos os outros tipos "primitivos" em Scala: `Long`, `Float`, `Double` etc.

Até a versão 2.9 do Scala, não era possível criar novos filhos dessa classe: a `AnyVal` era uma classe especial, que não podíamos estender. A partir do Scala 2.10, isso mudou. Foi introduzido o conceito de *Value Classes*, que permitem criarmos classes que, em casos especiais, podem também ser otimizadas e removidas durante o processo de compilação. Vamos falar um pouco mais sobre *Value Classes* no final do capítulo.

Por fim, temos o `AnyRef`. Essa classe sim é o verdadeiro equivalente do Scala para o `Object` do Java. É a classe base para todas as classes que criarmos estender exceto as *Value Classes*, como já mencionamos.

Não há nada de especial para se conhecer sobre essa classe, mas é importante saber que ela existe, para diferenciar o que é filho de `AnyVal` — valores que serão otimizados e potencialmente removidos em tempo de compilação — e o que é filho de `AnyRef`

— todos as demais classes e objetos, ou seja, a grande maioria dos elementos que usamos no dia a dia.

5.3 NULL, NOTHING, UNIT E ???

Vamos agora falar de quatro elementos especiais e muito importantes, começando pelo `Null`. A classe `Null` é filha de todas as outras classes, e isso inclui as classes que nós mesmos criamos.

Esse é um dos motivos pelos quais essa classe é especial: Scala não suporta herança múltipla, mas suporta algo similar por meio de *traits* (veja o capítulo 10. *Classes abstratas e traits*). Mas mesmo assim, a classe `Null` herda de todas as classes existentes.

Mencionamos antes que tudo em Scala é tratado como classes e objetos, e esse tratamento especial do `Null` é necessário para que isso seja possível. Outras coisas especiais da classe `Null` é que não podemos criar objetos a partir dela, e ela tem apenas uma instância: o `null`.

Com isso em mente, deve ser possível entender como o código a seguir funciona e compila normalmente:

```
var foto: Foto = null
```

O `null` é um objeto da classe `Null`. Como `Null` é filha de `Foto`, a atribuição anterior é válida: podemos atribuir um objeto de uma classe filha a uma referência de uma classe mãe.

A próxima classe especial é a `Nothing`. Ela é semelhante à classe `Null` em todos os aspectos citados, com uma diferença: não existe nenhuma instância desta classe. Essa classe é muito útil quando usada com tipos genéricos ou outras classes tipadas (exploraremos isso com mais detalhes no decorrer do livro).

Por exemplo, podemos criar uma lista vazia e a atribuir a uma referência de lista de `String` s:

```
val listaVazia = List[Nothing]()  
var lista: List[String] = listaVazia
```

Na primeira linha do exemplo anterior, estamos criando uma lista de `Nothing` , ou seja, uma *lista de nada*. Essa lista pode ser atribuída a qualquer referência de listas, de qualquer tipo — como fazemos na segunda linha do exemplo.

Como atribuir uma coleção vazia a uma referência de outro tipo de coleção é algo comum, o `object List` oferece um método utilitário que facilita esse cenário, chamado `empty` . O exemplo anterior poderia ser reescrito da seguinte forma:

```
var lista: List[String] = List.empty
```

A hierarquia de classes nesse caso é um pouco mais complicada, porém o resultado é exatamente o mesmo. Já o `Unit` , nosso terceiro elemento especial, é filho de `AnyVal` e significa um valor vazio. Usamos quando queremos representar que não temos nenhum valor, como faríamos com a palavra-chave `void` em Java.

Para manter a característica que mencionamos antes de tudo ser tratado como classes e objetos, Scala usa a classe `Unit` e seu único filho `()` para representar esse retorno. E assim como com `Null` e `Nothing` , também não podemos criar nenhum objeto dessa classe diretamente. Vejamos o seguinte teste no REPL:

```
scala> val x = ()  
x: Unit = ()
```

Agora vejamos a assinatura do método `println` do objeto `Predef` :

```
def println(): Unit
```

Ou seja, o método vai imprimir algo na tela, e não vai retornar

nenhum valor como resultado, exatamente como o `System.out.println` do Java, que retorna `void`.

Para finalizar, vamos falar do `???`. Diferente dos outros três elementos, `???` não é uma classe, e sim um método definido no objeto `Predef`. Vejamos seu código completo:

```
def ??? : Nothing = throw new NotImplementedError
```

Esse método simplesmente lança uma exceção, dizendo que o elemento atual ainda não foi implementado. O tipo de retorno dele é `Nothing`, para que seja compatível com qualquer um dos nossos métodos. Isso porque, como mencionamos anteriormente, `Nothing` é filha de todas as classes, inclusive as nossas. Agora deve ficar bem claro por que antes definimos nosso método `buscaFotos` dessa forma:

```
def buscaFotos(tag: Option[String]): Seq[Foto] = ???
```

Ainda não temos o corpo do método, mas gostaríamos de poder compilar o código mesmo assim. E mais: se alguém tentar usar esse método antes de adicionarmos uma implementação real, o `NotImplementedError` será lançado, garantindo que código incompleto não seja usado.

BOA PRÁTICA NO USO DO ???

O `???` deve ser usado apenas para fins de desenvolvimento, ou seja, quando ainda estamos trabalhando em alguma implementação. Nenhum código finalizado deve usar o `???`, pois outro desenvolvedor lendo o código claramente pensará se tratar de um código não finalizado.

5.4 EXCEPTIONS

Como toda linguagem que se preze, Scala também possui suporte a tratamento de erros. Para quem está acostumado com a linguagem Java, o mecanismo de exceções é muito parecido: em algum momento, uma exceção pode ser lançada, e isso causa a parada da execução do código que estava sendo executado.

Se invocarmos um método que lança uma exceção e não fizermos nada, nosso código também para, e a exceção é repassada para o código que chamou nosso método, até que não haja para quem repassar a exceção. Isso fará com que o programa pare.

Vamos reescrever o método `buscaFotos` com o mesmo comportamento de antes, mas agora sem o `???` e lançando uma exceção explicitamente:

```
def buscaFotos(tag: Option[String]) = throw new NotImplementedError  
()
```

Usamos a palavra chave `throw` para indicar que queremos lançar o erro. Existem algumas diferenças entre erros e exceções, mas vamos usar os dois termos como sinônimos por enquanto, pois a forma de tratá-los não muda. Falaremos um pouco mais sobre as boas práticas relacionadas a erros e exceções em seguida.

Se simplesmente invocarmos o método anterior sem fazer nada em especial, nosso programa vai parar a execução e nos mostrar o erro lançado. Para evitar isso, devemos capturar a exceção:

```
try {  
  buscaFotos(Some("scala"))  
} catch {  
  case t: Throwable => println("exceção ao tentar buscar fotos")  
}
```

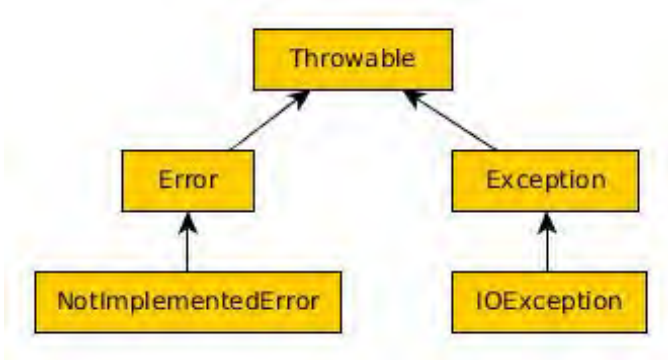
Esse mecanismo é muito similar ao que acontece em Java também, mas a sintaxe é mais flexível. O bloco `catch` funciona

como blocos de *pattern matching* (estudaremos mais sobre isso em outro capítulo). Portanto, temos muita flexibilidade para capturar e analisar os erros. Podemos, por exemplo:

```
try {  
  buscaFotos(Some("scala"))  
} catch {  
  case e: IOException => println("exceção de IO ao tentar buscar f  
otos")  
  case e: Exception => println("exceção indefinida ao tentar busca  
r fotos")  
  case _ => println("alguma outra exceção")  
}
```

Agora estamos capturando uma exceção mais específica. E se a exceção não for `IOException`, o segundo `case` vai capturá-la. E se o segundo `case` também não for suficiente, o terceiro será, pois ele captura qualquer coisa.

Nos exemplos anteriores, vimos quatro erros e exceções diferentes: `Throwable`, `Exception`, `IOException` e `NotImplementedError`. Para entender um pouco melhor como esses elementos estão relacionados, vamos ver a hierarquia com as classes em questão:



A hierarquia nessa figura é uma velha conhecida para os programadores Java, e Scala a utiliza diretamente — neste caso,

apenas adicionando o erro `NotImplementedError` . Com essa hierarquia em mente, é importante saber que, de forma geral e assim como seria o caso em Java, não é uma boa prática capturarmos `Throwable` diretamente, e nem mesmo `Error` .

Isso porque, na grande maioria das vezes, `Error` s representam problemas dos quais não podemos nos recuperar de forma correta, como `OutOfMemoryError` . Se o leitor executou o exemplo anterior, pode ter notado que o compilador emitiu o seguinte aviso, por estarmos capturando todos os erros e exceções:

```
Exceptions.scala:21: This catches all Throwables.  
  If this is really intended, use `case _ : Throwable` to clear thi  
s warning.  
[warn]     case _ => println("alguma outra exceção")  
[warn]         ^  
[warn] one warning found
```

Entretanto, alguns erros são capturáveis, como o `NotImplementedError` que vimos antes. Para facilitar nossa vida, nesses casos o Scala oferece uma classe auxiliar chamada `NonFatal` . Podemos então usar essa classe para capturar qualquer erro que seja seguro capturar.

Vamos então reescrever nosso último exemplo para que ele não capture erros que não deveria, mas ainda assim capture o nosso `NotImplementedError` :

```
try {  
  buscaFotos(Some("scala"))  
} catch {  
  case e: IOException => println("exceção de IO ao tentar buscar f  
otos")  
  case e: Exception => println("exceção indefinida ao tentar busca  
r fotos")  
  case NonFatal(t) => println("alguma outra exceção segura")  
}
```

NonFatal

O `NonFatal` usa o método `unapply` que vimos em um capítulo anterior para implementar a captura da exceção. Como o bloco `catch` utiliza *pattern matching* para determinar se determinada exceção deve ou não ser tratada, a função `NonFatal.unapply` pode verificar se a exceção em questão é realmente fatal, e simplesmente retornar `None` caso seja; ou `Some` com a exceção não fatal encontrada.

Falamos um pouco que o mecanismo de exceção do Scala é bem similar ao de Java. Mas existe uma diferença importante: em Scala, não existem exceções checadas. Ou seja, não é possível obrigar o tratamento de uma exceção, de forma alguma. Por um lado isso, é bom, pois não temos o excesso de blocos `try / catch` que acontece em muitos códigos Java, mesmo quando tratar tal exceção não faz sentido na camada na qual estamos trabalhando.

Por outro lado, isso pode ser um pouco ruim: precisamos sempre pensar no que pode dar errado nos nossos programas e prestar atenção nas exceções que podem ser lançadas pelo nosso código. É possível esquecermos de tratar algum tipo de exceção.

O ponto negativo mencionado é relativo: pensar nos cenários de erros e tratá-los adequadamente já é nosso trabalho como desenvolvedor, de qualquer forma. Não nos obrigar a tratar exceções é apenas uma forma da linguagem Scala nos permitir escolher exatamente onde queremos fazer esse tratamento.

5.5 VALUE CLASSES

Se olharmos o *scaladoc* da classe `Int`, veremos muitos métodos

interessantes. Além das operações óbvias que podemos fazer com números inteiros, temos outros métodos úteis como `isValidChar` ou `toHexString`.

Veremos com mais detalhes como esses métodos podem ser utilizados diretamente como se fosse parte da classe `Int` no capítulo 12. *Implicits*. O interessante para este momento é entender que *Value Classes* podem muitas vezes ser otimizados pelo compilador e sumir em tempo de execução.

NOVIDADE

Value Classes foram introduzidas na versão 2.10 do Scala. Em versões anteriores, estender `AnyVal` como faremos a seguir é ilegal e não compila.

Para entender melhor, vamos ver um exemplo. No código a seguir, criamos uma *Value Class* que nos permitirá dobrar o valor de um número inteiro:

```
class IntDobrado(val x: Int) extends AnyVal {  
  def dobrado: Int = x * 2  
}
```

Ela consiste apenas de um construtor que recebe um número inteiro e um método. Repare que definimos o valor inteiro como um `val`. Isso é parte dos requisitos para se criar uma *Value Class*. Feito isso, podemos utilizar essa classe da seguinte forma:

```
val dobro = new IntDobrado(10).dobrado  
println(dobro)
```

Até aqui, nada demais. O interessante de *Value Classes* é que a criação do objeto vista no exemplo anterior some depois que o código é compilado. O código anterior é reescrito pelo compilador

para algo como o seguinte:

```
val dobro = IntDobrado.dobrado(10)
println(dobro)
```

Ou seja, o método se tornou um método utilitário, em uma classe estática, e o compilador cuida disso, enquanto nos focamos em escrever um código mais correto. Essa alteração é bem pequena, e gera um ganho de performance minúsculo, mas que pode ser significativa em situações que requerem uma grande quantidade de alocação de objetos.

Se utilizarmos o `javap` para analisar o *bytecode* gerado pelo `scalac` para o exemplo anterior, veremos claramente a alteração da assinatura do método, que agora recebe um inteiro como parâmetro, e da sua localização em um contexto estático:

```
public final class IntDobrado {
  private final int x;
  public static boolean equals$extension(int, java.lang.Object);
  public static int hashCode$extension(int);
  public static int dobrado$extension(int);
  public int x();
  public int dobrado();
  public int hashCode();
  public boolean equals(java.lang.Object);
  public com.jcranky.scalando.cap05.IntDobrado(int);
}
```

Repare que os métodos `equals` e `hashCode` receberam tratamento semelhante. Veja também que os métodos novos receberam o sufixo `$extension`, e que os métodos originais ainda existem.

Isso é importante porque, em alguns casos, o compilador pode decidir que a otimização não é possível, portanto, precisa alocar um objeto real. O *bytecode* anterior garante que as duas opções estão disponíveis para o `scalac`.

A cereja do bolo, como mencionamos antes, veremos no

capítulo 12. *Implicits*. Lá, veremos como transformar o código anterior no código a seguir, e torná-lo ainda mais enxuto e, ainda sim, bastante legível:

```
10.dobrado
```

COLEÇÕES

A hierarquia de coleções do Scala pode ser bem complicada de se entender em um primeiro momento, com algumas arestas pontiagudas: coisas que somente quem quer criar novos tipos de coleções precisa realmente entender. E essa é a boa notícia: não precisamos conhecer todos os detalhes do funcionamento das coleções da linguagem para aproveitar o ganho de produtividade que temos com seu uso!

Vamos a seguir ver os principais elementos da API de coleções do Scala, focando no que é indispensável saber para trabalharmos de forma efetiva.

6.1 ELEMENTO BÁSICO: TRAVERSABLELIKE

No topo da hierarquia da API, temos a `trait TraversableLike`. As *traits* são uma espécie de interface, mas veremos bem mais sobre elas em um capítulo específico. Essa `trait` define operações disponíveis em todas as coleções, ou seja, tudo o que está lá pode ser usado em `Map`s, `Set`s, `List`s etc. — e até mesmo em `Option`s. São operações como `fold` e `filter`, que veremos no próximo capítulo.

Essa `trait` tem também muitas outras `trait`s como base, mas estas entram na categoria de elementos que mencionamos antes: não precisamos entendê-las completamente para usar a API. Com a `TraversableLike` em mente, já podemos fazer algumas

coisas bem legais.

Para um primeiro exemplo, vamos definir uma função que, dada uma foto, imprime seus dados na tela:

```
def imprimeFoto(f: Foto) = println(s"Foto: id = ${f.id} owner = ${f.owner}")
```

Agora, dada uma coleção qualquer de fotos, podemos facilmente imprimir todos os seus elementos na tela:

```
val fotos = // coleção qualquer de fotos
fotos.foreach(imprimeFoto)
```

O método `foreach` é mais uma das funções definidas em `TraversableLike` : ele recebe uma função como parâmetro e a executa para cada elemento da coleção. Vejamos a assinatura do `foreach` para isso fazer mais sentido:

```
abstract def foreach(f: (A) => Unit): Unit
```

Primeiro, essa é uma função abstrata: as implementações específicas vão definir exatamente como a coleção será percorrida. O importante é que ela será percorrida. Segundo, essa função tem como tipo de retorno `Unit` , ou seja, nada. A função que passarmos como parâmetro será executada, mas nada será devolvido como resultado.

Agora, a parte mais interessante: o parâmetro `f` . Para aceitar a nossa função, o `foreach` está dizendo que ela deverá receber um parâmetro `A` , em que `A` é o tipo de elemento da coleção (um coleção de fotos no nosso caso), e que a função deve devolver `Unit` .

Podemos dizer então que o `foreach` é uma função que recebe uma outra função de `A` para `Unit` . Lembrando de que essa é uma das características de linguagens funcionais: suportar funções que podem receber outras funções como parâmetro.

Esse é exatamente o caso da função `imprimeFoto` que criamos anteriormente: ela recebe uma foto e não devolve nada como resultado. Seu resultado é na verdade o que chamamos de *side-effect*, ou efeito colateral. Neste caso, o efeito é algo a ser impresso na tela.

Veremos com mais detalhes o lado funcional das coleções no próximo capítulo, mas já podemos entender pelo menos um pouco como as coisas funcionam.

6.2 SETS

Set s são conjuntos matemáticos que possuem basicamente duas características importantes: não permitem elementos duplicados e não garantem a ordenação dos elementos. A maioria das linguagens de programação suporta o conceito de Set s e seguem essas definições, e Scala não é exceção.

Vamos criar um Set simples:

```
val foto1 = new Foto("id1", "jcranky", 1, "uma foto do jcranky")
val foto2 = new Foto("id2", "jcranky", 1, "outra foto do jcranky")
val foto3 = new Foto("id3", "jcranky", 1, "mais uma foto do jcranky")

val fotos = Set(foto1, foto2, foto3, foto1)
```

Repare que tentamos duplicar a `foto1` no Set . Repare também que as demais fotos foram adicionadas ao Set seguindo a ordem natural do campo `id` . Vamos agora usar o que aprendemos na seção anterior para imprimir todas as fotos na tela no código a seguir:

```
fotos.foreach(println)
```

Estamos fazendo a impressão de forma um pouco mais simples dessa vez: estamos passando o `println` diretamente para o `foreach` em vez de declarar uma função separada para isso. O `println` é uma função que recebe qualquer coisa e a imprime na

tela. Essa assinatura é compatível com o que o `foreach` espera.

Vamos agora ver o resultado da execução do código anterior. Tenha em mente que transformamos a classe `Foto` em uma `case class`, para ganhar o `toString` bem definido:

```
Foto(id1,jcranky,1,uma foto do jcranky)
Foto(id2,jcranky,1,outra foto do jcranky)
Foto(id3,jcranky,1,mais uma foto do jcranky)
```

Em primeiro lugar, a foto repetida sumiu. E no nosso caso, a ordenação foi mantida, mas isso tem de ser tratado como um detalhe de implementação e não é garantido que aconteça. Logo, não podemos contar com isso.

Se for necessário, é possível garantir a ordenação: basta utilizar uma implementação de `Set` que adicione essa garantia. `Set` possui um filho chamado `SortedSet`, e todos os `Set`s que herdam dele garantem a ordenação de seus elementos. Para o leitor curioso, fica a recomendação de consultar o *Scaladoc* do `SortedSet` e seu principal filho, `TreeSet`.

Adicionar e remover elementos de um `Set` é extremamente simples e, graças à sintaxe flexível da linguagem Scala, muito legível:

```
val foto4 = new Foto("id4", "jcranky", 1, "ainda mais uma foto do
jcranky")
val novasFotos = fotos + foto4
```

Simplesmente somamos o novo elemento à coleção inicial. O detalhe importante a se prestar atenção aqui é que foi criado um novo `Set`. O original continua exatamente como antes. Se imprimirmos o `Set` `fotos`, o resultado continuará exatamente igual a antes. Se imprimirmos o `Set` `novasFotos`, teremos o seguinte resultado:

```
Foto(id1,jcranky,1,uma foto do jcranky)
Foto(id2,jcranky,1,outra foto do jcranky)
Foto(id3,jcranky,1,mais uma foto do jcranky)
```

```
Foto(id4,jcranky,1,ainda mais uma foto do jcranky)
```

Obedecendo à semântica de um `Set`, adicionar um elemento igual a algum que já existe não tem nenhum efeito prático. Ou seja, no exemplo a seguir, a comparação resultará em `true`:

```
val fotosIguais = fotos + foto1
fotosIguais == fotos
```

Teremos na prática um novo `Set`, mas este será exatamente igual ao original. Tenha em mente que isso não significa que todos os valores serão duplicados em memória: a implementação do `Set` (e das demais coleções) vai reutilizar os objetos que forem possíveis.

Por fim, podemos remover elementos do `Set` de forma similar, agora com o `-` (sinal de menos):

```
val menosFotos = fotos - foto1
menosFotos.foreach(println)
```

E o resultado será algo próximo do seguinte código, com a ordem dos elementos podendo variar:

```
Foto(id2,jcranky,1,outra foto do jcranky)
Foto(id3,jcranky,1,mais uma foto do jcranky)
```

6.3 LISTS

Outro tipo comum de coleção suportada pela grande maioria das linguagens de programação é a lista. A lista é a coleção mais próxima dos `Array`s: coleções ordenadas por índices.

Não há nenhum tipo de restrição em elementos repetidos, e a sua ordenação geralmente é a ordem na qual os elementos foram inseridos. Uma lista simples poderia ser definida e impressa como no seguinte código:

```
val fotos = List(foto1, foto2, foto3, foto1)
fotos.foreach(println)
```

E resultado do `foreach` anterior seria:

```
Foto(id1,jcranky,1,uma foto do jcranky)
Foto(id2,jcranky,1,outra foto do jcranky)
Foto(id3,jcranky,1,mais uma foto do jcranky)
Foto(id1,jcranky,1,uma foto do jcranky)
```

Uma das vantagens da lista é que podemos acessar seus elementos diretamente. O seguinte código acessa e imprime o segundo elemento da lista de fotos:

```
println(fotos(1))
```

Como `Arrays`, a indexação começa em 0 (zero). Por isso utilizamos o 1 (um) para acessar a segunda foto.

Adicionar e remover elementos de listas acontece de forma diferente do que fazemos com `Sets`. Isso é inevitável, já que são tipos diferentes de coleções e devem ser tratadas de forma diferente. Antes de alterar uma lista, vamos ver uma forma alternativa para criar a mesma lista que criamos anteriormente:

```
val fotos = foto1 :: foto2 :: foto3 :: foto1 :: Nil
```

Essa lista é exatamente igual à anterior, mas agora usando sintaxe específica para listas. Na verdade, como `Scala` suporta praticamente qualquer coisa como identificador, os `<::` (dois pontos, duas vezes) nada mais são do que um método.

Temos apenas uma coisa aqui que é regra de sintaxe do `Scala`: quando um método termina com `:` (dois pontos), ele é invocado no elemento à sua direita em vez de ser invocado no elemento à sua esquerda como seria o normal. Ou seja, o código anterior poderia ser reescrito da seguinte forma:

```
val fotos = Nil.::(foto1).::(foto3).::(foto2).::(foto1)
```

Repare que a ordem na qual os elementos foram adicionados foi oposta à usada no exemplo anterior. Ou seja, dessa vez começamos

pelo último elemento.

Mas e o `Nil` ? O `Nil` é um objeto que representa uma lista vazia. No código anterior, então, começamos com a lista vazia e acrescentamos um elemento por vez a esta lista.

Na verdade, estamos criando uma série de pequenas listas encadeadas e invocando o `<::` em cada uma delas. É fácil perceber isso quando olhamos a assinatura do método `<::`, cujo tipo de retorno é definido como uma lista:

```
def <::(x: A): List[A] // <::
```

A declaração simplificada do `Nil` é a seguinte:

```
object Nil extends List[Nothing]
```

O `Nothing` na assinatura garante que o `Nil` possa representar listas vazias de qualquer tipo de elemento.

Como mencionamos, e da mesma forma que com `Sets`, cada elemento acrescentado gera uma nova lista. Isso vale para todas as coleções imutáveis. Vamos discutir isso com calma no final do capítulo.

O resultado da chamada ao método `<::` é uma nova lista com o elemento adicionado, ou seja, já vimos como criar listas e adicionar elementos a elas de uma só vez. Uma forma alternativa para adicionarmos elementos seriam os métodos `:+` e `++`. Vejamos um exemplo a seguir:

```
val fotos2 = fotos :+ foto2
val fotos3 = foto2 ++ fotos
```

O segredo aqui é entender que a nova foto é adicionada ao final da lista no primeiro caso, e ao começo dela no segundo. A leitura do código fica intuitiva depois de entendido esse ponto: veja para que lado os `:` (dois pontos) estão apontando.

Outra forma muito comum e útil de se acessar elementos de listas é usando *pattern matching*. Podemos decompor uma lista diretamente e acessar os seus elementos de forma bem simples. No exemplo a seguir, separamos a cabeça e a cauda de lista de fotos e imprimimos a cabeça:

```
fotos match {  
  case head :: tail => println(head)  // ::  
}
```

Uma outra forma de fazer isso seria usar o `Nil`, como a seguir:

```
fotos match {  
  case head :: Nil => println(head)  // ::  
  case _ =>  
}
```

Nesse caso, o elemento só será impresso se a cauda da lista for vazia. Caso contrário, o *pattern matching* falhará. Por isso usamos o `case _ =>`, para evitar um erro em tempo de execução.

Idealmente, nesses casos, queremos tratar tanto o caso com a cauda vazia quanto o com uma cauda de verdade. Esse tipo de recurso é também bastante útil na implementação de algoritmos recursivos. Podemos, por exemplo, imprimir a cabeça da lista até que todos os elementos sejam impressos:

```
def printaLista(lista: List[Foto]): Unit = {  
  lista match {  
    case head :: Nil =>  
      println(head)          // ::  
  
    case head :: tail =>      // ::  
      println(head)  
      printaLista(tail)  
  
    case Nil =>  
  }  
}  
  
printaLista(fotos)
```

Nesse caso, também precisamos tomar cuidado para os casos extremos, por isso usamos o `case Nil =>` . Assim, evitamos um erro em tempo de execução caso a lista esteja vazia desde o começo.

Repare também que o `head :: Nil` vem antes do `head :: tail` . Isso é importante pois o `tail` também captura listas vazias, logo, o `head :: Nil` nunca seria executado se viesse depois — o compilador emitiria um *warning* nesse caso.

Dependendo do algoritmo, usar somente o `head :: tail` pode fazer mais sentido. Experimente remover completamente o `case head :: Nil =>` para ver o que acontece.

6.4 TUPLAS

Tuplas não são exatamente um tipo de coleção, mas são muito úteis para o agrupamento de objetos. Portanto, são elementos importantes semelhantes o suficiente para serem mencionados aqui.

As tuplas nos ajudam a organizar um pouco o nosso código. Por exemplo, podemos utilizá-las para retornar dois elementos de algum método, sem ter de criar uma nova classe para isso.

Se procurarmos no *Scaladoc*, encontraremos apenas duas classes relativas às tuplas: `Tuple1` e `Tuple2` . Isso porque ela é um elemento tratado especialmente pelo compilador. Antes de mais nada, vejamos um exemplo de uso de tuplas:

```
val dadosFoto = ("jcranky", "reunião dos scaladores")
println(s"owner: ${dadosFoto._1} - title: ${dadosFoto._2}")
```

Esse código ilustra duas coisas: como criar uma tupla, e como acessar seus elementos. O tipo de tupla anterior poderia especialmente ser chamado de par, pois estamos trabalhando com duas (um par de) variáveis de uma vez só. Mas essa seria uma definição incorreta, pois tuplas em Scala não são limitadas a apenas

dois elementos.

E aqui entra o tratamento especial que mencionamos antes: o compilador aceitará tuplas com até 22 variáveis — classes `TupleX`, em que `X` é o número de elementos da tupla, são geradas pelo compilador.

Ainda precisamos chamar atenção a dois pontos no código anterior: os tipos das variáveis, e o índice de acesso. Primeiro, apesar de termos usado duas `String`s, poderíamos ter usado qualquer tipo de elemento na tupla. Se digitarmos a declaração da tupla do código anterior no REPL, ele nos responderá com a seguinte mensagem:

```
dadosFoto: (String, String) = (jcranky, reunião dos scaladores)
```

Ou seja, nossa tupla é especificamente do tipo `(String, String)`. Estamos imaginando uma tupla do owner da foto, com o título dela. Se quisermos uma tupla com o `id` da foto e o título, podemos fazer o seguinte:

```
val dadosFoto2 = (123, "reunião dos scaladores")
println(s"id: ${dadosFoto2._1} - title: ${dadosFoto2._2}")
```

E no REPL o resultado será algo como o seguinte:

```
dadosFoto2: (Int, String) = (123, reunião dos scaladores)
```

E como mencionamos antes, não estamos limitados a dois elementos:

```
val dadosFoto = (123, "jcranky", "reunião dos scaladores")
println(s"id: ${dadosFoto._1} - owner: ${dadosFoto._2} - title:
${dadosFoto._3}")
```

É claro que muitas vezes é mais interessante criar uma classe nova para agrupar as informações que estamos colocando na tupla — e foi o que fizemos no nosso exemplo do Flickr: criamos a classe `Foto`. Mas muitas vezes, queremos apenas retornar dois ou três

valores de um método e não precisamos ou queremos uma classe nova.

Nesse caso, tuplas funcionam muito bem. Só tome cuidado para não exagerar. Apesar de Scala suportar tuplas com até 22 variáveis, uma tupla de tal tamanho seria extremamente difícil de se ler e entender.

O segundo ponto que precisamos discutir é a forma de acesso aos elementos. A forma que vimos até aqui foi usando o índice, como vimos nos exemplos anteriores. Cuidado: esse índice começa com o número 1 (um), e não 0 (zero) como seria com qualquer coleção. Na verdade, esses índices são `vals` na classe gerada para a tupla, e não um método de acesso único como seria com uma coleção ou `Array`.

Essa não é, porém, a única e nem mesmo a melhor forma de acessar elementos de uma tupla, pois é geralmente de difícil leitura. Temos algumas alternativas para resolver esse problema. A primeira delas é extrair os elementos da tupla diretamente em variáveis separadas:

```
val dadosFoto4 = (123, "reunião dos scaladores")
val (id, title) = dadosFoto4

println(s"id: $id")
println(s"title: $title")
```

Ou seja, podemos declarar variáveis usando os parênteses da tupla, e o compilador extrairá os elementos automaticamente. O resultado da segunda linha anterior, no REPL, seria:

```
id: Int = 123
title: String = reunião dos scaladores
```

Temos realmente duas novas variáveis para trabalhar. Poderíamos também reescrever o código anterior de uma forma ligeiramente mais simples:

```
val (id, title) = (123, "reunião dos scaladores")
```

Uma outra forma de acessar elementos de tuplas de maneira legível é usando *pattern matching*. Isso porque, na prática, o exemplo anterior é um atalho para *pattern matching*. Podemos, portanto, também acessar os elementos da tupla `dadosFotos` da seguinte forma:

```
dadosFoto4 match {  
  case (id3, title3) => println(s"id: $id3 - title: $title3")  
}
```

6.5 MAPS

Mapas são conjuntos de pares chave/valor, também conhecidos como *dicionários* em muitas linguagens. São conjuntos muito próximos de qualquer outra coleção, com uma diferença fundamental: os elementos são armazenados e acessados usando chaves.

E é aí que entram os *pares* chave/valor. Vejamos um exemplo simples, no qual vamos colocar os nomes de alguns métodos que o *web service* do Flickr aceita:

```
val services = Map(  
  ("busca", "flickr.photos.search"),  
  ("tamanhos", "flickr.photos.getSizes")  
)
```

Usamos tuplas para representar cada elemento (cada par chave/valor) do mapa. Fora isso, a criação do mapa é exatamente igual à criação de qualquer outro tipo de coleção.

O que não mencionamos antes, quando falamos de tuplas, é que existe uma sintaxe mais legível para representar tuplas de dois elementos — um açúcar sintático. O exemplo anterior poderia ser reescrito da seguinte forma:

```
val services = Map(  

```

```
"busca" -> "flickr.photos.search",  
"tamanhos" -> "flickr.photos.getSizes"  
)
```

Mas isso é apenas uma forma mais bonita de se escrever a mesma coisa, ou como já mencionamos, açúcar sintático. Use a forma que ficar mais legível para o seu código.

Se compararmos os dois mapas anteriores, veremos que ambos são exatamente iguais. O código a seguir imprimirá `true` :

```
println(services == services2)
```

Acessar os elementos do mapa também é muito simples. Vamos ver duas formas comuns para fazer isso:

```
val metodoBusca = services("busca")  
val metodoBuscaOpt = services.get("busca")
```

A primeira forma é útil quando temos certeza de que a chave existe no mapa pois, caso ele não exista, uma exceção do tipo `NoSuchElementException` é lançada. Normalmente, evitamos fazer esse tipo de acesso, pois seria fácil esquecer de tratar a exceção e gerar um potencial bug em tempo de execução.

A segunda forma é bem mais interessante, principalmente quando não sabemos se o elemento existe, o que é comum, e queremos tratar isso adequadamente — como vimos quando falamos de `Option`s. O método `get` retorna um `Option` com o valor referente à chave. Ou seja, será algo como `Some(valor)` , ou `None` , se não existir valor para a chave especificada.

Alternativamente podemos tratar a eventual não existência da chave utilizando um valor padrão:

```
val metodoBusca = services.getOrElse("busca", "método padrão")
```

Vamos agora ver algumas formas para alterar o mapa, lembrando de que estamos usando coleções imutáveis. Ou seja, cada

operação na verdade gera um novo mapa em vez de alterar o existente. Para adicionar um elemento, simplesmente o somamos ao mapa existente:

```
val novosServices = services + ("untagged" -> "flickr.photos.getUntagged")
```

Note que, mesmo usando o açúcar sintático, precisamos dos parênteses para forçar a precedência correta dos operadores. Executar essa operação sem os parênteses fica como um exercício para o leitor.

Similar à adição de elementos, remover também é bastante simples:

```
val menosServices = services - "busca"
```

O mapa novo é igual ao mapa antigo, menos a chave que estamos removendo. Também vamos precisar alterar elementos de vez em quando. Para isso, usamos o método `updated`:

```
val servicesAtualizados = services.updated("busca", "flickr.photos.newSearch")
```

6.6 ARRAYS

Um elemento à parte na API de coleções do Scala são os `Arrays`. Isso por dois motivos principais. O primeiro é que `Arrays`, diferente dos outros tipos de coleções, não são um `TraversableLike`. Isto é, as operações comuns que vimos antes a princípio não existem; pelo menos, não de forma direta.

Em `scala.Predef`, existem algumas conversões implícitas que transformam `Arrays` em outros objetos, que adicionam as operações que esperamos poder executar em qualquer tipo de coleção. Por tanto, mesmo `Arrays` não possuindo tais operações diretamente, ainda podemos contar com elas. Falaremos mais sobre

conversões implícitas em um capítulo específico, mas o leitor curioso pode olhar a classe `WrappedArray` para entender de onde os métodos estão vindo.

Olhando para o código, usar `Array` s é então muito parecido com usar qualquer outro tipo de coleção. Vejamos a seguir um pequeno exemplo:

```
val foto1 = new Foto("id1", "jcranky", 1, "uma foto do jcranky")
val foto2 = new Foto("id2", "jcranky", 1, "outra foto do jcranky")
val foto3 = new Foto("id3", "jcranky", 1, "mais uma foto do jcranky")

val fotos = Array(foto1, foto2, foto3, foto1)
```

Repare que o exemplo é praticamente igual ao que vimos anteriormente, apenas usando `Array` em vez de algum outro tipo de coleção. Temos apenas dois pontos que precisamos manter em mente quando decidirmos usar `Array` s: o primeiro é o fato de que eles são convertidos para arrays nativos da *JVM* quando o código é compilado.

O segundo ponto, em parte consequência do primeiro, é que arrays são mutáveis. Portanto, geralmente acabando sendo mais interessante usar `Array` s apenas para otimizar código que precisam usar alguma coleção de maneira muito intensa.

De qualquer forma, ao acessar elementos de `Array` , utilizamos exatamente a mesma estrutura que usamos para acessar elementos de qualquer outra coleção em vez de usar uma sintaxe especializada, como seria com a linguagem Java. No exemplo a seguir, vamos imprimir o primeiro elemento do array criado anteriormente:

```
println(fotos(0))
```

Ou seja, fazemos uso do método `apply` da classe `Array` , exatamente como qualquer outra coleção. E o mesmo aconteceu quando criamos o array, no qual usamos o método `apply` do

objeto `Array` .

6.7 COLEÇÕES IMUTÁVEIS VERSUS COLEÇÕES MUTÁVEIS

Como mencionamos antes, até aqui trabalhamos somente com coleções imutáveis, ou seja, coleções que não mudam. Quando adicionamos uma foto em um `Set` de fotos, por exemplo, o resultado foi um `Set` novo — o original permanece inalterado. Vamos relembrar do código e comparar as duas coleções:

```
val foto1 = new Foto("id1", "jcranky", 1, "uma foto do jcranky")
val foto2 = new Foto("id2", "jcranky", 1, "outra foto do jcranky")
val foto3 = new Foto("id3", "jcranky", 1, "mais uma foto do jcranky")

val fotos = Set(foto1, foto2, foto3, foto1)

val foto4 = new Foto("id4", "jcranky", 1, "ainda mais uma foto do jcranky")
val novasFotos = fotos + foto4

println(fotos == novasFotos)
```

Não vamos ver as coleções mutáveis, mas se realmente quisermos trabalhar com esse tipo de coleção, temos diversas disponíveis no pacote `scala.collection.mutable` . Por padrão, Scala sempre usará coleções imutáveis, logo, sendo essa a boa prática. Se realmente precisarmos utilizar uma coleção mutável, precisaremos importá-la explicitamente.

Vamos finalizar este capítulo discutindo um pouco sobre por que devemos dar preferência para as versões imutáveis das coleções, que, como mencionado anteriormente, são as usadas por padrão.

Primeiro, a implementação das coleções imutáveis é inteligente o suficiente para evitar desperdícios. Isto é, quando são criadas as novas coleções, tudo o que pode ser reaproveitado da estrutura

interna da coleção original é mantido.

O simples fato de estarmos trabalhando com esse tipo de coleção faz com que muita coisa possa realmente ser reaproveitada, pois a implementação pode contar com a imutabilidade como fato e fazer otimizações de acordo. E esse é exatamente o nosso segundo ponto: coleções (ou qualquer tipo de objeto, na verdade) podem ser reusadas e compartilhadas à vontade.

Tipicamente no mundo Java, quando falamos em compartilhar objetos, precisamos nos lembrar de proteger o acesso a esses objetos. Isso porque, se esse compartilhamento ocorrer em múltiplas threads, podemos ter problemas de concorrência no acesso a eles, como duas threads tentando acessar (e alterar) o mesmo objeto, ao mesmo tempo.

Porém, se tal objeto for imutável, esse problema não existe. Como ele nunca muda, deixá-lo ser acessado por uma, duas ou 100 threads não causa problema algum: não há risco de uma thread alterar uma informação sendo usada por outra.

É claro que não existem programas 100% imutáveis, ou então não teríamos um software resolvendo problema algum. Na prática, o que a imutabilidade traz é uma forma diferente de pensar: usamos objetos que não mudam.

Quando precisamos de algo novo, criamos um novo objeto, com as mudanças necessárias. Isso da mesma forma que sempre fizemos com `String`s em Java, por exemplo: as `String`s são imutáveis e sempre que precisamos de uma `String` diferente, criamos uma `String` nova!

Esses objetos novos, coleções novas, `String`s novas, ou sejam o que for, precisam ser gerenciados, mantidos, em algum lugar. Aqui entramos um pouco na questão do design da aplicação: ele tem de

levar a imutabilidade em conta.

Podemos, por exemplo, ter uma thread responsável por gerenciar mudanças em pontos chave, e distribuir trabalho entre outras threads, usando objetos imutáveis. Ou podemos fazer tudo em uma única thread, mas centralizando em poucos objetos a alteração dos modelos (facilitando assim uso de múltiplas threads no futuro). Ou podemos também usar o modelo de atores para gerenciar esses objetos — entre outras estratégias possíveis. Isto é, temos muitas opções, todas dependentes do design da nossa aplicação, mas que funcionam muito bem devido à imutabilidade.

PROGRAMAÇÃO FUNCIONAL

Neste capítulo, vamos finalmente focar em Programação Funcional. E vale um aviso: esse é o ponto sem volta. A partir do momento em que o leitor entender como usar Programação Funcional no dia a dia, especialmente os recursos simples pelos quais vamos iniciar este capítulo, trabalhar sem esse recurso passará a ser tedioso, chato e improdutivo. Vejamos a seguir.

7.1 O QUE É PROGRAMAÇÃO FUNCIONAL?

Começamos a falar de Programação Funcional no capítulo anterior, para dar uma ideia do que estava por vir. Agora vamos nos aprofundar no assunto.

Formalmente, uma linguagem é considerada uma linguagem funcional se ela suportar funções de alta ordem. Essa descrição pode assustar à primeira vista, mas isso apenas quer dizer que: *"Uma função é considerada de alta ordem se ela receber uma outra função como parâmetro e/ou retornar uma função como resultado de sua invocação"*.

Essa explicação vai parecer muito estranha para quem está acostumado com linguagens não funcionais. Para esclarecer um pouco, tenha o seguinte em mente: funções e variáveis tem a mesma importância e mecânica em linguagens funcionais.

LINGUAGENS PURAS?

Existe ainda mais uma forma de classificar as linguagens funcionais: as linguagens *puramente funcionais* e as *linguagens mistas*. Scala se encaixa no segundo grupo pois, como vimos nos capítulos anteriores, suporta recursos de linguagens orientadas a objetos e, como vamos ver neste capítulo, também suporta recursos de linguagens funcionais.

Algumas outras linguagens, como *Haskell*, são puramente funcionais. Essas linguagens são ideais para quem quer, ou por algum motivo *precisa*, trabalhar apenas com construções funcionais. A curva de aprendizado será, porém, um pouco mais árdua.

No capítulo anterior, vimos a definição do método `foreach` que está disponível nas coleções do Scala:

```
abstract def foreach(f: (A) => Unit): Unit
```

Esse método é um exemplo de método de alta ordem: ele recebe uma função como parâmetro! Nosso exemplo de uso da função foi bem simples:

```
fotos.foreach(println)
```

O `println` é uma função, disponível no escopo padrão do Scala, e o passamos como parâmetro para o método `foreach`. A forma anterior é a forma mais compacta de se escrever esse código, mas poderíamos escrevê-lo também da seguinte forma:

```
fotos.foreach(f => println(f))
```

Aqui estamos sendo bem mais explícitos: estamos explicitamente definindo uma variável que vai receber cada foto da

coleção antes de passá-la ao `println`. O resultado dos dois códigos será exatamente o mesmo. Existe ainda uma terceira forma de se obter esse mesmo resultado:

```
fotos.foreach(println(_))
```

O resultado é novamente o mesmo, porém agora estamos apenas indicando explicitamente o local onde o parâmetro para da função deverá ser aplicado. Entre as três opções, a melhor escolha será sempre aquela que ficar mais legível no código em questão. Portanto, quando o leitor estiver escrevendo código em Scala, deverá sempre se lembrar dessas opções e escolher a que considerar mais legível para seu projeto.

Vamos ver um outro método das coleções, que também recebe funções como parâmetro, e é extremamente útil, o `filter`:

```
def filter(p: (A) => Boolean): Set[A]
```

O tipo de retorno do `filter` vai variar, conforme o tipo de coleção. No caso, estamos considerando o `filter` de um `Set`. Esse método recebe uma função como parâmetro, e retorna um novo `Set`. Na assinatura, `A` representa o tipo dos elementos da coleção, que neste exemplo são `Foto`s, já que estamos trabalhando com coleções de fotos.

Vamos analisar a assinatura do parâmetro `p`:

```
p: (A) => Boolean
```

É basicamente uma declaração de variável! Inclusive, poderíamos declarar uma variável `p` como anteriormente, bastando substituir o `A` por um tipo concreto, como `Foto`:

```
val p: (Foto) => Boolean = ???
```

Se executarmos esse código, teremos um `NotImplementedError` imediatamente, pois o Scala tentará

executá-lo na hora, para saber que valor colocar no `p`. Poderíamos substituir o `???` por qualquer implementação de função que respeite a assinatura definida para o `p`. Por exemplo:

```
def fotosDoJCranky(foto: Foto) = foto.owner == "jcranky"
val p: (Foto) => Boolean = fotosDoJCranky
```

Geralmente, quando usamos funções armazenadas em variáveis, como o `p` do exemplo anterior, chamamos essas funções de *function literals*, ou funções literais, em português. No nosso exemplo, o `p` é mais conceitual do que realmente útil, então vamos usar apenas a função `fotosDoJCranky` a seguir:

```
fotos.filter(fotosDoJCranky)
```

E o resultado será uma nova coleção, contendo apenas as fotos cujo `owner` for `"jcranky"`. Poderíamos ter usado o `p` também, como a seguir:

```
fotos.filter(p)
```

Poderíamos também especificar a função passada como parâmetro *inline* em vez de declará-la e depois passá-la. A melhor opção depende do restante do seu código: a função vai servir para mais alguma coisa, ou é usada apenas naquele ponto? E qual forma fica mais legível no seu caso?

O mesmo exemplo anterior, agora definido *inline*, seria:

```
fotos.filter(foto => foto.owner == "jcranky")
```

Ou então, caso o parâmetro da função passada seja usado apenas uma vez:

```
fotos.filter(_.owner == "jcranky")
```

Lembrando de que esta última opção é apenas açúcar sintático, e seu efeito é exatamente o mesmo do que o exemplo anterior.

7.2 RECEBENDO FUNÇÕES COM DOIS OU MAIS PARÂMETROS

Até aqui, passamos funções com apenas *um parâmetro* como parâmetro para outra função. Fazer isso com funções com dois parâmetros ou mais é praticamente a mesma coisa, com alguns detalhes extras na sintaxe.

Vamos estudar esse caso usando o método `sortWith`, que podemos encontrar em listas. A assinatura deste método é a seguinte:

```
def sortWith(lt: (A, A) => Boolean): List[A]
```

Ou seja, o método recebe uma função como parâmetro, que por sua vez recebe dois elementos do tipo `A` e devolve um `Boolean` como resultado. Esse booleano deverá ser `true` se o primeiro parâmetro for menor que o segundo, e `false` caso contrário. O resultado do `sortWith` é uma nova lista, também de elementos do tipo `A`, ordenados conforme as regras da função passada — `A` é o tipo de elemento da lista em questão.

Repare que a lista de parâmetros da função a ser passada, na declaração do `sortWith`, está definida entre parênteses: isso é necessário quando a função tem dois ou mais parâmetros, e opcional caso a função receba apenas um parâmetro (como vimos na seção anterior).

Essa mesma regra vale na hora de declarar a função de forma inline. Vejamos um primeiro exemplo simples, ordenando uma lista de números:

```
val numeros = List(1,3,5,2,4)
val ordenada = numeros.sortWith((x, y) => x < y)
```

A lista `ordenada` terá os seus elementos em ordem crescente. Também poderíamos ter definido a função separadamente e a

passado como parâmetro depois:

```
def ehMaior(x: Int, y: Int) = x < y

val numeros = List(1,3,5,2,4)
val ordenada = numeros sortWith ehMaior
```

Neste caso estamos também aproveitando a sintaxe flexível do Scala para gerar um código fluente. Para fechar essa sessão, vamos ordenar uma lista de fotos por título:

```
val fotosOrdenadas = fotos.sortWith((f1, f2) => f1.title < f2.title)
```

Lembrando de que nossa `case class Foto` possui um atributo chamado `title` do tipo `String`, e que podemos usar operadores como `<` e `>` com `Strings` corretamente em Scala (veremos por que quando falarmos de `implicit`s).

Para tornar o exemplo anterior ainda mais sucinto, podemos usar açúcar sintático e eliminar a declaração dos parâmetros:

```
val fotosOrdenadas = fotos.sortWith(_.title < _.title)
```

Fica bem simples, desde que o funcionamento do `_` esteja claro. A primeira vez que ele aparece, ele representa o primeiro parâmetro. Na segunda vez, o segundo, e assim por diante. Para que esse açúcar funcione, é necessário que usemos todos os parâmetros recebidos exatamente uma vez — e isso serve para funções com qualquer quantidade de parâmetros.

7.3 ENCONTRANDO ELEMENTOS: FILTER E FIND

Vamos analisar agora dois métodos muito úteis da API de coleções, que recebem funções como parâmetros: `filter` e `find`. O primeiro, como diz o nome, filtra uma coleção — já vimos um pouco desse método neste capítulo.

O ponto deste método é que, em vez de percorrermos uma coleção manualmente e verificarmos um a um todos os elementos para encontrar quais se encaixam em determinado critério ou *predicado*, nós vamos apenas escrever uma função que descreve e implementa tal critério, e deixar que a própria coleção, seja qual for, se encarregue de descobrir quais elementos estão de acordo ou não. Ou seja, navegar pelos elementos da coleção passa a ser detalhe de implementação. Em termos práticos:

```
def fotoJCranky(foto: Foto) = foto.owner == "jcranky"
val fotosDoJCranky = fotos.filter(fotoJCranky)
```

Neste exemplo, estamos gerando uma coleção nova, contendo apenas as fotos cujo `owner` seja `"jcranky"`. A segunda linha poderia ser escrita com açúcar sintático e ser lido como uma DSL:

```
val fotosDoJCranky = fotos filter fotoJCranky
```

É claro que essa função não é muito flexível nem muito reusável, mas isso é fácil de resolver:

```
def fotoDe(owner: String, foto: Foto) = foto.owner == owner
val fotosDoJCranky = fotos.filter(fotoDe("jcranky", _))
```

O código ficou ligeiramente maior, mas agora muito mais útil. De qualquer forma, os dois casos são bons exemplos de como podemos usar funções para ganhar muito em produtividade e legibilidade para o nosso código.

O outro método de coleções que vamos analisar é bem parecido com o `filter`: o `find`. Em vez de gerar uma coleção nova, com todos os elementos encontrados seguindo determinado critério, o `find` retornará um `Option[A]`, onde `A` é o tipo de elemento da coleção.

O retorno é um `Option` em vez de simplesmente `A`, pois o elemento pode não ser encontrado, e retornar um `Option` nos obriga a lidar com esse fato. A assinatura do método `filter` é:

```
def filter(p: (A) => Boolean): List[A]
```

E a assinatura do método `find` :

```
def find(p: (A) => Boolean): Option[A]
```

Repare que os dois métodos recebem parâmetros exatamente iguais, e apenas o tipo de retorno é diferente. Ou seja, podemos usar a mesma função que definimos anteriormente:

```
val fotoDoJCrankyOpt = fotos find fotoJCranky
```

Agora o resultado será um `Option` com o primeiro valor encontrado que faça a função retornar `true` ; ou `None` , caso nenhum elemento da coleção seja compatível com o critério verificado pela função passada. E claro, poderíamos ter definido a função inline nos dois casos:

```
val fotosDoJCranky = fotos.filter(_.owner == "jcranky")  
val fotoDoJCranky = fotos.find(_.owner == "jcranky")
```

Devido à simplicidade do nosso critério usado para o `filter` e o `find` , esta última é provavelmente a melhor opção. Para casos complexos, definir uma função separada, como fizemos antes, pode ser mais interessante. Como sempre, cada caso é um caso, e é sempre importante priorizar a legibilidade do código.

7.4 TRANSFORMANDO ELEMENTOS: MAP

Uma das operações mais comuns em programação funcional é o `map` : ele nos permite transformar uma coleção de objetos em uma outra coleção, que pode ser do mesmo tipo ou não. O `map` também pode ser usado em outros contextos onde é muito útil, como em `Future s` e em `Option s`. Mas mesmo nesses casos, o conceito principal continua sendo o mesmo: transformar um elemento dentro de um contêiner em outro elemento.

Estudar `Future s` está fora do nosso escopo, mas abordaremos

o uso de `map` com `Option` s no final desta seção. Como já mencionado, estamos apenas transformando elementos dentro de um contêiner em outro elemento e, no caso, tanto as coleções, os `Option` s e os `Future` s podem ser vistos como tipos diferentes de contêiners.

Nosso exemplo será novamente aplicado em uma lista de fotos. Supondo que estamos desenvolvendo o código que gerará HTML com a lista de fotos que encontramos no flickr, podemos utilizar o seguinte código:

```
val lis = fotos.map(foto => <li>{foto.title}</li>)
```

Estamos nos aproveitando do fato de Scala suportar XML nativamente para criar os `lis`. O código entre chaves (`{ e }`) é scala. Assim, podemos percorrer a lista e gerar `lis` com o conteúdo correto para cada foto. O segredo aqui é entender que o resultado do código anterior é uma nova lista, agora de nós XML.

Vejamos a assinatura do método `map` :

```
def map[B](f: (A) => B): List[B]
```

Nessa assinatura, `A` é o tipo de elemento da lista original, e `B` é o tipo de elementos da lista resultante — sendo que `A` e `B` podem ser iguais, se for o que queremos. E como saber que tipo é o `B` ? Isso depende do tipo de retorno da função que passarmos para o `map`. Vamos declarar a função usada no `map` explicitamente:

```
def geraLi(foto: Foto): scala.xml.Elem = <li>{foto.title}</li>
```

Estamos declarando o tipo de retorno apenas para facilitar a leitura, mas ele poderia ter sido omitido. A nossa função sabe basicamente transformar uma foto em uma tag XML ``. O código a seguir tem exatamente o mesmo resultado do anterior:

```
val lis = fotos map geraLi
```

O código a seguir ilustra como poderíamos gerar uma página html completa simples:

```
val html =  
  <html>  
    <body>  
      <ul>  
        {fotos map geraLi}  
      </ul>  
    </body>  
  </html>
```

Extremamente simples. Se o leitor já tiver utilizando o Play Framework, provavelmente o que fizemos será familiar: o mecanismo de templates desse framework utiliza essa mistura de tags e código scala com transformações e outros truques, para nos permitir gerar páginas complexas com certa facilidade.

XML COM SCALA 2.11

A partir do scala 2.11 alguns recursos foram removidos da distribuição padrão da linguagem e precisam ser adicionados ao nosso classpath separadamente, como qualquer outra biblioteca. O suporte a XML é um desses recursos.

Mais informações especificamente sobre o `scala-xml` podem ser encontradas na sua página oficial no GitHub: <https://github.com/scala/scala-xml>.

Outro uso bastante comum de `map` são com `Option` s. Quando falarmos de *for comprehensions*, isso vai fazer ainda mais sentido, mas vamos ver um exemplo simples agora. Primeiro, vamos supor a seguinte `case class` para nossas fotos:

```
case class Foto(id: String, owner: String,  
  server: Int, title: String, tags: Option[ List[ String ] ])
```

Ou seja, agora temos uma lista de tags, que é opcional. Dada uma variável chamada `foto`, instância da classe anterior, o que teremos na variável `tagsText` no código a seguir?

```
val tagsText = foto.tags.map(tags => tags.mkString(","))
```

Primeiro, o tipo da `tagsText` será `Option[String]`. Essa é uma característica importante do `map`: ele sempre trabalha nos elementos internos do contêiner em questão, e o resultado será sempre um novo contêiner, do mesmo tipo. No caso, pense em um `Option` como uma coleção de zero ou um elemento.

Caso o campo `tags` seja `None`, ou seja, vazio, a função passada para o `map` não será executada e o resultado será um `None`. Em resumo, um `map` em um `Some` vai gerar um novo `Some`, com a nossa função aplicada sobre o valor original, e um `map` em um `None` simplesmente retornará o próprio `None`.

O código anterior pode ficar ainda mais interessante se nos lembrarmos das operações disponíveis em `Option` s:

```
val tagsText = foto.tags.map(tags => tags.mkString(",")).getOrElse("")
```

Agora nossa `tagsText` será uma `String` em vez de um `Option[String]`. Caso nossa `foto` não tenha nenhuma tag, teremos uma `String` vazia. Caso contrário, teremos uma `String` com as tags separadas por vírgulas.

Na API para acesso ao Flickr, essa estratégia será útil na hora de definir a `url` de busca de fotos:

```
def buscaFotos(tag: Option[String]) = {  
  val apiKey = "minha-apiKey"  
  val method = "flickr.photos.search"  
  val tagText = tag.map("&tags=" + _).getOrElse("")  
  
  val url = s"http://api.flickr.com/services/rest/?method=$method&  
api_key=$apiKey$tagText"  
  Source.fromURL(url).getLines.foreach(println)
```

```
}
```

Com esse código, foi bem simples omitir completamente o parâmetro `tags` do serviço do flickr, caso não passemos uma tag para o método `buscaFotos`.

7.5 MAPEANDO RESULTADOS COM COLEÇÕES ANINHADAS

Uma outra operação muito comum e importante em Scala (e em linguagens funcionais de forma geral) é o `flatMap`. Com ele, podemos fazer uma cadeia de mapeamentos mesmo quando o resultado de uma operação for uma coleção inteira, ou seja, quando o resultado de processar um determinado elemento for uma nova coleção em vez de um outro elemento simples.

Para entender melhor por que esse mapeamento precisa ser diferente de um normal, vejamos um exemplo a seguir. Desta vez, em vez de fazer uma consulta de fotos por uma tag, faremos uma série de consultas, usando uma série de tags diferentes. Primeiro, criamos uma lista de tags:

```
val tags = List("scala", "java", "typesafe")
```

Vamos agora fazer uma pesquisa para cada uma dessas tags, usando o `map`:

```
val fotos = tags.map(tag => buscaFotos(Some(tag)))
```

Para este exemplo, estamos considerando um método `buscaFotos` com a seguinte assinatura:

```
def buscaFotos(tag: Option[String]): List[Foto]
```

Dessa vez, em vez do `println` que fizéssemos na implementação anterior do `buscaFotos`, estamos usando uma implementação que retorna uma lista de fotos pronta. Veremos

como fazer essa implementação no capítulo de XML, pois para isso precisaremos parsear o XML de resposta do Flickr.

O ponto importante aqui é que, como cada chamada ao método `buscaFotos` resulta em uma coleção de fotos, teremos uma *lista de listas de fotos* como resultado final. Se executarmos o código anterior no REPL, teremos o seguinte como resultado:

```
fotos: List[scala.collection.immutable.Seq[Foto]] = // resultado o  
omitido
```

Obviamente isso não é ideal, visto que, para percorrer todas as fotos, teríamos uma complexidade extra de precisar percorrer cada lista interna da lista de listas... Até mesmo explicar isso textualmente não é agradável. E esse problema é recursivo, pois uma eventual transformação aninhada em algum momento poderia gerar uma lista de listas de listas.

Resolver isso é bem simples: basta usar o `flatMap`. Literalmente, trocaremos o `map` anterior por um `flatMap`, como a seguir, e esse método automaticamente faz com que toda a estrutura de contêiners seja *compactada* em um único nível:

```
val fotosCompactadas = tags.flatMap(tag => buscaFotos(Some(tag)))
```

E agora, o resultado será:

```
fotos: List[Foto] = // resultado omitido
```

Ou seja, o `flatMap` abriu as coleções internas, e colocou todo o resultado em uma única coleção, na qual podemos navegar diretamente através dos resultados em uma única estrutura simples, compacta, *plana*.

7.6 AGREGANDO RESULTADOS: FOLD E REDUCE

As operações de `fold` e `reduce` estão, junto com `map`, entre as mais úteis e mais utilizadas em programação funcional. Enquanto `map` realiza transformações, geralmente alterando o conteúdo de coleções (ou outros tipos de contêineres) de um tipo para outro, tanto `fold` quanto `reduce` agregam valores.

Esse tipo de operação é extremamente útil, mas um pouco mais complexa em um primeiro momento. O exemplo clássico seria a soma de todos os elementos de uma coleção de números:

```
val numeros = List(1,2,3,4,5)
val soma = numeros.reduceLeft((acumulado, x) => acumulado + x)
```

A variável `soma` terá o valor `15` quando a operação `reduce` terminar. A operação `reduce` vai ocorrer percorrendo todos os elementos da coleção, um por um, e nossa função será aplicada para cada elemento.

A assinatura (simplificada) do `reduceLeft` é a seguinte:

```
def reduceLeft(f: (A, A) => A): A
```

Simplificamos um pouco a assinatura e vamos voltar a ela no capítulo seguinte. Por enquanto, entenda que o tipo de retorno do `reduceLeft` é basicamente do mesmo tipo dos elementos da coleção. Isto é, o resultado é a *redução* dos elementos para um único valor, com base na função passada como parâmetro.

Essa função é mais um exemplo de função de alta ordem: ela recebe uma outra função como parâmetro, uma função com dois parâmetros neste caso. O primeiro parâmetro será o valor acumulado pelo nosso algoritmo, que neste caso calcula a soma dos elementos; e o segundo será cada um dos elementos da nossa coleção, um por um. O resultado da função passada em uma determinada iteração será usado como valor acumulado para a próxima iteração.

Temos apenas um problema: qual será o valor utilizado no acumulador na primeira iteração do `reduce`, quando ainda não calculamos nada? Especificamente no caso do `reduceLeft`, este valor será o primeiro elemento da coleção (ou o último, se usarmos o `reduceRight`).

Como mencionamos anteriormente, o tipo retornado pela nossa função tem de ser compatível com o tipo de elementos da coleção. Então, os elementos da coleção são também compatíveis com o tipo do nosso acumulador.

Passo a passo, é assim que acontece o cálculo da soma no exemplo anterior:

```
((((1 + 2) + 3) + 4) + 5)
((3 + 3) + 4) + 5)
(6 + 4) + 5)
10 + 5)
15
```

Veja o equivalente em código Scala:

```
def somar(x: Int, y: Int) = x + y
val somaManual = somar(somar(somar(somar(1, 2), 3), 4), 5)
```

Repare que usamos uma variante do `reduce` chamada `reduceLeft`. Existem três variantes principais desse método: `reduce`, `reduceLeft` e `reduceRight`.

A primeira versão percorre os elementos seguindo uma ordem arbitrária e indefinida, ou seja, não podemos contar com essa ordem em nossos algoritmos. A versão `reduceLeft` percorre os elementos da coleção a partir da esquerda, como vimos; e `reduceRight` faz essa iteração a partir da direita, ou seja, o oposto do que vimos anteriormente.

O exemplo anterior da soma de números é interessante em termos didáticos, entretanto, na prática, não precisamos dele: o

Scala já possui um método para isso. Veja:

```
val somaPronta = List(1, 2, 3, 4, 5).sum
```

A variável `somaPronta` terá o valor `15`. O mais interessante deste método é que, apesar de ele estar definido na `trait Traversable` — e portanto disponível para todas as coleções —, teremos um erro em tempo de compilação caso tentemos invocá-lo em um coleção de valores "não somáveis":

```
val somaInvalida = List("um", "dois", "três", "quatro", "cinco").sum
```

Veremos como isso é possível no capítulo 12. *Implicits*. Agora vamos supor que queremos agrupar todas as tags de todas as fotos de um álbum.

Com `reduce`, não seria possível fazer essa operação, pois temos uma coleção de fotos, e o que queremos como resultado é uma nova coleção, contendo as tags. O resultado do `reduce` sempre será do mesmo tipo da coleção original — no caso do exemplo anterior, um número, e no caso da nossa coleção de fotos, esse resultado seria uma foto.

O `fold` traz a solução para isso. Ele funciona de forma muito parecida com o `reduce`, mas permite que tenhamos como resultado qualquer tipo de elemento, o que é exatamente o nosso requisito para a lista de tags.

Nossa `case class Foto` está definida da seguinte forma para esse exemplo:

```
case class Foto(id: String, owner: String, server: Int,  
  title: String, tags: Option[ List[ String ] ])
```

E vamos trabalhar com a seguinte lista de fotos:

```
val foto1 = new Foto("id1", "jcranky", 1,  
  "uma foto do jcranky", Some(List("livro", "scala")))  
val foto2 = new Foto("id2", "jcranky", 1,
```



```

    "outra foto do jcranky", Some(List("scala", "jcranky"))))
val foto3 = new Foto("id3", "jcranky", 1,
    "mais uma foto do jcranky", Some(List("livro", "jcranky")))

val fotos = List(foto1, foto2, foto3)

```

O resultado que queremos aqui é uma lista com as tags "livro", "jcranky" e "scala", pois estas são as tags que aparecem nas fotos acima. Vamos começar definindo a estrutura do `fold` que fará essa operação:

```
val tags = fotos.foldLeft(Set.empty[String])( ??? )
```

Temos novamente a diferenciação do tipo de operação: *Left*, *Right* ou *indefinida*. Isso funciona exatamente da mesma forma que discutimos quando falamos de `reduce`. A principal diferença aparece a seguir: temos duas listas de parâmetros em vez de uma.

A primeira lista representa o ponto de partida do `fold`, o elemento "zero" em que os resultados serão acumulados. Se estivéssemos implementando a soma de uma lista de números, por exemplo, esse elemento poderia ser literalmente o número `0`.

No nosso exemplo, estamos usando um `Set` de `String`s, vazio. Conforme formos percorrendo a lista de fotos, vamos adicionar suas tags a esse `Set`.

Como mencionado anteriormente, poderíamos ter usado qualquer outro tipo de valor como elemento zero: uma lista, um número, uma `String` — o que fizer sentido para o algoritmo que estamos implementando. A segunda lista de parâmetros é a lógica em si: é exatamente o que faríamos com o `reduce`. Vamos finalizar nosso `foldLeft`:

```

val tags = fotos.foldLeft(Set.empty[String])(
    (tags, foto) => foto.tags.map(tags ++ _).getOrElse(tags)
)

```

Tome cuidado apenas com o tipo de retorno da função passada

ao segundo parâmetro. Ele deve ser igual ao tipo do elemento zero, e não igual ao tipo dos elementos da coleção, como seria com o `reduce`. Nossa função retornará o `Set` de tags somado às tags da foto atual, caso a foto tenha tags; ou simplesmente o `Set` de tags existente, caso a foto não tenha nenhuma tag definida.

Para deixar isso um pouco mais claro, vejamos a assinatura simplificada do método `foldLeft` a seguir:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

Note que o tipo do parâmetro `z` (o valor zero) é `B`, que é o mesmo tipo retornado pela função passada na segunda lista de parâmetros, e também o mesmo tipo retornado pelo próprio `foldLeft` como resultado final. O único ponto em que outro tipo é usado é no segundo parâmetro da função `op`, onde o tipo é `A`, que representa o tipo de elementos da coleção sendo trabalhada.

Em resumo, a principal diferença entre `fold` e `reduce`, portanto, é que no `fold` nós definimos o elemento inicial, o ponto zero ou inicial do acumulador, mas no `reduce` não. No `reduce`, o ponto de partida é o primeiro elemento da coleção.

Essa definição explícita do elemento inicial no `fold` é o que nos permite ter qualquer tipo como resultado, já que podemos usar qualquer tipo de elemento como o tal elemento zero. Isso também quer dizer que poderíamos reescrever qualquer `reduce` usando `fold`. Como exercício para o leitor, fica reimplementar o exemplo anterior da soma, usando `fold` em vez de `reduce`.

TIPAGEM AVANÇADA

Neste capítulo, vamos estudar alguns pontos um pouco mais avançados do sistema de tipos da linguagem Scala. A ideia é que, se o compilador já era capaz de nos ajudar a evitar alguns tipos de problemas, mantendo nossa produtividade, vamos agora aumentar a rede de segurança que podemos construir com a ajuda desse ambiente.

Alguns pontos, principalmente a segunda metade deste capítulo, podem ser um pouco mais complicados de se entender. Não se preocupe, eles são muito mais importantes para desenvolvedores de bibliotecas do que desenvolvedores de aplicações, e o leitor pode tranquilamente decidir voltar a esses pontos em algum momento no futuro.

A COMPLEXIDADE DO SCALA

Alguns recursos ligados à tipagem da linguagem Scala podem ser realmente complexos, logo, mais difíceis de se entender. É importante entender que, apesar de tais recursos existirem, eles não são indispensáveis para o trabalho do dia a dia de todos os desenvolvedores. Recursos como *covariância* e *contravariância*, *Types Bounds*, *Phantom Types*, *Type Projection*, *Path Dependant Types* etc. acabam sendo mais úteis em situações de nicho (como no desenvolvimento de bibliotecas) do que no desenvolvimento de aplicações.

8.1 TIPOS PARAMETRIZADOS

Já vimos vários usos de tipos parametrizados durante o livro e agora é a hora de formalizarmos esse conhecimento. O conceito é bastante intuitivo, por isso conseguimos chegar até aqui sem falar deles. Porém, existem alguns recursos mais avançados ligados a tipos parametrizados que pedem um estudo mais cuidadoso.

Vamos primeiro entender o uso básico de tipos parametrizados. Quando criamos nossos conjuntos de fotos, fizemos algo como:

```
val foto1 = new Foto("id1", "jcranky", 1, "uma foto do jcranky")
val foto2 = new Foto("id2", "jcranky", 1, "outra foto do jcranky")
val foto3 = new Foto("id3", "jcranky", 1, "mais uma foto do jcranky")

val fotos = Set(foto1, foto2, foto3)
```

Não dissemos explicitamente em lugar algum que este conjunto (ou *Set*) é um *conjunto de fotos*, mas Scala consegue inferir esse fato. No REPL, o resultado desse código seria algo como o seguinte:

```

foto1: Foto = Foto(id1,jcranky,1,uma foto do jcranky)
foto2: Foto = Foto(id2,jcranky,1,outra foto do jcranky)
foto3: Foto = Foto(id3,jcranky,1,mais uma foto do jcranky)

fotos: scala.collection.immutable.Set[Foto] =
    Set(Foto(id1,jcranky,1,uma foto do jcranky),
        Foto(id2,jcranky,1,outra foto do jcranky),
        Foto(id3,jcranky,1,mais uma foto do jcranky))

```

Repare que o conjunto de fotos foi declarado como um `Set[Foto]`. Isso quer dizer que podemos colocar apenas fotos neste conjunto. Poderíamos também declarar o tipo explicitamente:

```

val fotosExplicito: Set[Foto] = Set(foto1, foto2, foto3)

```

E dessa forma, o seguinte código não compilaria:

```

val fotos: Set[Foto] = Set(foto1, foto2, foto3, "foto em string?")

```

Neste último exemplo, precisamos tomar cuidado com a inferência de tipos. Se não tivéssemos definido explicitamente que queremos um conjunto de Fotos, o código compilaria — a diferença seria o tipo inferido para o conjunto resultante:

```

val fotos = Set(foto1, foto2, foto3, "foto em string?")

```

E o resultado seria algo como:

```

fotos: scala.collection.immutable.Set[ java.io.Serializable ] =
    Set(Foto(id1,jcranky,1,uma foto do jcranky),
        Foto(id2,jcranky,1,outra foto do jcranky),
        Foto(id3,jcranky,1,mais uma foto do jcranky),
        foto em string?)

```

Ou seja, Scala utilizou o tipo comum entre os elementos que adicionamos ao `Set`. E devido à inferência de tipos, chegou à conclusão de que esse conjunto é um `Set[Serializable]`, o tipo pai comum entre `Foto` e `String`s.

Um caso no qual a parametrização de tipos é bastante interessante é o método `empty` disponível nos objetos da maior parte das coleções do Scala. Como diz o nome, ele permite que

criemos coleções vazias, de forma bem simples. Por exemplo, podemos usar o código a seguir para criar uma nova lista vazia:

```
val listaVazia = List.empty
```

Essa abordagem tem um problema. O tipo da lista é definido como `Nothing`, pois o compilador não tem como saber qual é o tipo dos elementos que queremos armazenar, e escolhe o tipo mais específico possível para listas vazias, o `Nothing`. O código anterior geraria a seguinte saída no REPL:

```
listaVazia: List[Nothing] = List()
```

Porém, graças à parametrização de tipos, esse problema é fácil de resolver:

```
val listaStringVazia = List.empty[String]
```

E agora temos um resultado totalmente sob nosso controle. O resultado do código anterior, no REPL, seria o seguinte:

```
listaVazia: List[String] = List()
```

8.2 LIMITES DE TIPOS: TYPE BOUNDS

Além de classes e afins, métodos também podem ser parametrizados. Já vimos um pouco disso quando falamos do método `map` no capítulo passado. Relembrando, a assinatura do método `map` é exibida no código a seguir:

```
def map[B](f: (A) => B): List[B]
```

O método possui um tipo parametrizado, chamado `B`, que serve para capturar o tipo de retorno da função recebida e usá-lo para definir o tipo de retorno do próprio `map` — ou melhor, usado para definir o tipo dos elementos da coleção retornada. Neste caso, não há nenhum tipo de restrição, e `B` pode ser basicamente qualquer coisa.

Usando *Type Bounds*, ou limite de tipos, porém, podemos restringir, *limitar*, os tipos aceitáveis para determinados tipos parametrizados. Vamos ver outro exemplo extraído da API de coleções do Scala, o método `reduceLeft` :

```
def reduceLeft[B >: A](f: (B, A) => B): B
```

Temos novamente um tipo parametrizado chamado `B` , mas repare que dessa vez o `B` está definido de forma diferente: ele possui uma limitação nos tipos aceitáveis. No caso, temos depois do `B` a seguinte limitação: `>: A` . Uma forma simples de se ler essa restrição seria *B super A*, ou *B estendido por A*. Ou seja, qualquer tipo que seja o próprio `A` ou algum supertipo (tipo pai) dele são válidos na posição do `B` .

O objetivo do `reduceLeft` é reduzir os elementos de um determinado contêiner a um único elemento, e esse elemento precisa ser compatível com os elementos originais do contêiner. Já falamos um pouco do `reduceLeft` no capítulo anterior, mas usamos uma versão simplificada da sua assinatura. Agora com a assinatura completa, podemos entender alguns detalhes do algoritmo deste método, recebendo uma função, que recebe dois parâmetros e devolve um valor.

Na função a ser passada ao `reduceLeft` , o primeiro parâmetro é do tipo `B` , e o segundo é do tipo `A` . Tanto o valor retornado pela função quanto o valor final do `reduceLeft` são do tipo `B` . Isso, na prática, quer dizer que a função recebe um valor acumulado calculado pela iteração anterior da própria função e um elemento da coleção.

O detalhe é que, na primeira vez que a função é invocada, ainda não há um valor acumulado a ser passado pela função, portanto são passados dois valores da própria coleção. Isso obriga que o parâmetro acumulador seja de tipo compatível com o tipo dos

elementos da coleção, logo, faz a restrição $B \geq A$ ser necessária. Se não fosse assim, o exemplo que veremos a seguir seria impossível de se implementar.

Portanto, essa restrição nos permite fazer algo interessante: criar uma função que pode ser aplicada a tipos diferentes de coleções, desde que elas tenham elementos que *compartilhem um tipo pai em comum*. Vejamos um exemplo. Primeiro, para este exemplo, trabalharemos com uma versão simplificada da hierarquia de fotos e vídeos, a do código a seguir:

```
trait Media {  
  val tags: Set[String]  
}  
case class Foto(id: Int, tags: Set[String]) extends Media  
case class Video(id: Int, tags: Set[String]) extends Media
```

Com a hierarquia criada, vamos agora criar dois `Set` s, um com fotos e outro com vídeos:

```
val fotos = Set(Foto(1, Set("scala", "jcranky")), Foto(2, Set("jvm",  
  "jcranky")))   
val videos = Set(Video(1, Set("praia", "ipanema")), Video(2, Set("campo", "ferias")))
```

A seguir, vamos criar uma função que funciona com o `reduceLeft` dos dois `Set` s anteriores. O segredo é fazer com que essa função trabalhe com o tipo *pai* da `Foto` e do `Video` , ou seja, com a `Media` :

```
def acumulaTags(media: Media, outraMedia: Media): Media =  
  new Media { val tags = media.tags ++ outraMedia.tags }
```

A função `acumulaTags` recebe duas `Media` s e retorna uma nova `Media` como resultado. Neste resultado, para simplificar, estamos criando uma nova `Media` anônima que contém as tags das duas `Media` s recebidas nos parâmetros. Vejamos como aplicar essa função nos `Set` s de fotos e vídeos a seguir:

```
val tagsFotos = fotos.reduceLeft(acumulaTags)
```



```
val tagsVideos = videos.reduceLeft(accumulaTags)
```

Esse tipo de restrição é o que chamamos de *Lower Type Bounds*, ou limites de tipo inferior: estamos definindo o tipo base do parâmetro, mas qualquer tipo pai dele também é aceitável. Para tentar esclarecer um pouco mais, podemos imaginar o tipo concreto do `reduceLeft` dos `Set`s anteriores. No caso do `Set` de fotos, a assinatura da função seria algo como o código a seguir:

```
def reduceLeft[B >: Foto](f: (B, Foto) => B): B
```

O `B` será entendido como `Media` ao passarmos a função `accumulaTags`. Imaginar como seria para o caso do `Video` fica como exercício para o leitor.

Existe também a restrição oposta: *Upper Type Bounds*, ou limites de tipo superior. Esse tipo de restrição faz exatamente o oposto da anterior, isto é, aceita qualquer valor de um determinado tipo, ou algum tipo *filho* dele. A sintaxe é invertida também: `B <: A`.

Para entender um pouco melhor esse caso, vamos ver um exemplo. Aqui, vamos continuar usando nossa hierarquia com `Media` sendo o tipo pai, e os tipos filhos `Foto` e `Video`. Com essa hierarquia em mente, vamos criar duas funções capazes de imprimir `Set`s de `Media`s.

```
def printaFotos(medias: Set[Media]) = medias.foreach(println)
def printaFotos2[T <: Media](medias: Set[T]) = medias.foreach(println)
```

```
val setDeFotos: Set[Foto] = Set(Foto(1, Set.empty))
```

A primeira função aceita exatamente um `Set` de `Media`s; a segunda, um `Set` de `Media` ou de qualquer tipo filho. Aproveitamos e criamos também um `Set` de fotos. Se invocarmos a primeira função com essas fotos, como no código a seguir:

```
printaFotos(setDeFotos)
```

Teremos um erro como o seguinte:

```
Error:(30, 16) type mismatch;  
found   : scala.collection.immutable.Set[Foto]  
required: Set[Media]                ^
```

Já se invocarmos a segunda função com o `Set` de fotos, tudo funciona normalmente. É importante manter em mente que parte da restrição aplicada à primeira função é devido ao fato de o `Set` ser invariante.

Se tentássemos reproduzir o mesmo cenário com uma `List`, por exemplo, as duas funções compilariam normalmente e funcionariam como o esperado. Vamos falar um pouco sobre tipos invariantes, covariantes e contravariantes a seguir.

8.3 TIPOS INVARIANTES, COVARIANTES E CONTRAVARIANTES

Algumas vezes queremos criar tipos que precisam existir em função de outros tipos, ou seja, só fazendo sentido se forem criados em juntos com outros tipos. De maneira mais simples, esses são os tipos *parametrizáveis*, isto é, um elemento que recebe um tipo como parâmetro.

O uso canônico desse tipo de recurso é nas coleções. Um `Set`, por exemplo, é definido em Scala (simplificando) da seguinte forma:

```
trait Set[A]
```

Temos aqui um `Set` de elementos exclusivamente do tipo `A`, ou seja, `A` será definido quando o `Set` for instanciado, e não aceita nenhum tipo de variação. Dizemos então que `A` é invariante. Isso quer dizer que, se criarmos, por exemplo, um `Set` de fotos, esse `Set` terá de ser sempre da mesma classe `Foto`.

Vamos continuar usando a hierarquia de classes definida

anteriormente neste capítulo, e no código a seguir vamos criar um `Set[Foto]` :

```
val fotos = Set(Foto("1", "jcranky", 123, "scalax", None))
def printaFotos(fotos: Set[Foto]) = fotos foreach println
```

Também aproveitamos para redefinir o método `printaFotos` que vimos na sessão anterior. O método `printFotos` simplesmente percorre o `Set` e imprime todos os seus elementos. Podemos testá-lo da seguinte forma:

```
printaFotos(fotos)
```

E o resultado deve ser algo maluco como:

```
Foto(1,Set())
```

Até aqui, nada muito enigmático. Vamos agora criar uma nova classe, estendendo `Foto` :

```
class FotoDetalhada(id: Int, owner: String, server: Int, title: String,
                    tags: Set[String], detalhes: String)
  extends Foto(id, tags)

val fotosDet =
  Set(new FotoDetalhada(2, "jcranky", 321, "scalax2", Set.empty,
    "detalhes da foto"))
```

E novamente já aproveitamos para criar um `Set` com uma instância dessa nova classe. Agora a parte mais interessante, algo que já mencionamos rapidamente na sessão anterior. Ser tentarmos invocar o `printaFotos` com o `Set` de `FotoDetalhada`, algo como no código a seguir:

```
printaFotos(fotosDet)
```

O resultado será um erro de compilação, com uma mensagem similar à seguinte:

```
Error:(19, 15) type mismatch;
 found   : scala.collection.immutable.Set[FotoDetalhada]
 required: Set[Foto]
```

```
printaFotos(fotosDet)
```

Ou seja, o método esperava o tipo *exato* `Set[Foto]` . Quando usamos invariância, estamos portanto dizendo que não existe nenhuma relação de herança entre os tipos, mesmo que os seus tipos genéricos possuam tal relação. No nosso caso, `FotoDetalhada` é filha de `Foto` , mas como `Set` é invariante no seu tipo `A` , isso não importa.

Para fazer o exemplo anterior funcionar, podemos usar limites de tipo superior (*Upper Type Bounds*) como vimos na seção anterior, ou podemos usar uma coleção covariante como é o caso da `List` . Vejamos a assinatura dessa classe, novamente simplificando um pouco:

```
sealed abstract class List[+A]
```

Dessa vez, o tipo parametrizado foi definido como `+A` em vez de simplesmente `A` . Vamos declarar novamente a coleção de fotos e a de fotosDetalhadas :

```
val fotos = List(new Foto(1, Set.empty))
val fotosDetalhadas = List(
  new FotoDetalhada(2, "jcranky", 321, "scalax2", Set.empty, "detalhes da foto")
)
```

Temos dessa vez uma `List[Foto]` e uma `List[FotoDetalhada]` . Da mesma forma, vamos redefinir o método `printFotos` :

```
def printFotos(fotos: List[Foto]) = fotos foreach println
```

Só nos resta testar o cenário novamente, primeiro com as fotos :

```
printFotos(fotos)
```

E o resultado seria:

```
Foto(1,Set())
```

Vamos agora tentar passar o `fotosDetalhadas` para o método:

```
printFotos(fotosDetalhadas)
```

E dessa vez, a invocação funciona:

```
Foto(2, Set())
```

O que aconteceu é que, como `List` é covariante no tipo `A`, uma lista `List[FotoDetalhada]` é considerada como filha de uma lista `List[Foto]`. Logo, o método `printFoto` pode usar polimorfismo normalmente. Ou seja, a covariância criou uma relação de herança entre duas `List`s, com base no tipo dos seus elementos.

NÃO USE HERANÇA COM CASE CLASSES

Talvez o leitor tenha notado que evitamos usar *case classes* na classe `FotoDetalhada`. Isso foi de propósito, já que Scala não lida muito bem com herança de *case classes*. O problema é que os métodos `equals` e `hashCode` (e possivelmente outros) gerados nunca levam em conta a herança — e nem poderiam, pois precisariam conhecer todos os filhos existentes para isso. Portanto, as implementações geradas seriam incorretas e poderiam causar erros bem difíceis de se debugar.

Nosso exemplo, na verdade, não é ideal também. Isso porque, como o `equals` gerado pela *case class* `Foto` não conhece a `FotoDetalhada`, a seguinte comparação seria verdadeira:

```
val foto = Foto(1, Set.empty)
val fotoDetalhada = new FotoDetalhada(1, "jcranky", 123, "scala", Set.empty, "nada")

println(foto == fotoDetalhada)
```

O ideal provavelmente seria definir a `Foto` como uma classe normal, e a `FotoDetalhada` como uma *case class*. Mas isso depende dos detalhes do sistema sendo implementado.

A *contravariância* é o caso mais complicado de se explicar e entender, embora a teoria seja simples. A *contravariância* é o oposto da *covariância* e, em vez de termos uma relação de herança na qual uma classe com tipos parametrizados aceitam os filhos daquele tipo, a relação aceita os tipos *pais*.

Vejamos um exemplo desse recurso sendo usado na API do Scala. Vamos ver uma assinatura simplificada para focar no que estamos estudando agora:

```
trait Function1[-T1, +R]
```

O `Function1` define o suporte a funções que recebem um único parâmetro `-T1` e devolvam um valor `+R`. Quando usamos funções desse tipo em Scala, um objeto que implementa essa `trait` é criado.

Note o uso da *covariância* no tipo `+R`, que representa o tipo de retorno da função. Note também que o tipo usado para o parâmetro, `-T1`, é definido usando um sinal de `-`, o que indica que esse tipo é *contravariante*.

Vamos entender por que o tipo do parâmetro da função precisa ser *contravariante*. No código a seguir, declaramos uma função que imprime uma foto, de uma forma um pouco diferente: guardando a referência para a função diretamente em uma variável, ou seja, usando uma *function literal*:

```
val printaFoto: Foto => Unit = ???
```

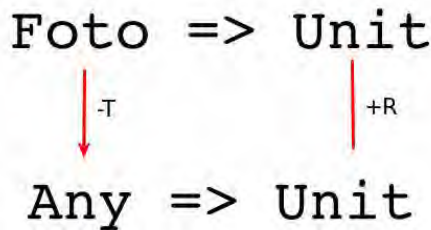
O `printaFoto` é, portanto, uma função de `Foto` para `Unit`. Ainda não definimos uma implementação, pois aí está o segredo da contravariância. Vamos começar com uma implementação bem simples que imprime a foto no console:

```
val printaFoto: Foto => Unit = println
```

Estamos atribuindo diretamente a função `println` à nossa função. O `println` é uma função na forma `Any => Unit`, ou seja, ela recebe qualquer coisa e não devolve nada.

Lembrando de que uma função será sempre *contravariante* nos seus parâmetros, e *covariante* no tipo do retorno, vemos que `println` é compatível com a assinatura da nossa `printaFoto`, pois `Any` é pai de `Foto` e `Unit` é exatamente o retorno que estamos esperando. Porém, poderia ser um tipo filho de `Unit` também, se tal tipo existisse. A figura a seguir ilustra esse cenário

um pouco melhor:



Preste atenção na direção do relacionamento de extensão. Um outro exemplo um pouco mais interessante seria o seguinte, no qual vamos declarar uma função usando nossa `FotoDetalhada` :

```
val printaFotoDetalhada: FotoDetalhada => Unit = (foto: Foto) => p
rintln(foto)
```

Agora estamos declarando que a função deve receber uma `FotoDetalhada` e devolver `Unit` . Em seguida, atribuímos uma função que recebe uma `Foto` e devolve `Unit` a essa função literal. É o mesmo relacionamento de herança do exemplo anterior, agora em um cenário um pouco mais prático.

O interessante é notar que, apesar de ser um conceito complexo de se entender, ele faz muito sentido. O tipo da variável que referencia à função recebe uma `FotoDetalhada` , mas a implementação recebe apenas uma `Foto` .

Na prática, estamos invocando a função que recebe a `Foto` . E se fosse possível passar um parâmetro filho de `FotoDetalhada` , teríamos potencialmente uma função impossível de ser invocada, pois seria um tipo completamente desconhecido para a implementação. Vamos inverter o exemplo anterior para deixar isso mais claro:

```
val printaFoto: Foto => Unit =
```

```
(fotoDetalhada: FotoDetalhada) => println(fotoDetalhada)
```

Esse exemplo não compila, gerando uma mensagem de erro como a seguinte:

```
Error:(11, 59) type mismatch;  
found   : FotoDetalhada => Unit  
required: Foto => Unit  
val printaFoto: Foto => Unit =  
    (fotoDetalhada: FotoDetalhada) => println(fotoDetalhada)
```

O problema é que, como a função declarada pode receber qualquer tipo de `Foto`, se esse código compilasse, seria possível passar outros filhos de `Foto` além da `FotoDetalhada`, sendo que a implementação em questão só é compatível com a `FotoDetalhada`.

UM POUCO DE AÇÚCAR: FOR COMPREHENSIONS

Vamos agora discutir um tópico focado em melhorar a legibilidade do código. Não vamos introduzir nenhum recurso realmente novo, mas sim estudar um pouco de *for comprehensions* que, pelo menos em Scala, é apenas açúcar sintático para simplificar a leitura de outros recursos que de outra forma poderiam acabar sendo mais complexos de se entender.

9.1 PERCORRENDO MÚLTIPLAS COLEÇÕES DE FORMA LEGÍVEL

Vamos supor que queremos encontrar todas as fotos para um conjunto de tags, pesquisadas individualmente. Vamos começar com o seguinte conjunto de tags, e com o objeto a ser usado no acesso ao Flickr:

```
val caller = new FlickrCaller()
val tags = Set("scala", "java", "jvm")
```

Tendo essa lista em mãos, que poderíamos ter obtido de um usuário, por exemplo, vamos fazer um simples loop para executar uma pesquisa no Flickr para cada tag e, em seguida, um loop para imprimir as fotos na tela.

```
tags.foreach { tag =>
  val fotos = caller.buscaFotos(tag)
  fotos foreach println
```

}

O código anterior é relativamente simples, porém com uma legibilidade um pouco aquém do que gostaríamos, devido ao aninhamento dos `foreach` percorrendo as coleções de tags e fotos. Além disso, se em vez de imprimir o valor na tela estivéssemos guardando o resultado acumulado, o código anterior ficaria ainda mais burocrático.

Em linguagens não funcionais, isso seria ainda pior, mas isso não quer dizer que não podemos melhorar. Scala nos oferece uma estrutura de `for` `s` que nada mais são do que açúcares sintáticos, que podemos usar para melhorar em muito a legibilidade desse tipo de código. Vejamos o mesmo exemplo ilustrado pelo código anterior, reescrito usando *for comprehensions*:

```
for {  
  tag <- tags  
  foto <- caller.buscaFotos(Some(tag))  
} println (foto)
```

Dentro do `for`, cada `<-` indica uma chamada ao método `foreach` da coleção à direita. E cada `foreach` subsequente é executado dentro do escopo do anterior, ou seja, aninhado. Poderíamos percorrer quantas coleções quiséssemos dessa forma, sem prejudicar a legibilidade significativamente. Vale lembrar novamente que o resultado da execução dos dois exemplos anteriores é exatamente o mesmo, pois o segundo caso nada mais é do que açúcar sintático para o primeiro.

As duas opções são funcionalmente iguais, porém a segunda acaba sendo um pouco mais legível. Este é mais um caso no qual, para se decidir qual opção usar, devemos ter a legibilidade em mente. Não existe uma regra que diga se é melhor usar ou não *for comprehensions*. Na prática, o leitor sempre deve escolher a opção que ficar mais legível para o seu cenário.

9.2 MANTENDO A IMUTABILIDADE

Na seção anterior, percorremos duas coleções e imprimimos na tela todas as fotos que encontramos. O problema dessa solução é que ela é baseada em efeitos colaterais: ela afeta a tela diretamente, ou seja, o `println` é uma forma de efeito colateral, de mutabilidade.

Uma forma mais elegante de resolver isso seria gerar uma coleção de fotos, com todas as fotos de todas as tags, e só então imprimir o resultado na tela. Assim, o código será mais funcional e menos imperativo, o que traz muitos benefícios, como discutimos anteriormente.

Vamos começar refatorando o exemplo anterior sem *for comprehensions*. Para isso, precisamos usar `map` e `flatMap`:

```
val fotos = tags.flatMap(tag => caller.buscaFotos(tag).map(_.title))
```

```
fotos foreach println
```

Desta vez, estamos fazendo algo a mais: estamos transformando o conjunto de fotos em um conjunto de `String` `s` com o campo `title` das fotos. Embora seja uma linha simples e curta, a sua legibilidade é discutível, pois temos muita coisa acontecendo de uma só vez.

A versão deste código com *for comprehensions* seria a seguinte:

```
val fotos = for {  
  tag <- tags  
  foto <- caller.buscaFotos(tag)  
} yield foto.title
```

```
fotos foreach println
```

O ponto-chave aqui, que faz o *for comprehension* usar `map` e `flatMap` em vez de `foreach`, é a palavra-chave `yield`. Ela diz

ao compilador Scala que o que queremos na verdade é fazer uma série de transformações em uma coleção e gerar um novo resultado. Assim como com o `foreach`, aqui não temos nada mais do que açúcar sintático.

No caso, transformamos um conjunto de tags em um conjunto de fotos, e este último em um conjunto de `String`s. Lembrando de que o tipo da primeira coleção envolvida na operação será respeitado (`Set` neste caso). Isso quer dizer que a nossa coleção resultante será outro `Set`, ou alguma coleção compatível.

Na prática, isso significa que, se tivermos fotos com nomes iguais, as duplicatas vão simplesmente desaparecer no resultado final — exatamente o comportamento esperado para `Set`s.

Os exemplos que vimos nesta seção e na anterior usaram apenas duas coleções no `for`, `tags` e `fotos`, mas poderíamos ter usado quantas quiséssemos. Portanto, se o ganho de legibilidade não tiver parecido muito significativo, imagine se estivéssemos percorrendo cinco coleções, ou dez, uma depois da outra.

Um outro recurso interessante de *for comprehensions* está ligado ao uso de filtros em coleções. Vejamos mais um exemplo, primeiro sem açúcar sintático:

```
val nomesFiltrados = tags.filter(_.startsWith("j"))
    .flatMap(tag => caller.buscaFotos(tag).map(_.title))
```

```
nomesFiltrados foreach println
```

E novamente podemos tornar esse código mais legível por meio do uso de *for comprehensions*. O código a seguir gera o mesmo resultado do código anterior:

```
val nomesFiltradosFor = for {
  tag <- tags if tag.startsWith("j")
  foto <- caller.buscaFotos(tag)
} yield foto.title
```

```
nomesFiltradosFor foreach println
```

Note que, no `if` dentro do `for`, nós não precisamos usar parênteses. Usá-los não seria errado, mas é desnecessário.

Um ponto interessante, e que devemos sempre ter em mente quando trabalhamos com `for comprehension`s, é o que acontece quando algum dos elementos que estamos percorrendo está vazio. O comportamento com coleções normais, como vimos nos exemplos anteriores, acaba sendo intuitivo. O comportamento com outros tipos de elementos pode surpreender um pouco mais, em um primeiro momento.

Vamos detalhar um pouco mais isso na seção a seguir, ao explorarmos *Monads*. Especificamente investigaremos `Option`s, mas lembre-se de que o conceito se aplica a qualquer tipo de *Monad*, incluindo `Future`s. E na verdade, incluindo o que já vimos neste capítulo, pois as coleções `Scala` também são *Monads*.

9.3 O SEGREDO DO FOR: MONADS

Não vamos nos aprofundar em *Monads* neste livro, mas vamos pelo menos desmistificar o termo um pouco, pois já usamos esse tipo de elemento diversas vezes até aqui, tanto com coleções quanto com `Option`s. Simplificando, o simples fato de esses elementos possuírem os métodos `map`, `flatMap` e `foreach` os tornam *Monads* e, portanto, utilizáveis em *for comprehensions*. Vamos entender a seguir um pouco do porquê disso ser interessante.

Se antes usamos `for`s para percorrer coleções, agora vamos fazer o mesmo com `Option`s. Lembre-se do pensamento sobre `Option`s que mencionamos antes: de certa forma, um `Option` nada mais é do que uma coleção que pode receber apenas um elemento, ou então estar vazio.

Dentro do nosso método `buscaFotos`, poderíamos ter a seguinte linha:

```
val tagText = tag.map("&tags=" + _).getOrElse("")
```

Vamos reescrevê-la usando *for comprehensions*:

```
val tagText = for {  
  t <- tag  
} yield "&tags=" + t
```

Não ganhamos muita coisa neste caso, e `tagText` continua sendo um `Option`. Como aconteceu no uso de *for comprehensions* com coleções, o tipo resultante do `for` é sempre compatível com o tipo utilizando dentro dele. Esse é um exemplo no qual usar `for` tem uma legibilidade um pouco pior do que trabalhar com `map` diretamente.

Agora vamos imaginar um caso mais interessante e um pouco mais complexo: percorrer uma série de `Option`s em vez de apenas um. Primeiro, vamos criar nossos `Option`s:

```
val userOpt = Option("jcranky")  
val passOpt = Option("1234")
```

Vamos supor também um método que receba usuário e senha, e retorna um `Option` com um token de autenticação, caso as informações recebidas estejam corretas:

```
def autentica(user: String, pass: String): Option[String] =  
  if (user == "jcranky" && pass == "1234") Some("token")  
  else None
```

Para completar, vamos juntar tudo isso:

```
val tokenOpt = for {  
  user <- userOpt  
  pass <- passOpt  
  token <- autentica(user, pass)  
} yield token
```

O mais interessante nesse código é que, assim que um `None` `for`

encontrado, seja no `userOpt` ou no `passOpt`, ele será usado como resultado do `for` — exatamente como aconteceria com uma série de `map`s e `flatMap`s. O único caso que gerará um `Some(token)` é um `userOpt` e um `passOpt` com valores válidos.

Para encerrar este capítulo, um exercício para o leitor: um pouco sobre a classe `Future`, que mencionamos brevemente no final da seção anterior. `Future`s é um assunto que mereceria um livro por si só, mas em Scala eles possuem uma característica importante para o que estamos discutindo: eles são *monádicos*, assim como os `Option`s e as coleções da linguagem.

Fica como um exercício para o leitor procurar exemplos de uso de `Future`s Scala, com e sem *for comprehensions*, e comparar com o que vimos neste capítulo.

CLASSES ABSTRATAS E TRAITS

Tendo nos aprofundado em programação funcional, vamos agora dar um passo para trás e estudar um pouco alguns elementos um pouco mais avançados, mas mais comuns em linguagens de programação orientada a objetos — lembrando de que Scala é tanto uma linguagem funcional quanto orientada a objetos.

10.1 CLASSES ABSTRATAS

Toda linguagem de programação que suporte Orientação a Objetos também suporta (ou deveria!) elementos abstratos. Esses são elementos *incompletos*, feitos para serem estendidos e completados em outro momento.

Em Java, estaríamos falando de *classes abstratas* e *interfaces*. Em Scala, vamos começar estudando as *classes abstratas* e, em seguida, examinar as *trait* s. Em um primeiro momento, podemos entender *trait* s como sendo o equivalente às interfaces do Java. Mas, como veremos, são na verdade muito mais poderosas.

JAVA 8

A partir da versão 8, a linguagem Java adicionou às interfaces o suporte a alguns recursos similares a alguns dos recursos suportados pelas *traits* do Scala. Em especial, interfaces Java agora suportam *default methods*, ou seja, métodos com uma implementação padrão.

Uma *classe abstrata* é muito próxima a uma classe normal (ou concreta), com uma diferença importante: ela pode conter elementos não implementados (abstratos), que deverão ser definidos pelas *classes filhas* — a não ser que essas classes filhas também sejam abstratas.

Vamos criar uma *classe abstrata*, que vai representar o ato de parsear uma resposta a uma requisição do Flickr:

```
abstract class ResponseParser {  
  def parse(str: String): Set[Foto]  
}
```

Nosso `ResponseParser` é uma classe responsável por parsear a resposta de uma pesquisa de fotos no Flickr e nos devolver as fotos encontradas. O Flickr, porém, pode devolver a resposta tanto em formato XML quanto em formato JSON; logo, o processo de parsing necessário vai variar. Estamos então desenhando nosso código para suportar diferentes implementações para o processo de parsing dessa resposta.

Neste exemplo, o método `parse` é abstrato. Não é necessário indicar isto explicitamente, pois Scala infere este fato simplesmente por o método não possuir um corpo definido. Já para a classe, a indicação de que se trata de uma *classe abstrata* é obrigatória.

Se declararmos apenas `class` em vez de `abstract class` no exemplo anterior, o código não compilaria, com um erro parecido com o seguinte, em que o compilador nos avisa que se trata de uma *classe não abstrata* que possui um elemento *abstrato*:

```
Error:(6, 8) class ResponseParser needs to be abstract, since method parse is not defined
class ResponseParser {
  ^
```

Um exemplo simples de *classe filha* para o nosso parser seria:

```
class XMLParser extends ResponseParser {
  def parse(str: String): Set[Foto] = ???
}
```

Estamos usando o `???` novamente para simplificar. Veremos uma implementação completa deste método no capítulo a seguir. O importante é que a assinatura do método é exatamente a mesma definida na superclasse e que usamos a palavra-chave `extends` para indicar a herança. Poderíamos também criar a versão `Json` da classe:

```
class JsonParser extends ResponseParser {
  def parse(str: String): Set[Foto] = ???
}
```

Uma limitação importante (mas que não afeta nosso exemplo) é que não podemos herdar de mais do que uma classe. Isto é, `Scala` não suporta *herança múltipla*, ao contrário de `C++` e igual a `Java`. `Scala`, no entanto, oferece uma alternativa que fica no meio do caminho entre linguagens que suportam e linguagens que não suportam *herança múltipla*: `traits`. Para quem conhece `Ruby`, seria algo similar aos *mixins* dessa linguagem. Vamos abordar `traits` a seguir.

10.2 TRAITS

A primeira forma de pensar em `trait` s é como interfaces Java. Elas representam contratos de funcionalidades que devem ser obedecidos. Vamos reescrever o `ResponseParser` da seção anterior para torná-lo uma `trait` em vez de uma *classe abstrata*:

```
trait ResponseParser {  
  def parse(str: String): Set[Foto]  
}
```

O código em si não mudou muito, mas existe uma diferença muito importante: enquanto não podemos herdar mais de uma classe, podemos implementar (ou *adicionar*) na nossa classe ou *subtrait* quantas outras `trait` s quisermos. Vamos criar uma `trait` para logar os acessos ao Flickr:

```
trait Logger {  
  def log(msg: String): Unit  
}
```

Nosso `logger` possui apenas um método abstrato, `log`. Agora podemos reescrever o `XmlParser` para usar essa `trait` e ter acesso à funcionalidade de logging:

```
class XmlParser extends ResponseParser with Logger {  
  def parse(str: String): Set[Foto] = ???  
}
```

Como pode ser visto no código anterior, mesmo o `ResponseParser` sendo agora uma `trait`, ainda usamos a palavra-chave `extends`. Essa é a regra: o primeiro elemento que herdamos sempre é definido com o `extends`, e ele pode ser tanto uma `trait` quanto uma *classe* ou *classe abstrata*. A partir daí, podemos incluir quantas `trait` s quisermos, usando a palavra-chave `with` — mas apenas `trait` s, não podemos mais usar classes neste ponto.

E para cada nova `trait` adicionada, precisamos usar o `with` novamente, como pode ser visto a seguir, agora com o `JsonParser`:

```
class JsonParser extends ResponseParser with Logger with Ordered[JsonParser] {
  def parse(str: String): Set[Foto] = ???
  def compare(that: JsonParser): Int = ???
}
```

Para o leitor curioso, fica a tarefa de investigar a `trait Ordered` utilizada no código anterior. No caso, ela não faz sentido para o nosso exemplo, então vamos nos concentrar no `XmlParser`, que não está usando essa `trait`.

E falando dessa classe, ainda temos um problema. O código ainda não compila, pois o método `log` é abstrato — ou seja, não foi implementado em lugar algum. Temos diversas formas de resolver isso. Começaremos pela mais simples, que é simplesmente colocar a implementação na `trait`, como no código a seguir:

```
trait Logger {
  def log(msg: String): Unit = println(msg)
}
```

Ou seja, `trait`s podem ter, além de métodos abstratos, métodos concretos. Como mencionado anteriormente, isso possibilita ter algo parecido com herança múltipla: podemos adicionar múltiplas `trait`s com diversos métodos concretos. Assim, em casos como o nosso `Logger`, podemos ter um bloco de funcionalidades prontas que simplesmente adicionamos às nossas classes, sem afetar a hierarquia original *de classes* e, portanto, sem impedir que herdemos de classes que realmente precisemos herdar.

Vamos melhorar um pouco o nosso exemplo pois, como o leitor talvez já tenha imaginado, ter um `Logger` amarrado a `println` no console não é uma boa prática. Primeiro, vamos voltar ao `Logger` abstrato visto antes, e vamos criar uma `trait` específica para a implementação que faz o `println` no console:

```
trait Logger {
  def log(msg: String): Unit
}
```

```
trait ConsoleLogger extends Logger {
  def log(msg: String): Unit = println(msg)
}
```

No código anterior vimos que, para uma `trait` herdar de outra, também usamos o `extends`. Vale a mesma regra que vimos com classes: primeiro usamos o `extends` e, em seguida, um `with` para cada `trait` extra que quisermos adicionar.

Para usar a implementação `ConsoleLogger`, poderíamos simplesmente adicioná-la ao `XmlParser` ou ao `JsonParser` com um `with`. Mas vamos aproveitar a oportunidade para ilustrar outro recurso muito interessante do Scala: a possibilidade de adicionar `trait`s a objetos no ato da sua instanciação.

No exemplo a seguir, estamos usando o `XmlParser` que usa a `trait Logger` abstrata. Para isso funcionar, precisamos marcar o `XmlParser` como abstrato, como a seguir:

```
abstract class XmlParser extends ResponseParser with Logger {
  def parse(str: String): Set[Foto] = ???
}
```

Na prática, estamos dizendo ao compilador que o `XmlParser` precisa conhecer algum tipo de `Logger` quando for instanciado. Mas não estamos definindo que `Logger` é esse. Em condições normais, não seria possível criar objetos dessa classe, mas podemos resolver isso da seguinte forma, adiando a decisão para o ponto exato da criação do objeto:

```
val parser = new JsonParser with ConsoleLogger
```

Como dissemos antes, no momento da criação do objeto é que estamos decidimos qual implementação de `Logger` queremos usar.

Para fechar essa sessão, um pequeno aviso: a "pseudo" *herança múltipla* permitida com o uso de `trait`s pode trazer alguns problemas. Tente imaginar quais e voltaremos a este tópico no final

do capítulo.

10.3 CLASSES SELADAS

Um recurso muito poderoso ligado a classes, `trait`s e herança que temos disponível em Scala são as *classes seladas* — ou *traits seladas* quando estivermos trabalhando com `trait`s. Esse recurso nos permite restringir a herança permitida de determinadas classes, de forma que os usuários das nossas APIs tenham menos chances de criar hierarquias de classes inválidas ou sem sentido

Ou seja, ajuda a evitar que alguém herde de uma classe ou `trait` que não foi desenhada para ser herdada. E assim também ficamos menos amarrados quando precisarmos realizar manutenção nas nossas classes herdáveis. Podemos usar a herança de forma bem mais controlada, diminuindo os riscos ligados ao uso desse recurso.

PROBLEMAS DA HERANÇA

Um problema típico no uso de herança é que uma vez que uma classe é disponibilizada publicamente, qualquer alteração nela pode causar grandes problemas, pois pode afetar usuários que a tenham usado em seu código e herdado dela. Existem muitas discussões sobre o *uso de herança versus composição* e o leitor é encorajado a pesquisar sobre o assunto.

Uma forma de impedir esse problema é usado a palavra-chave `final`, porém isso impede que nós mesmos usemos herança internamente, de forma controlada. Com classes seladas, temos um meio-termo: podemos herdar das nossas próprias classes, e ainda assim impedir que outros usuários do nosso código façam isso.

Nas classes que criamos em um outro capítulo para representar os tipos de mídias tratadas pelo Flickr, tínhamos algo como o seguinte:

```
abstract class Media(val value: String)

object Fotos extends Media("fotos")
object Videos extends Media("videos")
object Todas extends Media("all")
```

Uma pergunta que precisamos nos fazer é: seria interessante que o usuário da nossa API fosse capaz de criar novos filhos da classe `Media` ? Em alguns cenários, talvez isso faça sentido, mas neste caso vamos argumentar que, além dos três tipos anteriores, o Flickr não suporta nenhuma outra opção.

Além disso, se o Flickr vier a acrescentar novos tipos de mídia, provavelmente vamos querer adicionar suporte a essas mídias manualmente na API, em vez de permitir que o usuário faça isso ele mesmo. Assim podemos tomar todos os cuidados necessários e suportar o novo formato de forma correta.

Dito isso, o custo de impedir novos filhos dessa classe por parte do usuário não pode ser impedir que nós mesmos criemos esses filhos. *Classes seladas* nos oferecem exatamente a restrição que precisamos. Na prática, o que as *classes seladas* restringem é a localização dos filhos de uma determinada classe: tudo deve estar no mesmo arquivo fonte, ou seja, no mesmo arquivo `.scala` .

Vamos, então, no código seguinte, proteger a classe `Media` :

```
sealed abstract class Media(val value: String)

object Fotos extends Media("fotos")
object Videos extends Media("videos")
object Todas extends Media("all")
```

Pronto. A partir deste momento, todos os filhos da classe `Media` devem ser criados no mesmo arquivo onde a própria classe

foi escrita, portanto os usuários da API não conseguirão herdar dela. Se tentarmos criar uma classe herdando de `Media` em um outro arquivo como a seguir:

```
object Musicas extends Musicas("musicas")
```

Teremos um erro como o seguinte:

```
Error:(40, 26) illegal inheritance from sealed class Media
  object Musicas extends Media("musicas")
                        ^
```

SEALED TRAIT

Se a base da sua hierarquia de classes for uma `trait`, não se preocupe: a palavra-chave `sealed` se aplica exatamente da mesma forma, com as mesmas restrições. Todos os implementadores de tal `trait` deverão ser definidos no mesmo arquivo da `trait` em si.

Um exemplo de uso desse recurso é a classe `Option` da API padrão do Scala. Como já estudamos antes, os únicos filhos válidos para a classe `Option` são `Some` e `None`. Logo, não faria sentido algum permitir novas heranças a partir de `Option`.

Se investigarmos o código-fonte da linguagem, veremos que esses elementos estão todos definidos no arquivo `Option.scala`. Vejamos suas assinaturas:

```
sealed abstract class Option[+A] extends Product with Serializable

final case class Some[+A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

A classe `Option` é parecida com a nossa classe `Media`, e o objeto `None` é parecido com nossos tipos de `Media`s. Já a classe

`Some` tem um cuidado extra: ela é definida como `final`, para evitar heranças a partir dela — e assim completando a segurança que estamos buscando nesse código.

No caso dos *objects*, eles não são `final`, pois já não podemos herdar deles de qualquer forma. Eles são literalmente objetos, e não *classes* ou *trait s*.

FILHOS DE CLASSES SELADAS DEVEM SER FINAL

É muito importante definir todos os filhos de determinado elemento selado como `final`, pois o `sealed` apenas restringe a herança dos elementos marcados, e não da hierarquia toda. No caso da API de `Option s` do Scala, se `Some` não fosse `final`, por exemplo, seria possível para qualquer um criar novos filhos de `Some` e, assim, quebrar a proteção que ganhamos com o `sealed`.

10.4 HERANÇA MÚLTIPLA E O PROBLEMA DO DIAMANTE

Como mencionamos antes, Scala não suporta herança múltipla, mas permite algo bem parecido com o uso de *trait s*. Na verdade, esse recurso é tão parecido com herança múltipla que traz também um problema em comum: o *problema do diamante*.

Para entender esse problema, vejamos a seguinte hierarquia de classes. Teremos uma *trait base* `LogBase` definindo apenas um método, `log(msg: String): Unit`, como a seguir:

```
trait LogBase {  
  def log(msg: String): Unit  
}
```

Até aqui, nada demais. Vamos agora definir dois filhos para essa `trait` : `LogArquivo` e `LogConsole` . Eles não vão fazer nada mais do que implementar o método definido pelo pai:

```
trait LogArquivo extends LogBase {  
    override def log(msg: String) = println(s"logando $msg no arquivo")  
}  
  
trait LogConsole extends LogBase {  
    override def log(msg: String) = println(s"logando $msg no console")  
}
```

Os dois filhos definidos são também `trait` s. No código a seguir, vamos usar uma delas, apenas para ilustrar. Como estamos trabalhando com `trait` s, precisaremos criar uma classe anônima que a implementa — o que é bem simples de se fazer, já que `LogArquivo` não possui nenhum membro abstrato:

```
new LogArquivo{}.log("olá trait!")
```

OVERRIDE DE MÉTODOS ABSTRATOS

Repare nos exemplos que, quando implementamos o método abstrato da `trait` base `LogBase` , nós utilizamos a palavra-chave `override` . Em Scala, quando sobrescrevemos métodos concretos, somos obrigados a usar essa palavra-chave. Quando implementamos métodos abstratos, não.

Mesmo assim, é uma boa prática utilizar esse recurso. Assim, se a classe pai mudar o nome do método, por exemplo, o compilador nos avisará que não estamos sobrescrevendo ou implementando nada, ou seja, o código não compilará mais.

Vamos agora ao ponto principal do problema. O que acontece

se criarmos uma classe (ou mesmo `trait`) que implementa as duas `trait` s anteriores e invocarmos o método `log` ? Qual implementação será invocada, a da `trait LogArquivo` ou a da `trait LogConsole` ? Vejamos um exemplo:

```
val logger = new LogArquivo with LogConsole
logger.log("importante!")
```

O resultado será o seguinte:

logando importante! no console

Ou seja, o método da última `trait` mencionada na criação do objeto é o invocado. Podemos confirmar isso invertendo as `trait` s na criação do objeto e verificando o resultado novamente:

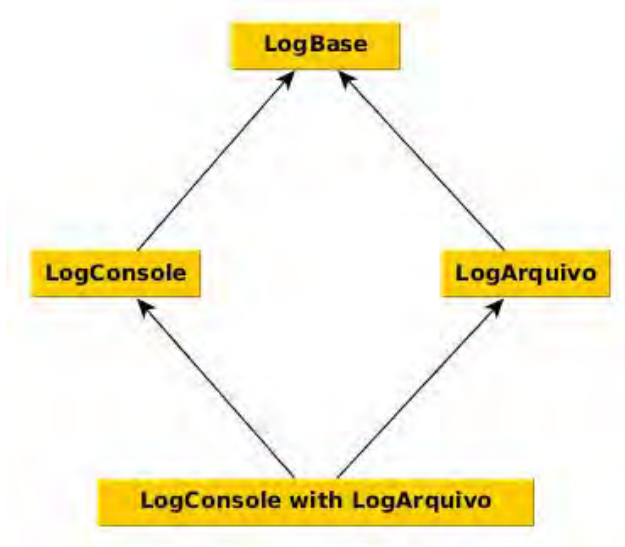
```
val logger = new LogConsole with LogArquivo
logger.log("importante!")
```

E o resultado será:

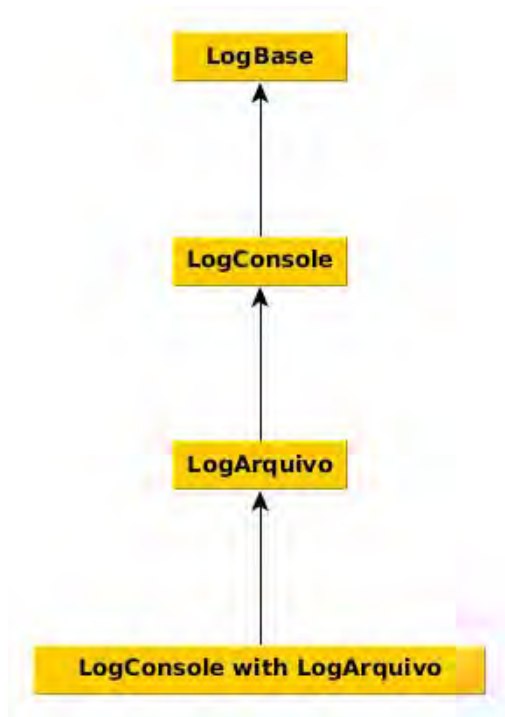
logando importante! no arquivo

Ou seja, apesar de nossa hierarquia de classes ter gerado um diamante (daí o nome "problema do diamante"), o compilador Scala teve de escolher qual método vai realmente ser executado. Geralmente, essa escolha será da direita para a esquerda, na ordem de declaração das `traits`, podendo variar em casos mais complexos.

O diagrama a seguir ilustra melhor a hierarquia e o diamante que acabamos de criar:



O processo usado para resolver esse problema é geralmente chamado de *linearização*, e o resultado seria algo como o ilustrado no diagrama a seguir, para o último exemplo de código. As traits `LogArquivo` e `LogConsole` estariam em posições invertidas para o exemplo anterior.



Em um primeiro momento, talvez não pareça muito importante entender esse processo de linearização. Afinal, tirando o caso de sobrescrita de métodos com a mesma assinatura, tudo funciona bem, sem muito esforço.

Porém, existe um caso que tornam as coisas um pouco mais complicadas e que exige um mínimo de entendimento desse processo: o uso do `super` para acessar membros de classes mãe. No nosso exemplo anterior, poderíamos querer que múltiplas classes de logging acumulassem a tarefa de logar — e não substituí-la.

Em Orientação a Objetos, fazemos isso invocando o método da *superclasse* de dentro da nossa *subclasse*. Vamos fazer isso então:

```
trait LogArquivo extends LogBase {
```

```

    override def log(msg: String) = {
      super.log(msg)
      println(s"logando $msg no arquivo")
    }
  }

  trait LogConsole extends LogBase {
    override def log(msg: String) = {
      super.log(msg)
      println(s"logando $msg no console")
    }
  }

```

Tudo parece bem, mas na verdade temos um problema aqui. Ao compilar o código anterior, teremos os seguintes erros:

```

<console>:10: error: method log in trait LogBase is accessed from
super. It may not be
  abstract unless it is overridden by a member declared `abstract'
and `override'
      super.log(msg)

<console>:11: error: method log in trait LogBase is accessed from
super. It may not be
  abstract unless it is overridden by a member declared `abstract'
and `override'
      super.log(msg)

```

Ou seja, estamos tentando acessar um método no `super` que é abstrato. Apenas com o código anterior, é impossível para o compilador saber qual é a implementação a ser invocada. A correção do problema é bem simples e já é mencionada na mensagem de erro. Vamos apenas entender o que está acontecendo.

Quando invocamos o `super`, o que queremos é invocar a implementação de determinado método na superclasse. Porém, no nosso exemplo, quem é essa implementação?

Não é possível determinar isso, pois o único supertipo de nossas `trait` s de logging é o `LogBase`, cuja implementação do método que queremos também é abstrata. Isso gera um erro que pode ser corrigido como explicado na própria mensagem de erro.

Na prática, o que temos de fazer é avisar ao compilador que sabemos que estamos sobrescrevendo um método abstrato — e que um elemento concreto é esperado e será definido em outro ponto do código. Façamos isso então:

```
trait LogArquivo extends LogBase {  
  abstract override def log(msg: String) = {  
    super.log(msg)  
    println(s"logando $msg no arquivo")  
  }  
}  
  
trait LogConsole extends LogBase {  
  abstract override def log(msg: String) = {  
    super.log(msg)  
    println(s"logando $msg no console")  
  }  
}
```

E pronto, agora tudo compila. Porém, na hora de testar, ainda temos um problema, como podemos ver a seguir:

```
val logger = new LogConsole with LogArquivo
```

E o resultado seria:

```
<console>:10: error: object creation impossible, since method log  
in trait LogArquivo of  
type (msg: String)Unit is marked `abstract' and `override' and ov  
errides incomplete  
  superclass member method log in trait LogConsole of type (msg: S  
tring)Unit  
    val logger = new LogConsole with LogArquivo
```

O que está faltando agora é a garantia de que um elemento concreto existe. Todos os métodos `log` até aqui são abstratos, mesmo os que possuem algum corpo — os `abstract override s`. Vamos criar uma nova classe que cria uma implementação vazia, mas concreta:

```
class EmptyLogger extends LogBase {  
  override def log(msg: String): Unit = {}  
}
```


Com isso feito, agora podemos finalizar o exemplo:

```
val logger = new EmptyLogger with LogConsole with LogArquivo
logger.log("Olá Scaladores!")
```

Tivemos de começar a criação do objeto com o tipo concreto, e depois adicionamos os tipos abstratos. Essa ordem é importante, pois precisamos fazer com que o método concreto fique no topo da hierarquia linearizada. Feito isso, vamos testar esse logger seguinte, e ver o que acontece:

```
logando Olá Scaladores! no console
logando Olá Scaladores! no arquivo
```

Agora tudo funciona: resolvemos o problema e, ao mesmo tempo, ganhamos a habilidade de logar facilmente em dois lugares ao mesmo tempo.

PARSEANDO XML

Mencionamos algumas vezes que o Flickr pode retornar resultados tanto em JSON quanto em XML. Vamos neste capítulo abordar um pouco sobre o suporte a XML do Scala, com foco em entender como uma resposta a uma requisição ao Flickr pode ser transformada em objetos Scala.

Portanto, veremos apenas como ler XMLs, e não como gerá-los. Isso porque, além de não ser algo muito comumente necessário, é algo bem mais complexo de se fazer com essa API e, portanto, está fora do escopo deste livro. O leitor curioso, no entanto, pode encontrar muitas referências a respeito na internet.

11.1 O BÁSICO DE XML EM SCALA

O suporte a XML da linguagem Scala é um recurso controverso. Isso porque Scala suporta XML de forma que faz parecer algo nativo da linguagem. Isso é feito através de uma API disponível por padrão até a versão 2.10 da linguagem, mas que foi modularizada (e removida da distribuição padrão) a partir do Scala versão 2.11.

BIBLIOTECA XML MODULARIZADA

Neste ponto, é importante o leitor verificar se a biblioteca XML do Scala está no seu classpath, para que seja possível compilar e executar os exemplos deste capítulo. Se estiver usando o `sbt`, basta adicionar algo como `"org.scala-lang.modules" % "scala-xml_2.11" % "1.0.5"` na lista de dependências do projeto.

Já no REPL, será necessário baixar o `jar` da biblioteca e passá-lo manualmente no início da sessão. Algo como `scala -classpath scala-xml.jar` no terminal de comandos, com o caminho completo para o local onde a biblioteca foi baixada.

Em Scala, podemos escrever trechos de código XML como parte do código-fonte em si. Vejamos um exemplo a seguir:

```
val fotos =  
  <fotos>  
    <foto>uma foto</foto>  
    <foto>outra foto</foto>  
  </fotos>
```

A variável `fotos` no exemplo anterior é do tipo `scala.xml.Elem`. Essa classe é filha de `Node`, que por sua vez herda de `NodeSeq`. Como dizem seus nomes, essas classes representam nós e sequências de nós, respectivamente.

A hierarquia dessas classes pode muitas vezes parecer confusa — e de fato em alguns casos isso é verdade, e não vamos estudá-la profundamente. Vamos focar no que precisamos saber para implementar nosso `XMLParser` de forma a parsear as respostas do Flickr e gerar as instâncias da classe `Foto`.

Para navegar pelas tags XML, podemos usar diversos métodos oferecidos pela classe `NodeSeq`. Muitos desses métodos lembram a sintaxe de *XPath* — mas apenas lembram, não é um mapeamento formal dessa linguagem. Mesmo assim, quem conhece *XPath* vai se sentir mais à vontade aqui. Podemos obter uma lista com todos os corpos de texto das tags `<foto>` da seguinte forma:

```
fotos \ "foto"
```

A `\` é um método que encontra todos os nós com determinado nome dentro da *tag atual*, apenas no nível atual ou *raiz* do elemento XML sendo trabalhado, sem pesquisar subtags. Podemos usar o método `\\` para pesquisar tags em qualquer nível, ou seja, na tag raiz e também em todas as subtags presentes em qualquer nível do elemento XML. O resultado é um `NodeSeq` com duas tags `<foto>`:

```
res5: scala.xml.NodeSeq = NodeSeq(<foto>uma foto</foto>, <foto>outra foto</foto>)
```

O `NodeSeq` inclui (estende) a `trait Seq` e, assim, pode ser tratado como qualquer outra sequência da API de coleções do Scala. Vamos então usar um `map` para obter todos os nós de texto, de todas as tags `<foto>`:

```
(fotos \ "foto").map(_.text)
```

O `text` é um método que retorna o texto do nó atual. O resultado do código anterior será uma lista com o corpo textual de todos os nós encontrados. Algo como o exemplo a seguir:

```
res6: scala.collection.immutable.Seq[String] = List(uma foto, outra foto)
```

11.2 PARSEANDO A RESPOSTA XML DO FLICKR

A seguir, temos um exemplo extraído de uma resposta real de uma busca por fotos com a tag scala, executada no momento da escrita desta sessão. Vamos usá-lo para implementar nosso parser.

```
<rsp stat="ok">
  <photos page="1" pages="5573" perpage="10" total="55728">
    <photo id="14204168540" owner="122040014@N07" secret="f19c546f
fa" server="3871" farm="4"
      title="Scala, Salerno, Italy" ispublic="1" isfriend="0" isfa
mily="0" />
    <photo id="14406205613" owner="59634982@N05" secret="1073a5b91
7" server="3871" farm="4"
      title="La Scala" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="14197741819" owner="24271543@N03" secret="850b867db
0" server="3847" farm="4"
      title="Pomarez, Landes: décoration par Ydan Sarciat sur les
arènes."
      ispublic="1" isfriend="0" isfamily="0" />
    <photo id="14382256984" owner="34527570@N05" secret="c77830c9f
4" server="2900" farm="3"
      title="Vertigo" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="14380318122" owner="68397559@N00" secret="46fb211db
1" server="5506" farm="6"
      title="Teatro alla Scala" ispublic="1" isfriend="0" isfamily=
"0" />
    <photo id="14194771350" owner="108099607@N06" secret="72d20104
b1" server="3926" farm="4"
      title="IMG_0228" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="14189676529" owner="77913019@N06" secret="99a0b200d
c" server="3917" farm="4"
      title="Milan - Night" ispublic="1" isfriend="0" isfamily="0"
/>
    <photo id="14373635404" owner="36117086@N05" secret="117c612f6
9" server="5577" farm="6"
      title="Samsung Motors 3" ispublic="1" isfriend="0" isfamily=
"0" />
    <photo id="14351480646" owner="36117086@N05" secret="96581e18f
5" server="3857" farm="4"
      title="Samsung Motors 3" ispublic="1" isfriend="0" isfamily=
"0" />
    <photo id="14371242601" owner="36117086@N05" secret="701e103e5
8" server="5543" farm="6"
      title="Samsung Motors 3" ispublic="1" isfriend="0" isfamily=
"0" />
  </photos>
</rsp>
```

A resposta do Flickr é paginada e temos 10 respostas por página. Podemos ver claramente que temos muito mais páginas disponíveis através do atributo `pages : 5573`. Vamos nos concentrar em parsear esta página pois, para as demais, bastaria efetuar novas consultas para cada página e invocar o método de parseamento novamente.

Para esse capítulo, vamos utilizar um método `buscaFotos` que receberá uma tag a ser usada na busca, e retornará uma sequência (`Seq`) de fotos. A assinatura será a seguinte:

```
def buscaFotos(tag: Option[String]): Seq[Foto]
```

Ou seja, vamos retornar uma lista de objetos `Foto`. Esse método, por sua vez, usará o `XMLParser` para gerar essa lista a partir da `String` (que por acaso é um documento XML) de resposta do Flickr. Portanto, a última linha desse método agora será:

```
new XMLParser().parse(Source.fromURL(url).mkString)
```

Essa não é necessariamente a forma mais eficiente de se fazer a leitura da resposta, pois estamos lendo-a toda de uma vez, para só então parseá-la. Mas como uma página de resposta é relativamente pequena, esse código é o suficiente para nosso exemplo.

Outro ponto que talvez o leitor queira mudar em um exemplo real é o `new Parser()`, para reutilizar um parser em vez de criar um novo a cada operação de parsing. Novamente, isso é o suficiente para nosso exemplo, mas pode não ser em outros casos.

No código anterior, criamos o parser, lemos a resposta do Flickr e passamos essa resposta como uma `String` para esse parser. Para implementar o parseamento, a primeira coisa que temos de fazer é transformar a `String` em um objeto XML, como no código a seguir:

```
import scala.xml.XML
```

```
val xmlResp = XML.loadString(str)
```

XML é um objeto com diversos métodos interessantes, entre eles o `loadString` usado no código anterior. Ele recebe uma `String` e retorna um objeto `Elem` com o documento XML equivalente à `String` especificada. Para ler as fotos, usamos o mesmo mecanismo que estudamos na sessão anterior, obtendo a lista com as tags `photo` e, em seguida, lendo os atributos que nos interessa:

```
xmlResp \\ "photo" map { p =>
  Foto(
    (p \ "@id" text).toLong,
    p \ "@owner" text,
    (p \ "@server" text).toInt,
    p \ "@title" text)
}
```

Por questão de simplicidade, estamos ignorando as *tags*, pois elas não estão disponíveis na listagem de fotos. Precisariamos fazer uma busca por detalhes de cada foto e obter as tags a partir de lá. Dessa vez, para obter a lista de fotos, estamos usando o método `\\`, que ignora em qual nível do XML a tag se encontra. Poderíamos também ter feito `xmlResp \ "photos" \ photo`.

Em seguida, para cada foto, obtemos o atributo correspondente ao campo no construtor da foto. O `@` indica que queremos ler um atributo em vez de uma tag. O método `text` faz a mesma coisa que fizemos no caso das tags: obtém o texto do atributo.

No caso dos atributos `id` e `server`, que são `Long` e `Int` respectivamente, temos de transformar o texto no valor numérico correspondente. Fazemos isso utilizando o `toLong` e o `toInt`, que são métodos disponíveis implicitamente em todas as `String`s. Falaremos mais sobre *implicit*s no próximo capítulo.

A forma final do método `parse` é a seguinte:

```
override def parse(str: String): Seq[Foto] = {
  val xmlResp = XML.loadString(str)

  xmlResp \\ "photo" map { p =>
```

```

Foto(
    (p \ "@id" text).toLong,
    p \ "@owner" text,
    (p \ "@server" text).toInt,
    p \ "@title" text)
}
}

```

Uma outra funcionalidade da API de XML é alterar ou criar XMLs. Ou seja, a operação de escrita em contraste com a operação de leitura que vimos até aqui. Infelizmente, diferente da leitura, a escrita de XMLs com essa API não é algo tão simples de se fazer, e foge do escopo do livro. Alterar XMLs, portanto, fica como um exercício para o leitor que tiver estômago e interesse — ou necessidade real em algum projeto.

NAMESPACE BUG

No momento da escrita deste livro, a API de XMLs do Scala ainda possui um bug relacionado com o manuseio de namespaces `xml`. Quando estamos lidando com documentos XMLs simples, não há nenhum problema significativo. Mas ao ler e escrever documentos XMLs que lidem com múltiplos *namespaces*, muitas vezes a API é incapaz de mesclá-los corretamente, colocando dois (ou mais) *namespaces* no mesmo nível, gerando um documento XML inválido.

Esse é um bug que pode causar muita dor de cabeça, mas acontece em casos muito específicos. Neste caso, a solução é, ou usar outra API, ou corrigir os *namespaces* manualmente — isto é, removê-los da representação em *String* do documento.

IMPLICITS

Implicits são o toque Scala para a flexibilidade extra geralmente encontrada apenas em linguagens dinâmicas, como Python ou Ruby. Algumas dessas linguagens oferecem, por exemplo, um recurso conhecido como *Monkey Patch*: a capacidade de adicionar ou alterar funcionalidades diretamente em tipos existentes.

Esse tipo de recurso é extremamente poderoso, mas também muito perigoso, pois é fácil afetar código que não deveria, ou então causar efeitos difíceis de serem entendidos, principalmente em tempo de manutenção. Como veremos a seguir, diferente dessas linguagens, o que podemos fazer em Scala é bem mais controlado e seguro, e ainda assim bastante poderoso.

12.1 ADICIONANDO FUNCIONALIDADE A TIPOS EXISTENTES: CONVERSÕES IMPLÍCITAS

Começando exatamente na flexibilidade mencionada, vamos falar de conversões implícitas. Temos na verdade dois tipos de elementos implícitos em Scala: *conversões* e *variáveis*. Vamos abordar as conversões nesta seção, e ver variáveis na seguinte.

Em linguagens dinâmicas, geralmente é possível adicionar funcionalidades em classes e objetos já existentes. Seria como se pudéssemos adicionar novos métodos na classe `String` do Java,

por exemplo. Em Ruby e Python, isso é conhecido como *Monkey Patch*. Porém, isso é muito específico de linguagens dinâmicas, pois depende muito da tipagem dinâmica dos objetos.

Voltando para Scala, e lembrando de que estamos na JVM (onde esse tipo de coisa é mais complicado), os `ClassLoader`s nos protegem desse tipo de alteração. Isso é uma coisa boa, pois muitos problemas podem aparecer de forma inexplicável com esse tipo de recurso. Apenas imagine quantas bibliotecas não seriam afetadas por uma alteração na classe `String`.

Lembre-se também de que estamos falando da linguagem Scala, uma linguagem fortemente tipada. A ideia, com Scala, é sempre tentarmos ter a maior quantidade possível de informação sobre nossas classes e objetos em tempo de *compilação*, possibilitando ao compilador nos ajudar em muito a manter o código mais correto e seguro.

Mesmo com as desvantagens mencionadas, poder acrescentar novas funcionalidades a elementos existentes é muito útil, e Scala permite isso por meio de *conversões implícitas*. Em suma, em vez de usar *monkey patch* e alterar uma classe dinamicamente, o que fazemos em Scala é criar uma nova classe que contém a classe original e adicionar as novas funcionalidades — ou seja, um *Wrapper*. Abra uma sessão do REPL e teste o código a seguir:

```
"99".toInt
```

O resultado será algo como:

```
res0: Int = 99
```

O que aconteceu aqui? Se olharmos o *JavaDoc* da classe `String`, não vamos encontrar nenhum método `toInt`. E Scala realmente usa a *String* do Java. Isto é, se procurarmos uma classe `String` no *scaladoc*, não encontraremos essa classe — existem

outros elementos com `String` em parte do nome, mas não a `String` propriamente dita. Mesmo assim, a chamada ao método `toInt` funcionou normalmente no código anterior.

Para entender o que está acontecendo, vamos dar uma olhada no object `Predef`, que já mencionamos antes. Especificamente, vejamos a implementação do método `wrapString`:

```
implicit def wrapString(s: String): WrappedString =  
  if (s ne null) new WrappedString(s) else null
```

NE

O `ne` (not equal) é uma forma alternativa para `!=` definida na classe `AnyRef` que, como já sabemos, é a base de todas as classes em Scala.

O elemento mais importante para a nossa discussão neste momento aparece logo de cara: a palavra-chave `implicit`. Repare também no tipo de retorno desse método: `WrappedString`.

Olhando o *ScalaDoc* da classe `WrappedString`, encontraremos o método `toInt` que surgiu "do nada" anteriormente. Lembrando de que os elementos definidos no object `Predef` estão automaticamente disponíveis para nosso código, o que está acontecendo é o seguinte:

1. O compilador encontra a `String` "99" e a tentativa de invocar um método que não existe em `String`;
2. O compilador começa a procurar *conversões implícitas* que possam transformar a `String` em algum outro objeto que possua o método `toInt`;
3. Essas conversões podem estar em qualquer lugar visível para o código sendo compilado;

4. O compilador encontra o método `wrapString` e invoca-o, passando a nossa `String` ;
5. O método `toInt` é invocado no objeto retornado pela função.

Ou seja, na prática, uma conversão implícita é um método qualquer que, quando marcado com a palavra-chave `implicit` , pode ser invocado automaticamente pelo compilador em determinadas situações. Para que esses passos funcionem corretamente, temos ainda de observar as regras a seguir:

- A função implícita a ser usada precisa receber um objeto do tipo exato a ser convertido;
- O tipo retornado pela função tem de possuir o método que precisamos.

Isso basicamente quer dizer que nunca acontecerá de o compilador Scala tentar converter um objeto para um tipo e, em seguida, converter o resultado para um terceiro tipo. A conversão é sempre direta.

A resolução de métodos implícitos já pode ser um processo lento em alguns casos. E se fosse possível fazer essas conversões indiretas, teríamos uma explosão de combinações que tornariam o processo inviavelmente lento.

Vejamos um exemplo agora com a nossa API Flickr. Vamos supor que queremos adicionar a capacidade de imprimir as fotos, mas não queremos (ou não podemos) alterar diretamente a classe `Foto` .

Esse tipo de cenário é bem comum: dificilmente podemos alterar classes de bibliotecas. Mas, usando `implicit` , temos uma forma alternativa de acrescentar funcionalidades a tais bibliotecas. Foi exatamente o que foi feito com `String` s anterior. Voltando ao

nosso exemplo, queremos poder fazer algo como:

```
Foto(1234, "jcranky", 1234, "Foto 1").print
```

Obviamente o código anterior não compilaria, pois `Foto` não possui um método chamado `print`. Para fazê-lo funcionar, vamos criar uma classe `wrapper` para a foto e implementar a funcionalidade de impressão, como a seguir:

```
class PrintableFoto(foto: Foto) {  
  def print = println("printando foto...")  
}
```

Por fim, só falta a conversão implícita. Se definirmos um método como a seguir, no escopo de onde queremos invocar o método `print`, já teremos um exemplo completo.

```
implicit def toPrintableFoto(foto: Foto): PrintableFoto = new Prin  
tableFoto(foto)
```

Pronto! Podemos definir esse método dentro da classe que quer imprimir a `Foto`, ou em qualquer outro escopo acessível pelo código em questão. Ou podemos usar qualquer objeto e importar o método, seja diretamente ou importando o objeto todo com `_` (underline).

A partir da versão 2.10 do Scala, existe uma forma alternativa e mais simples para se escrever algumas conversões implícitas. Em vez de declarar uma classe `wrapper` e um método implícito que faz a conversão, nós criamos apenas a classe e a declaramos `implicit`. No código a seguir, vamos reescrever o exemplo anterior, agora com `implicit class`:

```
implicit class PrintableFoto(foto: Foto) {  
  def print = println(s"printando foto [$foto]...")  
}  
  
Foto(1234, "jcranky", 1234, "Foto 1").print
```

Esse código funciona exatamente como o outro exemplo, porém

é um pouco mais enxuto. A principal desvantagem é que tal classe precisa ser definida dentro do escopo de outra classe ou objeto. Ou seja, ela não pode ser *top level*, não pode existir por si só. Na prática, isso quer dizer que *implicit classes* são mais interessantes para *wrappers* simples, com poucas linhas de código.

12.2 CONVERSÕES IMPLÍCITAS AMBÍGUAS

Logo que aprendemos a usar conversões implícitas, uma dúvida muito comum é: o que acontece se tivermos duas conversões implícitas disponíveis? A resposta é exatamente o que esperamos de uma linguagem forte e estaticamente tipada como Scala: um erro de compilação.

Assim, sempre podemos descobrir qual é exatamente a conversão implícita sendo aplicada, se necessário. Não haverá conflitos em tempo de execução. Vejamos um exemplo:

```
implicit def toPrintableFoto(foto: Foto): PrintableFoto = new PrintableFoto(foto)
implicit def newPrintableFoto(foto: Foto): PrintableFoto = new PrintableFoto(foto)
```

Neste caso, estamos definindo duas funções iguais, mudando apenas o nome. As duas funções têm o mesmo resultado também: transformam uma `Foto` em `PrintableFoto`. Como as duas são *implícitas*, ambas podem ser aplicadas quando o compilador Scala precisar executar essa transformação, e isso não é válido.

Porém, nenhum acontece aqui, pois as funções ainda não estão sendo usadas. Entretanto, se tentarmos executar qualquer código que precise da conversão de `Foto` para `PrintableFoto`, como a seguir:

```
Foto("foto1", "jcranky", 1234, "Foto 1", None).print
```

Teremos um erro como o seguinte:

```

<console>:13: error: type mismatch;
  found   : Foto
  required: ?{def print: ?}
Note that implicit conversions are not applicable because they are
ambiguous:
  both method toPrintableFoto of type (foto: Foto)PrintableFoto
  and method newPrintableFoto of type (foto: Foto)PrintableFoto
  are possible conversion functions from Foto to ?{def print: ?}
      Foto("foto1", "jcranky", 1234, "Foto 1", None).print
          ^
<console>:13: error: value print is not a member of Foto
      Foto("foto1", "jcranky", 1234, "Foto 1", None).print

```

A mensagem de erro começa avisando que o método que estamos tentando invocar na `Foto (print)` não existe. Em seguida, ele também nos avisa que não foi possível usar conversões implícitas, pois existem duas conversões disponíveis e elas são ambíguas, ou seja, o compilador não sabe qual usar.

E como mencionamos anteriormente, repare que o erro acontece apenas na tentativa de uso da conversão, e não na definição da função, e diz claramente qual é o problema: a conversão é ambígua. Isso é importante, pois o ponto de definição das funções e o ponto de uso delas são coisas distintas. Poderíamos, por exemplo, estar importando duas conversões do mesmo tipo e teríamos o mesmo erro.

Outro exemplo muito interessante do uso de conversões implícitas está no framework de testes *Specs2*. Usando esse recurso, o framework consegue nos oferecer funcionalidades bastante interessantes, como podemos ver no exemplo a seguir. Nele os métodos parecem surgir "do nada" em `String s`, nos permitindo escrever testes em um estilo mais comumente visto em linguagens dinâmicas:

```

"the xml parser" should {
  "turn the xml into the model class" in {
    val fotosXml =
      <rsp stat="ok">
        <photos>

```

```

        <photo id="123" owner="jcranky" server="6" title="jcranky test"></photo>
    </photos>
</rsp>

    new XMLParser().parse(fotosXml.toString()) must_== Seq(
        Foto(123, "jcranky", 6, "jcranky test"))
    }
}

```

O código anterior é possível porque a `trait Specification` adiciona à nossa classe de testes uma série de conversões implícitas que sabem transformar a `String` em questão em algum objeto interno do framework de testes *Specs2*, que possui os métodos que precisamos, como o `should` e o `in` deste exemplo.

12.3 PASSANDO PARÂMETROS SEM PASSAR NADA: PARÂMETROS IMPLÍCITOS

Outro recurso muito interessante da linguagem Scala são os *parâmetros implícitos*. Esse tipo de parâmetro é muito útil em cenários nos quais precisamos passar o mesmo parâmetro para diversas funções.

Em vez de ficar passando o parâmetro o tempo todo, nós declaramos o parâmetro como implícito apenas uma vez e, em seguida, o próprio compilador se encarregará de passar esse parâmetro sempre que encontrar uma função que receba um parâmetro implícito do mesmo tipo.

Uma das grandes vantagens disso é que, além de deixar o código um pouco mais "limpo", também fica bem mais fácil substituir o valor passado para esses vários parâmetros de uma única vez. Fica fácil, por exemplo, usar uma variável para testes, e outra para produção.

Vejamos um exemplo usando o *Anorm*, um framework de

persistência usado pelo Play Framework. O exemplo a seguir foi extraído da *Lojinha* (<http://github.com/jcranky/lojinha>), um projeto de código aberto em Scala que, como diz o nome, implementa uma pequena loja. Este código está apenas inserindo categorias de produtos no banco de dados.

```
def create(displayName: String, urlName: String) =
  DB.withConnection { implicit c =>
    SQL("INSERT INTO category(display_name, url_name) VALUES({display_name}, {url_name})").on(
      'displayName -> displayName, 'urlName -> urlName).executeUpdate()
  }
```

Perceba que, à primeira vista, pode parecer que o parâmetro `c` (que é um objeto da classe `Connection`) nunca é usado. Porém, ele é na verdade passado implicitamente ao método `executeUpdate`. Assim, não precisamos ficar poluindo nosso código em cada chamada que precisar da conexão com o banco de dados.

Para visualizar melhor o que está acontecendo, vejamos a assinatura do método `executeUpdate`:

```
def executeUpdate()(implicit connection: java.sql.Connection): Int
```

Esse método tem duas listas de parâmetros: a primeira é vazia; e a segunda recebe a `Connection`. Como o parâmetro `connection` está marcado como `implicit`, sempre que esse método for invocado, podemos omitir a segunda lista de parâmetros, desde que haja uma variável implícita do mesmo tipo disponível.

Podemos disponibilizar a variável implícita de duas formas. Uma delas é como no exemplo anterior, em que recebemos um parâmetro e o marcamos como implícito. Assim, além dele poder ser recebido implicitamente, esse parâmetro também pode ser repassado para outros métodos, também de forma implícita.

Outra forma é declarar diretamente uma variável, e adicionar o

marcador `implicit` a ela. O exemplo anterior poderia ser reescrito da seguinte forma, supondo que vamos construir a conexão com o banco de dados manualmente — ou talvez utilizar uma conexão falsa para testes:

```
implicit val c: Connection = ???

SQL("INSERT INTO category(display_name, url_name) VALUES({displayName}, {urlName})").on(
  'displayName -> displayName, 'urlName -> urlName).executeUpdate(
)
```

Só precisamos é claro trocar o `???` por um objeto `Connection` de verdade. Vamos agora ver um exemplo de uso de parâmetros implícitos em nossa API Flickr. Vamos começar reescrevendo o método `buscaFotos` para receber um parâmetro implícito:

```
def buscaFotosComImplicits(tag: Option[String])(
  implicit parser: ResponseParser): Seq[Foto] = {

  val method = "flickr.photos.search"
  val tagText = tag.map("&tags=" + _).getOrElse("")

  val params = s"?method=$method&per_page=10&api_key=$apiKey$tagText"
  val url =
    s"https://api.flickr.com/services/rest/$params"

  parser.parse(Source.fromURL(url).mkString)
}
```

O método que busca fotos no Flickr agora recebe duas listas de parâmetros, e a segunda delas é uma lista de *parâmetros implícitos*. É importante entender aqui que a segunda lista de parâmetros é inteira implícita, ou seja, todos os parâmetros que colocarmos nela serão implícitos.

A palavra-chave `implicit` se aplica à lista de parâmetros toda, e não aos parâmetros individuais. É por isso também que separamos os parâmetros no método do exemplo anterior em duas listas de

parâmetros, assim uma delas é normal, e a outra é implícita.

Para podermos invocar esse método, vamos criar uma variável implícita e invocá-la da mesma forma como faríamos com o `buscaFotos` original, como a seguir:

```
implicit val parser = new XMLParser()

val fotos = buscaFotosComImplicits(None)
fotos foreach println
```

O resultado será o mesmo obtido ao se invocar a versão anterior do método. Porém agora podemos substituir o `parser` a ser usado na sua execução simplesmente trocando o `implicit val` anterior por alguma outra implementação — uma que saiba ler JSON, por exemplo, ou quem sabe uma implementação falsa para testes.

12.4 COMO O SUM SOMA VALORES "SOMÁVEIS"?

Para entender como esse método da API de coleções do Scala funciona, vamos começar vendo a sua assinatura. Dessa vez, vamos ver a definição completa do método em vez da definição simplificada que vimos em um capítulo anterior:

```
def sum[B >: A](implicit num: Numeric[B]): B = foldLeft(num.zero)(
  num.plus)
```

O método possui apenas um parâmetro, e ele é implícito: `num: Numeric[B]`, onde `B` é o tipo dos elementos da lista ou qualquer tipo pai. Na prática, isso quer dizer que podemos somar qualquer tipo de elemento que possua um `Numeric[T]` equivalente. Alguma classe que seja uma `Numeric[Int]`, por exemplo, permite que somemos números inteiros.

E como podemos ver na implementação do método, é esse objeto quem sabe realizar a operação de soma em si. No caso do

`Int` , temos o seguinte código no `object Numeric` (entre outros):

```
implicit object IntIsIntegral extends IntIsIntegral with IntOrdering
```

Sendo que `IntIsIntegral` é definido como o seguinte:

```
trait IntIsIntegral extends Integral[Int]
```

E finalmente, `Integral` tem a seguinte assinatura:

```
trait Integral[T] extends Numeric[T]
```

Ou seja, `IntIsIntegral` é um `Integral` de `Int` , que por sua vez é um `Numeric` de `Int` — que era exatamente o que estávamos procurando. Vamos entender agora como o `Numeric` funciona, em conjunto que a hierarquia que acabamos de aprender.

Como diz o nome, o `Numeric` é uma `trait` que define operações *numéricas* para elementos de algum tipo. Um `Numeric[Int]` , por exemplo, atua em elementos do tipo inteiro. E a API do Scala possui `Numeric` s, organizados como vimos anteriormente, para todos os tipos numéricos. Entre essas operações está a soma, como vimos no trecho de código no começo da seção.

Assim, temos no objeto `Numeric` a definição de diversos *objetos implícitos* implementando a `trait Numeric` , com todas as operações matemáticas que essa `trait` exige. E esses objetos estão disponíveis para serem usados pelo método `sum` automaticamente em qualquer lugar que requeira um `Numeric[T]` .

Como o parâmetro do método `sum` é implícito, e como os objetos que implementam os `Numeric` s são também implícitos, sempre que invocamos o método `sum` o compilador procura entre esses objetos — ou entre objetos criados em algum outro lugar, como veremos a seguir — algum objeto que seja compatível com o tipo dos elementos da coleção sendo somada.

A coisa fica ainda mais interessante quando entendemos que podemos usar esse mecanismo como ponto de extensão da API. Por exemplo, se quisermos suportar a invocação do método `sum` em coleções de nossas classes, tudo o que precisamos fazer é oferecer um objeto `Numeric` equivalente.

Vamos supor que queremos suportar a soma de nossas fotos. É um exemplo estranho, mas que mostra a flexibilidade desse recurso. Vejamos o seguinte código, onde vamos criar um `Numeric[Foto]` :

```
class FotoNumeric extends Numeric[Foto] {  
  override def plus(fotoX: Foto, fotoY: Foto): Foto =  
    fotoX.copy(title = fotoX.title + fotoY.title)  
  
  override def fromInt(x: Int): Foto =  
    new Foto(x, "unknown", 0, "unknown")  
  
  // demais operações omitidas  
}
```

Neste exemplo, estamos definindo que a soma de duas fotos é a concatenação dos seus títulos. Estamos omitindo as demais operações por questão de simplicidade, mas para o exemplo funcionar, temos também de definir as demais operações matemáticas, como `times` e `minus`.

Feito isso, precisamos definir o objeto implícito que será usado quando necessário. Existem diversos lugares nos quais podemos colocar esse objeto, mas o melhor nesse caso será o `object Foto`. Isso pois ele poderá ser usado automaticamente quando algum elemento implícito envolvendo a `Foto` for procurado. Vejamos como ficaria:

```
object Foto {  
  implicit object FotoNumeric extends FotoNumeric  
}
```

E pronto! Já podemos somar listas de fotos usando o método `sum`. Vejamos isso em funcionamento:

```
val fotos = List(
  Foto(123, "jcranky", 1234, "Foto 1"),
  Foto(124, "jcranky", 4321, "Foto 2")
)

println("Somando fotos: " + fotos.sum)
```

Esse é, é claro, um exemplo estranho, mas que mostra a flexibilidade da API. Matematicamente essa operação não faz sentido e o resultado será um pouco estranho. Algo como no código a seguir:

```
Somando fotos: Foto(124,jcranky,4321,unknownFoto 1Foto 2)
```

Os títulos foram concatenados, como esperávamos, mas existe um elemento a mais nele: o `unknown`. Se olharmos novamente nossa implementação do `Numeric[Foto]`, vemos o seguinte:

```
override def fromInt(x: Int): Foto =
  new Foto(x, "unknown", 0, "unknown")
```

E o que aconteceu foi que a implementação do `sum` utiliza uma chamada a `fromInt(0)` como ponto de partida para a soma. Fica como exercício para o leitor explorar essa implementação na API do Scala.

COLOCANDO TUDO JUNTO

Neste capítulo, vamos juntar todos os pedaços que vimos até aqui e criar uma API de ponta a ponta para acessar o Flickr. No processo, brevemente mencionaremos alguns pontos fora do escopo deste livro. Ao leitor interessado, é fortemente recomendado ler mais sobre tais pontos, pois são coisas importantes na hora de se criar um projeto completo. Entre outras coisas, isso inclui o uso do `sbt` e da biblioteca de configuração *Typesafe Config*.

É importante lembrar também de que o código completo da API está disponível no GitHub: <https://github.com/jcranky/scalando>. Portanto, visite o repositório para explorar todos os detalhes do projeto.

13.1 BIBLIOTECAS E FERRAMENTAS

Antes de mais nada, vale a pena lembrar ao leitor onde encontrar a documentação da API do Flickr: <https://www.flickr.com/services/api/>. Afinal, esta é a API que estamos acessando. E o leitor é encorajado a acessar a documentação e se familiarizar um pouco com essa ela. Vale lembrar também que estamos implementando apenas uma pequena parte desta API, a busca por fotos, e que ela oferece muitos outros recursos.

Para o projeto completo funcionar correta e profissionalmente, precisamos utilizar algumas ferramentas e bibliotecas para nos auxiliar, em especial, o já mencionado *sbt* (*scala build tool*); e o projeto completo será dependente do uso dessa ferramenta.

Se ainda não o tiver instalado, vá ao site <http://www.scala-sbt.org/> e faça isso. O processo é simples e está documentado em <http://www.scala-sbt.org/0.13/docs/Setup.html> — para a versão 0.13, que é a mais atual no momento da escrita deste livro.

Em termos de bibliotecas, além do *scala-xml* que exploramos no capítulo *Parseando XML*, vamos usar a *Typesafe Config* (<https://github.com/typesafehub/config>), para nos ajudar a lidar com as configurações do ambiente. Ou seja, para ler coisas como a *api key do flickr* de um arquivo de configuração em vez de ter essa informação *hard-coded* no código-fonte do projeto.

Em programação funcional, sempre que possível, tentamos evitar utilizar exceções e, em vez disso, manipular os erros explicitamente. Assim, evitamos surpresas quando executamos o nosso código.

Para esse fim, vamos utilizar a classe *Either* da *API padrão do Scala*. Nós não a estudamos no decorrer do livro, mas vamos vê-la brevemente a seguir. Ao leitor curioso, fica como exercício investigar seu uso no código do exemplo e o *scaladoc*.

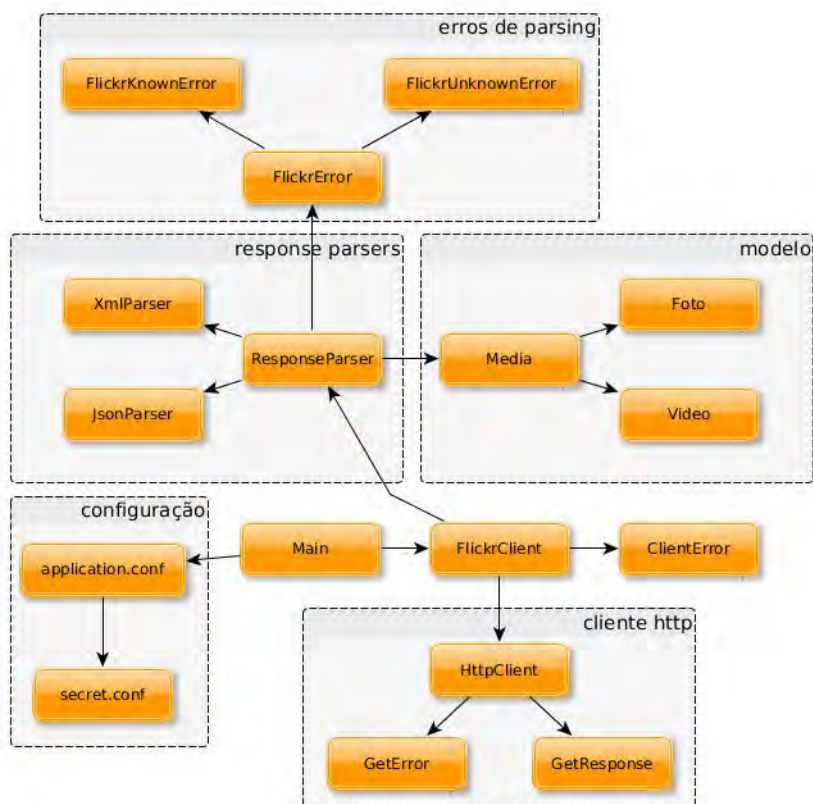
Em paralelo, outra biblioteca que usaremos e que já mencionamos brevemente em um capítulo anterior é o *Specs2* (<https://etorreborre.github.io/specs2/>). No ecossistema Scala, existem basicamente duas excelentes opções para se escrever testes automatizados: *Specs2* e *ScalaTest* (<http://www.scalatest.org/>). As duas merecem ser investigadas pelo leitor curioso, mas para a nossa biblioteca, escolhemos utilizar o *Specs2*.

Ainda no quesito testes, vamos também usar o *Mockito*. Ele é um velho conhecido do mundo Java, e é suportado tanto pelo *ScalaTest* quanto pelo *Specs2*. Portanto, vamos utilizá-lo em conjunto com o *Specs2*. Para isso, precisamos apenas adicionar uma *trait* ao nosso teste, como no código a seguir:

```
class FlickrClientSpec extends Specification with Mockito
```

13.2 COMPONENTES DA API

Nossa API está organizada como no diagrama a seguir:



No diagrama, agrupamos os elementos relacionados para

facilitar a leitura. Não agrupamos apenas três elementos, e vamos começar por eles: `Main` , `FlickrClient` e `ClientError` .

`Main` é a classe principal do exemplo. Na verdade, ele é um `object` , e não é parte da API em si: ela serve apenas para testarmos o código enquanto desenvolvemos. Uma API "normal" provavelmente não conteria tal classe, exceto talvez como parte da documentação e instruções de uso.

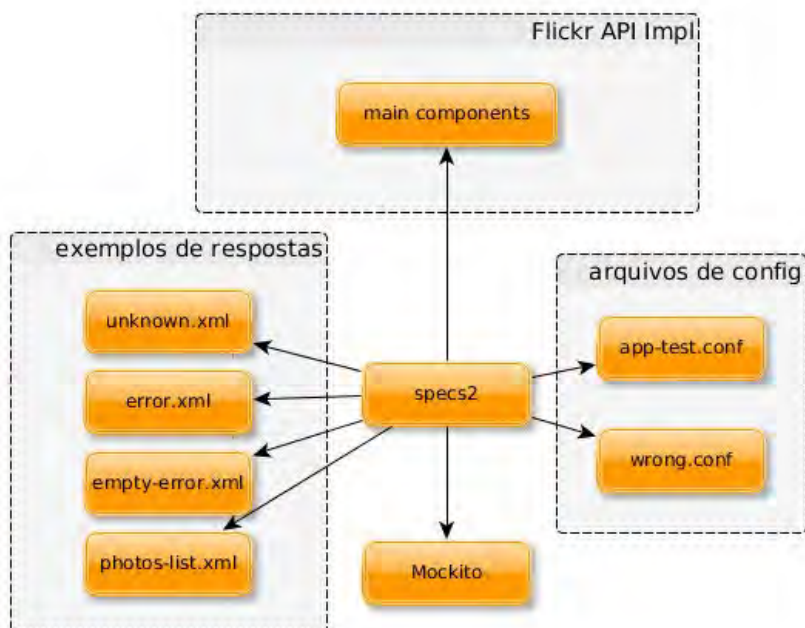
Para usar a API, o `Main` carrega as configurações do arquivo `application.conf` e usa essas informações para criar o `FlickrClient` . Para evitar termos de *commitar a api-key do Flickr*, esse arquivo de configuração também adiciona um outro arquivo, `secret.conf` , que sobrescreve a configuração em questão, e que é ignorado pelo *git* do projeto.

`FlickrClient` representa nossa API de verdade; é esta classe que os clientes da API deverão utilizar. Associada a ela, temos a `ClientError` , generalizando qualquer tipo de erro na comunicação com o Flickr. Para o exemplo do livro, essa classe suporta apenas uma operação: `FlickrClient.buscaFotos` .

Abaixo do `FlickrClient` , temos os elementos responsáveis pelo acesso ao *web service* do Flickr, efetivamente feito pelo `HttpClient` . A responsabilidade deste componente é apenas acessar o Flickr usando os parâmetros recebidos e retornar a resposta como uma `String` — encapsulada no `GetResponse` .

O `FlickrClient` , além de ser o ponto de entrada na API, também é quem "amarra" os componentes internos da implementação. Isto é, ele vai receber a resposta bruta do Flickr mencionada anteriormente, e vai repassá-la ao `ResponseParser` , que será o componente responsável por transformar a resposta `String` em objetos do nosso modelo.

Vamos agora dar uma olhada nos componentes utilizados nos testes automatizados do projeto. O diagrama a seguir dá uma visão geral do que estamos usando e, é claro, ele é bem mais simples do que o diagrama anterior:



O principal elemento que temos aqui é o framework *specs2*. É ele que usamos para escrever todos os testes. Quando necessário, usaremos também o *Mockito* para manter o escopo correto dos testes, e faremos isso utilizando uma pequena DSL oferecida pelo próprio *specs2*.

Temos também dois arquivos de configuração: o `app-test.conf`, usado para os testes que precisam de configurações corretas para funcionar; e o `acre.conf` que, na verdade, não existe. Tentamos ler esse arquivo para ter certeza de que o código sabe lidar com configurações inválidas ou inexistentes.

Por fim, temos quatro arquivos XML:

- `unknown.xml`
- `error.xml`
- `empty-error.xml`
- `photos-list.xml`

Eles representam possíveis respostas do Flickr, e são usados para simular a interação com a nossa API e verificar se o comportamento da implementação está correto.

13.3 CONSIDERAÇÕES FINAIS

Algumas considerações, e a explicação sobre o `Either`, precisam ser feitas antes de finalizarmos o capítulo.

Primeiramente, por questão de simplicidade, estamos ignorando o problema da paginação dos resultados. Em uma API real, porém, esse problema precisaria ser resolvido pois, entre outras coisas, o Flickr limita a quantidade de fotos retornadas por requisição a quatro mil. Veja a documentação em questão aqui: <https://www.flickr.com/services/api/flickr.photos.search.html>.

Quando vimos os exemplos de respostas do Flickr, talvez o leitor tenha reparado que a tag `<photos>` possui alguns atributos que ignoramos:

- `page`
- `pages`
- `perpage`
- `total`

Além disso, o serviço que faz a busca de fotos aceita esses atributos como parâmetros. Juntando essas duas coisas, é possível implementar uma paginação de forma relativamente simples. Fazer

isso fica como exercício para o leitor interessado.

Mudando agora o foco para o `Either`. De forma geral, quando estamos trabalhando com programação funcional, evitamos usar exceções para tratamento de erros, pois esse é um mecanismo imperativo e mais complicado de compor e testar.

Existem diversas outras formas de lidar com erros, e o `Either` é a opção disponível na biblioteca padrão do Scala. Ele não precisa ser usado exclusivamente para isso, mas se encaixa bem.

O `Either` é uma *classe abstrata* que possui apenas dois filhos, de forma similar ao que vimos antes com `Option`: `Left` e `Right`. Na versão final, o nosso `ResponseParser` ficou da seguinte forma:

```
sealed trait ResponseParser {  
  def parse(xmlStr: String): Either[FlickrError, Seq[Foto]]  
}
```

Repare no tipo retornado pelo método `parse`: `Either[FlickrError, Seq[Foto]]`. Em outras palavras, o método agora retorna, ou `FlickrError`, ou `Seq[Foto]`. Além de evitarmos ter de lançar exceções para lidar com os erros, temos uma outra vantagem bastante importante aqui: estamos obrigando o cliente do método a lidar com o problema. Ou seja, não será possível para ele esquecer de tratar eventuais erros.

Se olharmos o método `get` do nosso `HttpClient`, veremos a mesma técnica sendo utilizada:

```
class HttpClient {  
  def get(url: String): Either[GetError, GetResponse] = ???  
}
```

Ou seja, aqui estamos dizendo que vamos retornar, ou `GetError`, ou `GetResponse`. E novamente, o cliente do método é obrigado a lidar com o problema.

Finalmente, no `FlickrClient`, temos novamente a mesma estrutura:

```
class FlickrClient( /* parâmetros omitidos */ ) {  
  def buscaFotos(tags: List[String]): Either[ClientError, Seq[Foto  
]] = ???  
}
```

Quando o método `main` invoca o `FlickrClient`, pedindo por fotos, ele terá então de lidar com o fato de que é possível que o pedido falhe. Neste caso, estamos fazendo isso usando *pattern matching* e declarando uma ação para o caso do sucesso, e uma outra ação para o caso da eventual falha. Vejamos o código a seguir:

```
flickrClient.buscaFotos(List("scala")) match {  
  case Right(fotos) => fotos.foreach(println)  
  case Left(err) => println(s"Error getting fotos: ${err.msg}")  
}
```

Aqui, `Right` representa o sucesso, e `Left` a falha, e estes correspondem à posição dos tipos declarados dentro do `Either` anteriormente. Não existe uma regra que nos obriga a posicionar os elementos dessa forma, e o compilador não vai reclamar se fizermos o oposto. Entretanto, o comum é colocar o sucesso no `Right`, aproveitando o significado em inglês da palavra *Right* que, além de *direita*, também quer dizer *correto*.

Vamos ver mais um exemplo de código da API, agora extraído do `FlickrClient`. O código a seguir é usado para invocar o web service do Flickr e, caso a resposta recebida seja válida, transformá-la em uma lista de fotos. Ele é um pouco mais complexo, pois precisa lidar com erros em dois níveis:

```
val response = httpClient.get(url)  
  
response.fold(  
  (err) => Left(ClientError(err.msg)),  
  (resp) => responseParser.parse(resp.body) match {  
    case Right(parsed) => Right(parsed)  
    case Left(error) => Left(ClientError(error.toString))  
  })
```

```
}  
)
```

Lembrando de que o tipo do `val response` é `Either[GetError, GetResponse]`, o que estamos fazendo é uma operação de `fold` um pouco diferente do que vimos em outro capítulo. Aqui, o `fold` recebe dois parâmetros: o primeiro é uma função para ser aplicada caso o `Either` seja `Left`, e o segundo caso ele seja `Right`.

No caso do `Left`, ou seja, do *erro*, estamos apenas transformando o erro recebido em um erro conhecido e válido para esta camada do sistema. Isto é, estamos fazendo uma pequena transformação no erro, mas o mantemos na aplicação, pois o usuário que invocou esse código terá de lidar com ele de alguma forma.

Já no caso do `Right`, ou seja, sucesso, estamos pegando a resposta recebida pelo cliente `http` e a repassando para o `reponseParser`. Essa operação, por sua vez, também pode funcionar ou falhar.

Para verificar isso, usamos um *pattern matching*, de forma bem similar ao que vimos antes, e mapeamos o sucesso para ser o retorno final dessa operação, ou a falha novamente para um erro válido para esta camada da aplicação.

Vimos neste capítulo apenas alguns trechos de código, mas o projeto completo pode ser encontrado no GitHub, em <https://github.com/jcranky/scalando>. Não deixe de passar lá!

O FIM, E O COMEÇO

O livro apresentou os principais conceitos e os elementos mais importantes da linguagem Scala e, se o leitor chegou até aqui, é porque provavelmente gostou do que viu. E agora, o que vem a seguir? Quais são os próximos passos é provavelmente a pergunta a ser respondida.

Primeiramente, se ainda não conhece o grupo de usuários Scala de São Paulo, os *Scaladores*, dê uma olhada em <http://scaladores.com.br>. O grupo é muito amigável com iniciantes, seja na lista de discussão, nas conversas no *Slack* ou nas reuniões presenciais. Também são abordados tópicos mais avançados com frequência, e muitas das apresentações do grupo estão disponíveis no *YouTube*, em <http://youtube.com/scaladores>.

Para o leitor que quiser um ponto de vista prático, mas diferente da linguagem, também abordando os principais recursos, o livro *Scala for the Impatient*, de Cay Horthmann, é uma boa pedida. Outro livro interessante abordando um bom conjunto de recursos da linguagem Scala é o *Programming Scala*, por Dean Wampler e Alex Payne. E é claro, para quem quer uma visão completa e detalhada, o livro do criador da linguagem, Marting Odersky, *Programming in Scala* é o caminho.

Um outro tipo de leitura que pode ser interessante é sobre o padrão de design *Singleton*. Isso porque essa é a base para a implementação de `object` s em Scala. Não vamos deixar nenhuma

referência específica para isso, pois é muito simples encontrar algo a respeito na internet. O padrão *Factory Method*, ou em português *Método de Fábrica*, que está relacionado muitas vezes ao uso do método `apply` em Scala, também vale a pena ser investigado.

Sendo mais específico, é preciso o leitor decidir que tipo de uso quer fazer da linguagem. Se quiser, por exemplo, explorar possibilidades divertidas, mas ainda não muito maduras ou difundidas no mercado, existem opções como *Scala com Android* (<https://www.youtube.com/watch?v=ZoYJidA7nIw>) ou o *desenvolvimento de plugins para Minecraft com Scala*, usando o *EasyForger* (<http://easyforger.com/>).

Seguindo uma linha mais "tradicional", o desenvolvimento de aplicações web ou APIs *RESTful*, opções como o *Play Framework* (<http://playframework.com/>), *Scalatra* (<http://scalatra.org/>), *Lift* (<http://liftweb.net/>) ou *Spray.io* (<http://spray.io/>) valem o tempo investido.

Para quem se interessa por aplicações *back-end* altamente performáticas, provavelmente o contexto no qual Scala mais chama atenção, o caminho é estudar o framework *Akka* (<http://akka.io/>) e o modelo de atores (<https://www.youtube.com/watch?v=Zwjxkci6xdw>). Como também dar uma olhada no *Finagle* (<https://twitter.github.io/finagle/>) do *Twitter*, e estudar mais a fundo a API de *Futures* do Scala (<http://docs.scala-lang.org/overviews/core/futures.html>). Essas são diversas opções para se desenvolver aplicações altamente concorrentes e performáticas em Scala.

Deixando frameworks de lado e indo na direção dos recursos mais avançados, o livro *Scala Puzzlers*, por *Andrew Phillips* e *Nermin Serifovic*, é uma excelente leitura. Ele aborda casos específicos de uso da linguagem onde seu funcionamento é mais enigmático ou até mesmo surpreendente em alguns momentos.

Por fim, outro assunto que pode interessar o leitor e que também é um pouco mais avançado é o uso da linguagem de forma *puramente funcional*. Para esse assunto, o livro *Functional Programming in Scala*, por *Paul Chiusano* e *Rúnar Bjarnason*, é uma boa leitura.

E é claro, não deixe de passar no fórum da Casa do Código! É só acessar <http://forum.casadocodigo.com.br/>.

Boa sorte e até a próxima!