

Ruby

Aprenda a programar
na linguagem mais divertida



© Casa do Código

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

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

Edição

Vivian Matsui

Carlos Felício

[2020]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

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

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

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

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

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

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

ISBN

Impresso e PDF: 978-85-66250-24-4

EPUB: 978-85-5519-000-1

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

AGRADECIMENTOS

A primeira pessoa e mais importante à qual gostaria de agradecer é minha esposa, Rafaela Fernandes, e também à Clarinha, pela compreensão nas várias noites em que tive de deixar de dar atenção a elas para me dedicar à escrita do livro.

Aos meus pais e minha irmã que apoiaram e incentivaram bastante o processo de escrita durante todos estes meses.

Agradecimentos especiais ao Paulo Silveira e Guilherme Silveira, que me deram a oportunidade ímpar de trabalhar na Caelum, umas das melhores empresas de tecnologia para trabalhar no país. Um agradecimento novamente ao Paulo Silveira e Adriano Almeida, que na QCON 2012, em uma conversa de poucos minutos, me apoiaram na ideia de escrever um livro sobre Ruby.

A todos os amigos que trabalham ou trabalharam comigo na construção do novo CMS do portal R7.com, pois foi participando deste time que aprendi e ensinei grande parte das coisas que sei sobre Ruby. Em especial, aos amigos: Bruno Grasselli, Elber Ribeiro, Fernando Meyer, Caio Filipini e Rafael Ferreira. Dos amigos citados, um agradecimento especial ao Rafael Ferreira, um dos melhores desenvolvedores com que tive a oportunidade de trabalhar. Uma pessoa com quem aprendo algo novo todos os dias.

Por fim, obrigado a Deus por colocar esta oportunidade na minha vida.

QUEM SOU EU?

Meu nome é Lucas Souza, formado em Engenharia da Computação na Universidade de Ribeirão Preto. Trabalho profissionalmente com desenvolvimento de software há sete anos. Durante boa parte desses anos, trabalhei dentro de empresas situadas em Ribeirão Preto, mas há quatro anos estou em São Paulo. Nesse período, trabalhei principalmente com Java e Ruby.

Em 2005, já programava utilizando PHP, mas decidi que gostaria de aprender outras linguagens e optei por aprender Java. Rapidamente comecei a trabalhar com Java e, no ano de 2006, participei de um projeto com o qual foi possível aprender não só Java, mas também boas práticas de desenvolvimento de software: testes, integração contínua, refatoração de código etc.

No ano de 2008, tive a oportunidade de conhecer a Caelum. Foi quando resolvi me mudar para São Paulo após receber o convite para trabalhar como consultor. Após alguns meses, tive a oportunidade de me tornar instrutor dos cursos de Java existentes na época. Fui editor-chefe do InfoQ Brasil por quase dois anos, onde era responsável pela manutenção, publicação e revisão de todo o conteúdo técnico do site.

Também participei da criação dos novos cursos de Hibernate e JSF da Caelum, onde desenvolvi o gosto pela escrita. Paralelo a isso, tive contato com vários outros desenvolvedores da Caelum, que me incentivaram a aprender um pouco sobre Ruby, que já era uma vontade minha naquele tempo.

Em 2011, recebi o convite para ser um dos integrantes do time responsável por desenvolver o novo CMS do portal R7.com, que

seria escrito principalmente em Ruby. Aceitei o desafio, e desde então me dedico diariamente ao aprendizado de coisas novas em relação a essa linguagem. Mas não só isso, eu gosto particularmente de resolver problemas relacionados à arquitetura que visam melhorar a escalabilidade e alta disponibilidade do portal.

Procuro sempre as melhores formas de escrever códigos legíveis e testáveis utilizando Ruby. Apesar de ser um apaixonado pela linguagem e considerá-la uma das melhores com as quais já trabalhei, costumo criticar seus pontos fracos, inclusive neste próprio livro. Acho que cada problema possui uma linguagem melhor para resolvê-lo.

UM BREVE PREFÁCIO

Ruby é uma linguagem dinâmica, orientada a objetos e que possui algumas características funcionais. Seu criador, Yukihiro Matsumoto, queria uma linguagem que juntasse programação funcional e imperativa, mas acima de tudo que fosse uma linguagem legível. Esta é uma das grandes vantagens da linguagem, ser extremamente legível.

Este livro é basicamente um tutorial e uma referência para a linguagem Ruby. Ele cobre a maioria das características da linguagem e também suas principais APIs: `String` , `Enumerable` , `File` etc. Além disso, veremos questões mais avançadas, que permitirão um maior aproveitamento da linguagem, como metaprogramação, distribuição de código e gerenciamento de dependências.

POR QUE RUBY?

Além das características citadas anteriormente, Ruby é a linguagem que eu utilizo para a maioria dos programas que escrevo, principalmente quando vou começar aplicações web. Trabalho há dois anos com Ruby, e posso dizer que a linguagem é extremamente produtiva e simples, consigo fazer coisas simples, com poucas linhas de código.

Nos últimos anos, a linguagem progrediu assustadoramente. A comunidade cresceu bastante: possui o Rubygems, onde se encontra um grande número de projetos que auxiliam o dia a dia do desenvolvedor Ruby. No GitHub, a grande maioria dos repositórios é escrita em Ruby, o que permite que a comunidade contribua cada vez mais para a melhoria do ambiente em volta da linguagem.

Além disso, o framework MVC Ruby on Rails permite a criação de aplicações web com extrema rapidez. Essa agilidade tem sido considerada por várias *startups* no momento da criação de seus produtos. A vantagem é que o número de vagas disponíveis no mercado cresce a cada dia, principalmente em polos de desenvolvimento como a Califórnia.

Atualmente, aprender apenas Rails não é o suficiente. É necessário um bom conhecimento da linguagem para criar códigos que facilitem a manutenção e criação de novas funcionalidades. Aprender mais sobre a linguagem Ruby faz com que você consiga escrever códigos mais legíveis e deixe de lado vícios que podem ter vindo de outras linguagens com as quais você trabalhava.

Sumário

1 Uma introdução prática à linguagem Ruby	1
1.1 Quando? Onde? Por quê?	1
1.2 Instalação	2
1.3 Tudo pronto... mãos à massa: inferência de tipos	9
1.4 Tipagem forte e dinâmica	11
1.5 Uma linguagem interpretada e com classes abertas	13
1.6 Onde eu usaria Ruby?	15
2 Seu primeiro passo no Ruby: convenções e as diferentes estruturas primitivas	18
2.1 Mais tipos no Ruby	18
2.2 Comente seu código	19
2.3 O trabalho com números	20
2.4 Representação de textos com as Strings	21
2.5 Estruturas de controle	24
2.6 Entenda o valor nulo	25
2.7 Substitua o "if not" por "unless"	26
2.8 Iterações simples com for, while, until	27
2.9 As outras formas de declarar Strings	29

2.10 Próximos passos	31
3 O começo da nossa aplicação	33
3.1 A definição da classe Livro	34
3.2 Crie a estrutura do projeto	40
3.3 Defina os atributos de instância	44
3.4 Sobrescrevendo o método to_s	46
3.5 Alteração e leitura de atributos	48
3.6 Atributos nem tão privados assim	54
3.7 Grandes poderes, grandes responsabilidades	55
4 Estruturas de dados	58
4.1 Trabalhe com Arrays	58
4.2 Guardando nossos livros	60
4.3 Percorrendo meu array	63
4.4 Como separar os livros por categoria: trabalhe com Hash	64
4.5 Indo mais a fundo: Hashes no Ruby 1.9	71
4.6 Indo mais a fundo: o operador =	72
4.7 Indo mais a fundo: o tipo Set	73
4.8 Próximos passos	78
5 Ruby e a programação funcional	79
5.1 O que é programação funcional	79
5.2 Funções puras	79
5.3 Comandos que retornam valores	81
5.4 Funções de alta ordem: higher-order functions	83
5.5 Crie seu próprio código que usa um bloco de código	85
5.6 Explorando a API Enumerable	90

5.7 Para saber mais: outras maneiras de criar blocos	97
5.8 Para saber mais: currying	104
5.9 Para saber mais: closure	107
5.10 Próximos passos	108
6 Explorando API File	110
6.1 Um pouco da classe File	110
6.2 Serialização de objetos	112
6.3 Salvando objetos em arquivos	114
6.4 Recuperando objetos salvos	117
6.5 Próximos passos	121
7 Compartilhando comportamentos: herança, módulos e mixins	122
7.1 Herança: compartilhando comportamentos com classes	123
7.2 Herança e variáveis de instância	128
7.3 Os custos no uso da herança	135
7.4 Módulos	140
7.5 Indo mais a fundo: constant Lookup de dentro para fora	149
7.6 Duck Typing: o polimorfismo aplicado no Ruby	151
7.7 Herança ou mixing? Qual devo usar?	157
8 Metaprogramação e seus segredos	160
8.1 Entenda o self e method calling	160
8.2 O impacto do self na definição de classes	164
8.3 Singleton Class e a ordem da busca de métodos	166
8.4 Indo mais a fundo: acessando a singleton class	170
8.5 Metaprogramação e as definições de uma classe	172

8.6 Criando um framework para persistir objetos em arquivos	184
8.7 Gerenciando exceções e erros	192
8.8 A exclusão de dados implementada com metaprogramação	197
8.9 Method lookup e method missing	213
8.10 Utilizando expressões regulares nas buscas	228
8.11 Próximos passos	235
9 As bibliotecas no universo Ruby	237
9.1 Como manusear suas gems com o Rubygems	237
9.2 Gerenciando várias versões de uma gem	241
9.3 Gerencie dependências com o Bundler	242
9.4 Criando e distribuindo gems	245
9.5 Distribuição da biblioteca	248
9.6 Próximos passos	251
10 Criando tasks usando rake	252
10.1 Parâmetros na rake task	256
10.2 Tasks com pré-requisitos	259
10.3 Próximos passos	261
11 RVM (Ruby Version Manager)	263
11.1 Instalação	263
11.2 Instalando diferentes Rubies	265
11.3 Organize suas gems utilizando gemsets	266
11.4 Troque automaticamente de gemsets com .rvmrc	269
11.5 Próximos passos	271
12 Ruby 2.0	272

12.1 Evitando monkey patches com refinements	272
12.2 Named parameters	276
12.3 Utilize prepend em vez de include	279
12.4 Utilizando lazy load no módulo Enumerable	281
12.5 Encoding UTF-8	283
12.6 Próximos passos	283
13 Apêndice: concorrência e paralelismo	285
13.1 Threads	287
13.2 Múltiplos processos	288
13.3 Fibers	289
13.4 O design pattern Reactor	291
13.5 Conclusão	292

Versão: 24.5.24

UMA INTRODUÇÃO PRÁTICA À LINGUAGEM RUBY

Vamos começar com um pouco da história e ver características importantes da linguagem Ruby, para compará-la com outras com que você já deve ter trabalhado. Também mostraremos por que vários programadores têm falado e usado tanto esta linguagem, pela qual espero que você se apaixone.

1.1 QUANDO? ONDE? POR QUÊ?

A linguagem Ruby foi criada por Yukihiro Matsumoto, mais conhecido como Matz, no ano de 1995 no Japão. Ela tinha como objetivo ser uma linguagem mais legível e agradável de se programar.

Mas, além das características orientada a objetos, Ruby também foi criada para possuir um forte quê de linguagem funcional, tendo recursos poderosos e essenciais desse paradigma, como lambdas e *closures*. Ela foi inspirada em outras linguagens como Perl, Smalltalk e Lisp, e hoje está entre as linguagens mais usadas, muito em função da disseminação do seu principal

framework MVC, o Ruby on Rails (<http://rubyonrails.org>).

Algumas características do Ruby devem ser fixadas desde o começo dos estudos, pois vão facilitar bastante nossa curva de aprendizado, como a tipagem forte e dinâmica, além do fato de a linguagem ser interpretada. Fique tranquilo caso você ainda não tenha escutado falar sobre alguns desses conceitos.

1.2 INSTALAÇÃO

Em março de 2017, foi lançada a versão 2.4.1 da linguagem. Há um apêndice neste livro para você fazer a instalação da versão mais recente, que traz novos recursos, também discutidos no fim do livro.

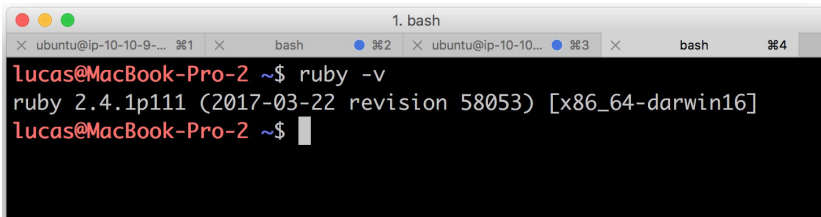
Nesta seção, vamos aprender a instalar o Ruby em cada sistema operacional.

Mac OS

Adivinhe? No Mac OS, o interpretador Ruby já está instalado! Abra o terminal e execute:

```
ruby -v
```

Nas versões El Capitan, Yosemite, Mavericks e Sierra, a versão 2.0 vem incluída por padrão, conforme você pode ver na imagem a seguir:

A screenshot of a terminal window on a Mac. The window title is "1. bash". The terminal shows the command "ruby -v" being executed. The output is "ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]". The prompt "lucas@MacBook-Pro-2 ~\$" is visible at the bottom.

```
lucas@MacBook-Pro-2 ~$ ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
lucas@MacBook-Pro-2 ~$
```

O Ruby possui vários interpretadores disponíveis. Se você deseja utilizar versões mais novas que não estão disponíveis diretamente a partir do seu sistema operacional, existem gerenciadores de versões do Ruby, por exemplo, o RVM (*Ruby Version Manager*) ou Rbenv. No apêndice *Gerenciadores de versões do Ruby*, explicarei como elas funcionam e como instalá-las.

Linux

Se você for um usuário Linux, as distribuições, em sua maioria, disponibilizam alguns interpretadores Ruby. Caso você esteja usando a versão 17.04 do Ubuntu, que é a mais recente, basta instalar o pacote do interpretador Ruby utilizando o `apt-get install`. Abra um terminal e execute o comando:

```
sudo apt-get install ruby2.4
```

Agora você pode conferir a versão instalada executando em um terminal:

```
ruby -v
```

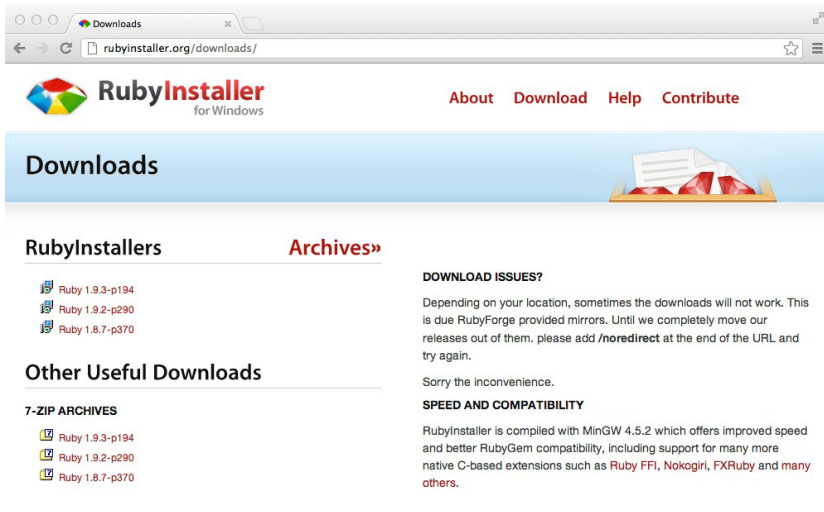
Você verá na saída do terminal algo como:

```
Terminal
lucas@lucas ~
$ ruby -v
ruby 1.9.3p0 (2011-10-30 revision 33570) [x86_64-linux]
lucas@lucas ~
$
```

Windows

Caso o seu sistema operacional seja Windows, a maneira mais simples e fácil é usar umas das versões do RubyInstaller, que permite que você faça a instalação com apenas alguns cliques.

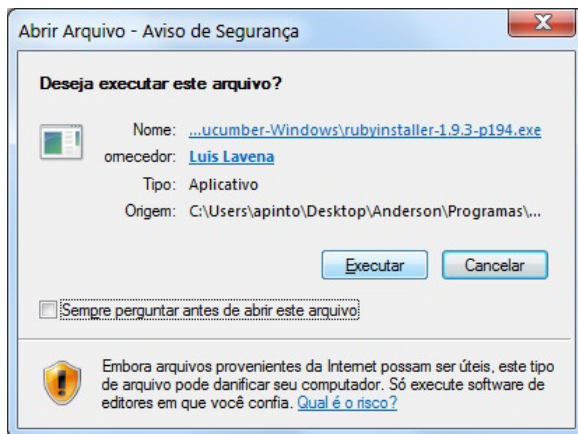
O primeiro passo é baixar a última versão do RubyInstaller. Para isso, acesse o site: <http://rubyinstaller.org/downloads/>.



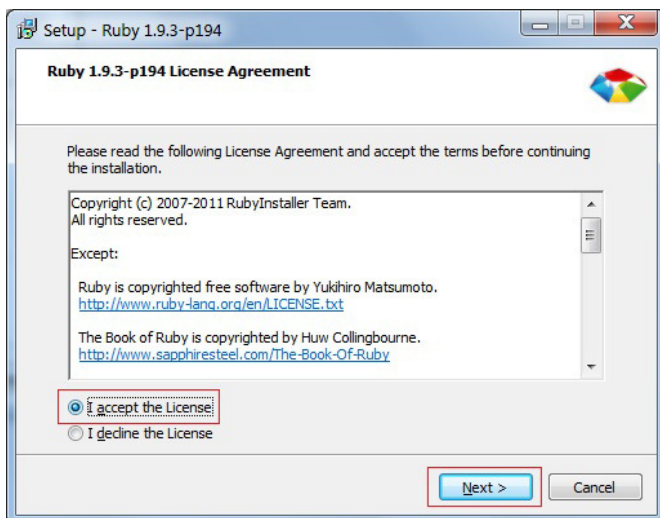
Baixe a versão Ruby 2.3.3 , um arquivo executável que instalará automaticamente o interpretador Ruby em sua máquina.

Quando o download terminar, execute o arquivo `rubyinstaller-2.3.3.exe`.

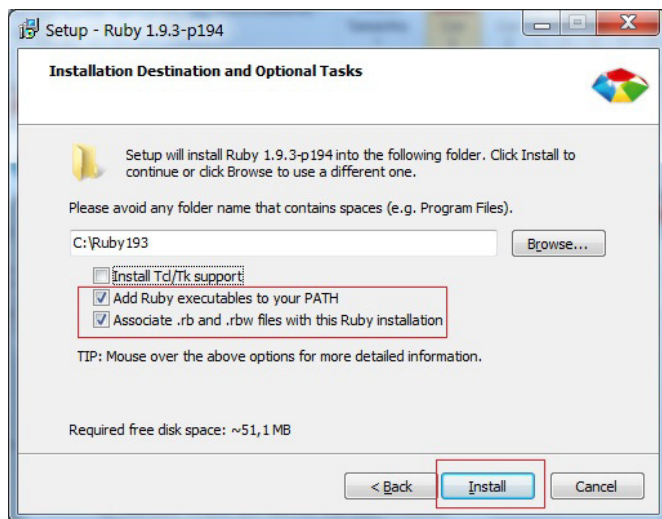
Clique em **Executar** para prosseguir com a instalação.



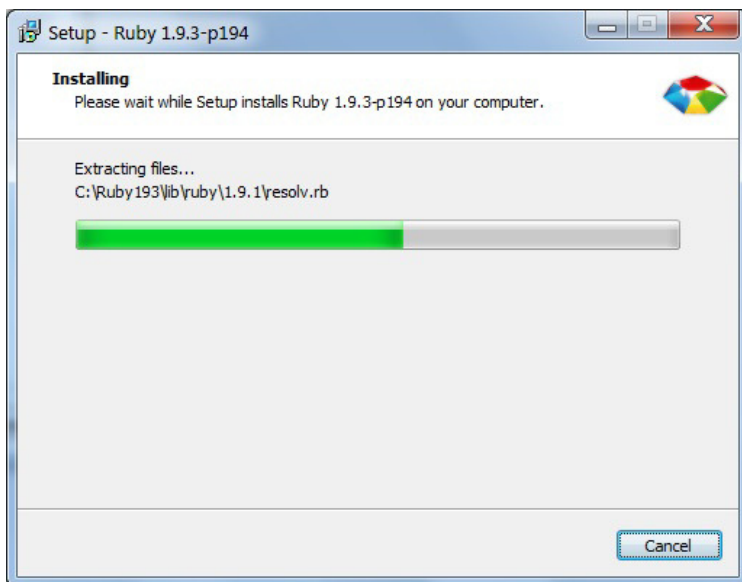
Em seguida, clique no botão **I accept the License** e, depois, no botão **Avançar**.



Agora marque as opções `Add Ruby executables to your PATH` para poder posteriormente executar o Ruby a partir de um terminal. Marque também a opção `Associate .rb and .rbw file with this Ruby installation` para que os arquivos `.rb` sejam interpretados como arquivos que contêm código Ruby. Por fim, clique na opção `install`.



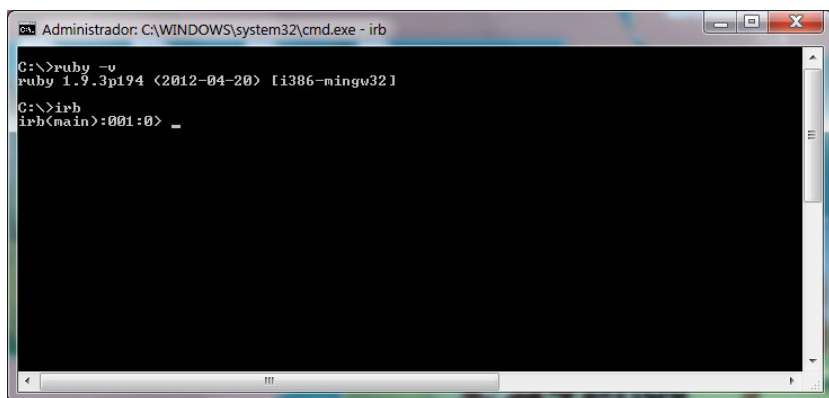
A instalação será feita no diretório selecionado na tela anterior.



A instalação será completada com sucesso. Para finalizar, basta clicar no botão `Finish`.



Para testar que a instalação foi feita com sucesso, abra um terminal e execute o comando `ruby -v` e veja que o Ruby foi instalado:



1.3 TUDO PRONTO... MÃOS À MASSA: INFERÊNCIA DE TIPOS

Um dos conceitos básicos em linguagens de programação é a declaração de variáveis, que é apenas uma associação entre um nome e um valor. Em Ruby, basta definirmos o nome da variável e atribuir um valor usando o sinal `=` (igual).

```
idade = 27
```

Esse código deve ser executado dentro do IRB (*Interactive Ruby Shell*), um pequeno *shell* que permite que códigos Ruby sejam criados e testados. Como os códigos dos primeiros capítulos são simples, vamos testá-los dentro do IRB. Para executá-lo, basta digitar `irb` no terminal de sua preferência, ou no console do Windows.

Outra forma de criar e executar código Ruby é criando um arquivo `.rb` e executá-lo utilizando o comando `ruby`. Se o código anterior fosse digitado dentro de um arquivo `idade.rb`, para executá-lo faríamos:

```
ruby idade.rb
```

Durante os primeiros capítulos do livro, usaremos o IRB. Conforme os códigos ficarem complexos, será preferível que eles sejam criados dentro de arquivos `.rb`, para que seja fácil você fazer as edições. O IRB será utilizado quando quisermos fazer testes rápidos.

Ao executarmos o código, estamos definindo uma variável chamada `idade` e atribuindo o valor `27`. Mas qual o tipo desta variável? Não é necessário declararmos se ali será guardado um

número, um texto ou um valor booleano?

A variável que criamos é do tipo `Fixnum`, um tipo especial do Ruby que representa números inteiros. Mas não declaramos essa informação na variável `idade`. Sendo assim, como o Ruby sabe que o tipo da variável é numérico?

Ao contrário de outras linguagens, como C, em que é necessário declarmos o tipo da variável, na linguagem Ruby não precisamos fazer isso, pois o interpretador infere o tipo da variável automaticamente durante a execução do código. Esta característica é conhecida como inferência de tipos.

TESTE SEUS CÓDIGOS ONLINE

Se você estiver com um pouco mais de pressa e quiser testar os códigos de exemplo logo, você pode usar o site <http://tryruby.org>. Ele funciona como um IRB, porém, dentro do seu browser favorito. É extremamente útil para quando você quer fazer um teste rápido, mas está em um computador que não tenha o Ruby instalado.

Para verificar de que tipo é a variável `idade`, basta executar o código a seguir:

```
idade = 27  
puts idade.class
```

Quando invocamos `.class` em qualquer variável, o interpretador Ruby retorna o tipo da variável, que será impressa no IRB pelo método `puts`. Neste caso, o tipo retornado é `Fixnum`.

1.4 TIPAGEM FORTE E DINÂMICA

Se eu não declaro qual o tipo da minha variável, quer dizer que o tipo dela não importa para meu interpretador?

Para a linguagem Ruby, a resposta é **não**. E esta definição é uma das que causam mais confusão na cabeça das pessoas que estão começando a programar em Ruby. Por ser uma linguagem com inferência de tipos, muitos pensam que o tipo não importa para o interpretador como acontece com o PHP. Confuso? Vamos ver isso na prática.

Se criarmos duas variáveis em um código PHP qualquer, sendo que uma contém uma `String`, como valor `27` e outro de tipo numérico com valor `2`:

```
<?php
$idade = 27;
$multiplicador = "2";

$idade * $multiplicador;
?>
```

Ao executarmos esse código, o resultado será `54`. Para o interpretador do PHP, quando executamos uma operação matemática envolvendo uma variável do tipo `int` e uma do tipo `String`, os valores das variáveis podem ser automaticamente convertidos para outros tipos, a fim de possibilitar a operação.

Se executarmos exatamente o mesmo código em Ruby:

```
idade = 27
multiplicador = "2"

idade * multiplicador
```

Opa! Não funcionou. O interpretador retornou `TypeError: String can't be coerced into Fixnum`. O Ruby não permite que uma variável `Fixnum` seja somada com outra do tipo `String`, diferentemente do que ocorre no PHP, que permite este tipo de operação. Isso acontece porque, para o interpretador Ruby, é impossível multiplicar um número com um texto.

Essa característica da linguagem de realizar operações em variáveis de tipos diferentes é o que chamamos de tipagem fraca ou forte. No caso do Ruby, em que o tipo é determinante para o sucesso da operação, dizemos que **a linguagem tem tipagem forte**.

Tendo a tipagem forte em mente, vamos ir mais além. Execute o seguinte código:

```
idade = 27
idade = "27"
```

Funcionou?

Esse código funciona normalmente, porque não estamos fazendo nenhuma operação que misture os tipos. O código apenas atribui um `Fixnum` à variável `idade`, e depois atribui um outro valor, só que do tipo `String`. Quando a linguagem permite que o tipo da variável possa ser alterado durante a execução do programa, dizemos que **ela tem tipagem dinâmica**.

Em contrapartida, outras linguagens possuem tipagem estática, ou seja, uma vez que a variável é criada como um `int`, ela só poderá ser um `int`. Esse é o caso do Java, em que o compilador cuida de fazer essa checagem.

Esta é uma das características mais criticadas pelos desenvolvedores que preferem linguagens com tipagem estática.

Uma das alegações é que, em linguagem com tipagem estática, podemos descobrir erros durante a compilação. Porém, veremos durante o livro que a tipagem dinâmica é uma das características mais poderosas do Ruby, trazendo também muitos benefícios.

Vale lembrar de que os conceitos de forte, fraca, dinâmica e estática não são tão simples assim. É mais fácil falar se uma linguagem é mais ou então menos fortemente tipada que outra, por exemplo. Neste capítulo, apenas tocamos de leve nesse assunto.

1.5 UMA LINGUAGEM INTERPRETADA E COM CLASSES ABERTAS

Ruby é uma linguagem interpretada, ou seja, não existe um processo de compilação para um binário executável, como acontece na linguagem C, por exemplo. Em Ruby, existe um arquivo com a extensão `.rb` e um programa cujo papel é interpretar o conteúdo deste arquivo, transformando-o em instruções de máquina e executando o comportamento esperado.

Vamos para um exemplo prático que precisamos resolver: descobrir o plural de todas as palavras, por exemplo. Um caminho natural para solucionar este problema é criar um método que receba uma `String` como entrada e adicione um `s` após a sua última letra (sim, estou pensando apenas nos casos mais simples).

Vamos ao código que soluciona esse problema:

```
def plural(palavra)
  "#{palavra}s"
end
```

```
puts plural("caneta") # canetas
puts plural("carro") # carros
```

Não se preocupe ainda com o significado do `#` dentro da `String` no método `plural`, vamos falar bastante dele durante o livro. O `puts` é um método simples que pega o argumento passado e joga para a saída padrão. Dentro do IRB, bastaria escrever `plural("caneta")` que automaticamente o interpretador joga o resultado no console. Faça o teste!

Olhando este código com cuidado, ele não nos parecerá tão orientado a objetos assim. Repare que os dados e os comportamentos estão separados, mas podemos tentar melhorar isso. Já que o método `plural` age somente na própria `String`, nada mais natural que ele esteja na classe `String`, assim usá-íamos:

```
puts "caneta".plural
puts "carro".plural
```

Porém, ao executarmos esse código, receberemos o seguinte erro como retorno:

```
NoMethodError: undefined method 'plural' for "caneta":String
```

Isso significa que objetos do tipo `String` não possuem o comportamento `plural`. Mas espera aí... e se escrevêssemos o método `plural` dentro da classe `String`?

```
class String
  def plural
    "#{self}s"
  end
end
```

Agora tente executar o código que coloca a `String` "caneta" no `plural`:

```
puts "caneta".plural # canetas
```

Agora funciona! O que fizemos foi "abrir" a classe `String` **durante** a execução do código e adicionamos um novo método que estará disponível para todos os objetos do tipo `String` que existem. Esse recurso é conhecido como **classes abertas** (*OpenClasses*), recurso que veremos bastante ainda durante os próximos capítulos e discutiremos bastante suas vantagens e desvantagens.

Por enquanto, é importante termos em mente alguns problemas em relação a linguagens interpretadas. Um desses problemas é não conseguirmos descobrir erros do programador durante a codificação do programa, mas apenas quando tentamos executá-lo, já que não temos um compilador que consegue checar por erros para nós. Para contornar esse problema, os desenvolvedores Ruby, desde cedo, mantêm o hábito de desenvolver testes de unidade para suas classes, a fim de tentar descobrir possíveis erros existentes no programa o mais rápido possível.

1.6 ONDE EU USARIA RUBY?

É muito comum encontrar a linguagem Ruby sendo usada para a criação de scripts para ler e processar arquivos, automatizar builds e deploys, fazer *crawling* de sites etc.

Mas, sem dúvidas, os maiores cases de sucesso da linguagem Ruby estão ligados a aplicações web. A criação do Rails inquestionavelmente foi o que mais alavancou o sucesso da linguagem. Ele é um framework web que segue o padrão MVC (*Model-View-Controller*) e evangeliza o **Convention over**

configuration, ou seja, acaba com aquela quantidade enorme de XMLs que existem na maioria dos frameworks web do mercado e que dificultam muito a manutenção.

Surgiu dentro da empresa 37signals e foi criado por David Heinemeier Hansson, que resolveu extrair parte da lógica web de um projeto chamado Basecamp. Existem vários cases de sucesso na web como o próprio Basecamp, A List Apart (revista com dicas de desenvolvimento de aplicações web), Groupon (maior site de vendas coletivas de mundo) etc.

Existem também vários SaaS (*Softwares as a Service*) disponíveis, que foram criados em Rails e que permitem a criação de aplicações em minutos. O Shopify é um deles e permite a criação de sua própria loja virtual, com template personalizado, lista de produtos, pagamento em cartão de crédito, isso tudo com apenas alguns cliques. Além de vislumbrar o uso na criação da sua própria loja, você pode pensar em criar o seu próprio SaaS usando Rails com bastante rapidez.

Foi o que o pessoal da MailChimp fez. Eles criaram um serviço que ajuda você a criar o design do seu e-mail de *newsletters*, compartilhá-lo em redes sociais e integrar com outros serviços que você já utiliza. Tudo isso com poucos cliques.

O importante é que o mercado em torno da linguagem Ruby é muito grande atualmente, e a tendência é que continue crescendo nos próximos anos. São várias as vagas existentes para trabalhar com Ruby e/ou Rails hoje em dia. Vários serviços estão sendo criados e uma grande quantidade de startups apostam na linguagem e nas suas aplicações.

Tenha em mente que o quanto mais você dominar a linguagem Ruby mais fácil será para você conhecer profundamente os frameworks como o Rails e as diversas bibliotecas relacionadas.

Vamos começar nossa jornada?

SEU PRIMEIRO PASSO NO RUBY: CONVENÇÕES E AS DIFERENTES ESTRUTURAS PRIMITIVAS

Acabamos de aprender um pouco sobre o Ruby: escrevemos já algum código, o suficiente para que pudéssemos fazer um "olá mundo" e entender alguns dos conceitos por trás da linguagem.

Antes de criar uma aplicação, que começará a ser escrita no próximo capítulo, precisamos nos acostumar com algumas sutilezas da linguagem e também com as estruturas básicas. Vamos aprender mais sobre os tipos existentes em Ruby, como representar ausência de valores, as diferentes maneiras de escrever as estruturas condicionais e também os loops.

Uma boa dica é continuar testando os nossos códigos no IRB, e também experimentando suas curiosidades para obter diferentes resultados.

2.1 MAIS TIPOS NO RUBY

Como vimos no capítulo anterior, a declaração de variáveis em Ruby é um processo bastante simples, bastando escolher o nome da sua variável e atribuir um valor para ela. Esteja atento, pois deve ser respeitada a regra de começá-las com letras, `$` ou `_`.

```
1 = "Lucas" # não funciona
nome = "Lucas" # funciona
$nome = "Lucas" # funciona
_nome = "Lucas" # funciona
```

Mas e quando nossa variável possui nomes compostos? Em algumas linguagens, é convencionado que cada palavra que formará o identificador da variável comece com letra maiúscula, exceto a primeira, por exemplo, `telefoneCelular`, ou então `numeroDeRegistro`. Essa convenção é conhecida como *Camel Case*. Em Ruby, utilizamos outra convenção para separar as palavras. Nela, usamos sempre letras minúsculas e como separador o `_`, dessa forma poderíamos ter a variável `telefone_celular`.

```
telefone_celular = "(11) 91234-5678"
```

2.2 COMENTE SEU CÓDIGO

Se você estava atento, deve ter reparado que algumas vezes utilizamos uma `#` (cerquilha) no código que mostrava os diferentes exemplos de declaração de variável em Ruby. Estávamos na verdade fazendo um comentário. Esse é o conhecido comentário de 1 linha. Assim, se existirem N linhas e você desejar comentar todas elas, terá de adicionar `#` linha por linha.

```
#idade = 27
#ano = 2013
```

Para esses casos em que precisamos comentar diversas linhas,

podemos utilizar os comentários em bloco, que podem ser feitos envolvendo o conteúdo a ser comentado entre as demarcações `=begin` e `=end`.

```
=begin
  idade = 27
  ano = 2013
=end
```

Com isso, esses códigos serão desconsiderados na interpretação do seu programa.

2.3 O TRABALHO COM NÚMEROS

Em qualquer aplicação que é desenvolvida, em algum momento é preciso trabalhar com números. Muitas das informações podem ser representadas com um número inteiro, como `idade`, `ano` ou `quantidade_estoque`, por exemplo. Nesse caso, declaramos a variável com o número:

```
idade = 27
ano = 2013
```

Ao trabalharmos com números inteiros, podemos descobrir seu tipo perguntando para a variável qual é a sua classe:

```
idade.class
=> Fixnum
```

Repare que `=> Fixnum` não precisa ser digitado, ele apenas indica qual será a saída do IRB. Ao longo do livro, utilizarei esta abordagem para indicar qual o retorno de uma chamada dentro do IRB, facilitando sua leitura.

Números inteiros geralmente são do tipo `Fixnum`, exceto em casos extremamente grandes em que o `Bignum` é empregado. No

entanto, números muito grandes e que precisamos representar de forma literal podem ficar complicados de ler no código. Por exemplo, para a quantidade de habitantes no mundo em 2013, poderíamos fazer:

```
habitantes = 7000000000
```

O número 7 bilhões de habitantes é consideravelmente grande. Mas e se fossem 70 bilhões?

```
habitantes = 700000000000
```

Repare que não é tão fácil perceber que há um zero a mais nesse grande número. Para esses casos, Ruby nos permite separar os milhares através do `_` :

```
habitantes = 7_000_000_000
```

Com isso, conseguimos deixar o número literal mais legível.

Em outras situações, podemos precisar definir números que tenham casas decimais. Por exemplo, para representar o peso de uma pessoa:

```
peso = 77.9
```

Repare que a separação das casas decimais é o `.` (ponto), e não a `,` (vírgula).

2.4 REPRESENTAÇÃO DE TEXTOS COM AS STRINGS

Muitas das vezes precisamos de variáveis que guardem informações que são compostas por texto, como o nome de uma pessoa e seu identificador em alguma rede social. Esse tipo de

informação é o que chamamos de `String` . Em Ruby, qualquer caractere ou conjunto de caracteres cercado de aspas simples ou duplas é considerado uma `String` :

```
nome_completo = "Lucas Souza"
twitter = '@Lucasas'

puts nome_completo.class # => String
puts twitter.class # => String
```

Mas por qual motivo e quando eu uso `String` com aspas simples ou duplas? Imagine que você possui uma amiga chamada Joana d'Arc e precisamos armazenar o nome dela em uma variável:

```
nome_com_aspas_simples = 'Joana d'Arc' # não funciona
nome_com_aspas_duplas = "Joana d'Arc" # funciona
```

Repare que, quando definimos `String` com aspas simples, não conseguimos adicionar outra aspa simples dentro do texto, pois dessa forma o interpretador não sabe onde fica o final da `String` . Neste caso, o único jeito é usar aspas duplas.

Mas existem outros casos nos quais `Strings` com aspas duplas são mais interessantes. Precisamos agora exibir uma mensagem de boas-vindas contendo o nome da pessoa que está definido dentro da variável `nome` . O mais natural seria repetir o comportamento de outras linguagens e usar o operador `+` (mais) para realizar uma concatenação.

```
nome = "Joana d'Arc"
boas_vindas = "Seja bem-vinda(o) " + nome
puts boas_vindas # => Seja bem-vinda(o) Joana d'Arc
```

Porém, em Ruby, quando precisamos concatenar duas `String` , preferimos fazer o uso da interpolação:

```
nome = "Joana d'Arc"
```

```
boas_vindas = "Seja bem-vinda(o) #{nome}"  
puts boas_vindas # => Seja bem-vinda(o) Joana d'Arc
```

Basta colocar dentro da `String` a variável dentro de `# {variavel}`. É uma maneira mais elegante e comum de ser usada em códigos Ruby. O detalhe é que, em `Strings` definidas com aspas simples, não é possível fazer uso da interpolação.

Prefira sempre o uso de `String` com aspas duplas e priorize sempre a interpolação quando for concatenar.

Existe diferença de performance entre `String` com aspas simples e duplas?

Existe muita discussão sobre qual tipo de declaração de `String` devemos utilizar, com o argumento de que as declarações com aspas simples são mais rápidas do que as declarações com aspas duplas.

As `String` declaradas com aspas simples podem ser sutilmente mais rápidas que as declaradas com aspas duplas, porque o analisador léxico da linguagem Ruby não tem de checar se existem marcadores de interpolação `#{}` . Este número pode variar de interpretador para interpretador, e vale ressaltar que este tempo seria menos durante o parser do código, e não durante a sua execução.

A diferença é tão insignificante que não vale a pena perdermos tempo com esse tipo de comparação. Eu particularmente prefiro declará-las sempre com aspas duplas, porque, se em determinado momento eu precisar interpolar o valor de alguma variável dentro desta `String`, ela já foi criada de uma maneira que não preciso alterá-la.

A única preocupação que devemos ter é com o que vamos interpolar dentro de uma `String`. Por exemplo:

```
puts 'mensagem' # => mensagem
puts "#{sleep 1}mensagem" # => mensagem
```

A `String` declarada com aspas duplas interpola a chamada de um método da classe `Kernel` que interrompe a execução do código por 1 segundo. Ou seja, o tempo para impressão da segunda `String` `mensagem` será de pelo menos 1 segundo.

2.5 ESTRUTURAS DE CONTROLE

Ruby possui as principais estruturas de controle (`if`, `while` etc.) assim como as linguagens Java, C e Perl. Porém, ao contrário destas que usam chaves `{` e `}` para definir o conteúdo da estrutura, em Ruby usa-se a palavra reservada `end` apenas para finalizar o corpo da estrutura.

Supondo que desejamos imprimir conteúdo da variável `nome` apenas se a `idade` for maior que `18`, usamos a estrutura `if`, que em Ruby possui a seguinte sintaxe:

```
idade = 27
nome = "Lucas"

if(idade > 18)
  puts nome # => Lucas
end
```

Uma maneira de deixar o código Ruby ainda mais simples é removendo os parênteses da chamada do `if`.

```
idade = 27
nome = "Lucas"
```

```
if idade > 18
  puts nome # => Lucas
end
```

Uma das vantagens do Ruby é que, na maioria das vezes, podemos omitir o uso dos parênteses. Isso é o que chamamos de *Syntax Sugar* (ou "açúcar sintático") da linguagem, que visa deixar o código mais legível.

Falando em deixar o código mais legível, se o corpo do seu `%if` possuir apenas uma linha, prefira uma sintaxe mais enxuta:

```
idade = 27
nome = "Lucas"

puts nome if idade > 18 # => Lucas
```

Repare que parece que estamos lendo um texto em inglês: "Imprima nome se a idade for maior que 18". Esta é uma das grandes vantagens da linguagem: maior legibilidade sempre que possível.

2.6 ENTENDA O VALOR NULO

Quando desejamos representar algum valor vazio em Ruby, usamos a palavra reservada `nil`. O `nil` não representa uma `String` vazia ou o número zero, ele representa um valor vazio, um espaço vazio. Quando atribuímos `nil` a uma variável, queremos dizer que ela não possui nenhum valor.

```
caixa = nil
```

A nossa variável `caixa` não possui nada dentro dela, ou seja, é vazia.

Neste contexto, podemos imprimir uma mensagem de boas-

vindas caso o conteúdo da variável `nome` tenha algum valor não nulo usando o método `nil?` :

```
nome = "Lucas"
puts "Seja bem-vindo #{nome}" if not nome.nil? #
=> Seja bem-vindo Lucas
```

Se executarmos esse código, a mensagem *'Seja bem-vindo Lucas'* será exibida. Mas e no caso de a variável possuir o valor `nil` ?

```
nome = nil
puts "Seja bem-vindo #{nome}" if not nome.nil?
```

Nesse caso, nenhuma mensagem será exibida: a variável `nome` é nula e o método `nil?` retorna `true` . Como fazemos a negação usando o `not` , o valor é invertido e, portanto, `false` .

Um pouco complicado, não é? Você vai se acostumar rapidamente.

2.7 SUBSTITUA O "IF NOT" POR "UNLESS"

Podemos simplificar o código anterior usando a condicional negada, o `unless` ("a menos que", no bom português).

```
nome = nil
puts "Seja bem-vindo #{nome}" unless nome.nil?
```

Podemos ler o código: *"Imprima 'Seja bem-vindo ...' a menos que o nome seja nulo"*. Na maioria das vezes que implementamos um `if not` , ele pode ser convertido para um `unless` .

E se eu disser que podemos melhorar ainda mais o nosso código? Interessante, não? Pois é, existe algo que você ainda não sabe sobre as variáveis com valor `nil` .

Se usadas dentro de condicionais como `if` e `unless`, a variável quando `nil` assume automaticamente o valor `false` e, no caso contrário, assume o valor `true`.

```
nome = nil
puts "Seja bem vindo #{nome}" if nome
```

A variável `nome` possui o valor `nil` e assume o valor `false` na condicional anterior, sendo assim nenhuma mensagem é impressa no terminal. Se a variável possuir algum valor não `nil`:

```
nome = "Lucas"
puts "Seja bem-vindo #{nome}" if nome #
=> Seja bem-vindo Lucas
```

A mensagem *'Seja bem-vindo Lucas'* é impressa, pois a variável não é `nil` e, portanto, assume o valor `true`.

2.8 ITERAÇÕES SIMPLES COM FOR, WHILE, UNTIL

For

Existem diversas formas de iterar um determinado número de vezes por um código Ruby. Como em outras linguagens, existem os conhecidos `while`, `until` e `for`. Sem dúvida, a maneira mais usada em códigos Ruby com a qual você vai se deparar ao longo do tempo é o `for`.

Desejamos imprimir os números de 1 até 100. Apenas adicionaremos uma mensagem *'Número: X'* para deixarmos nossas mensagens mais elegantes.

```
for numero in (1..100)
  puts "Número: #{numero}"
```

end

Esse código atribui à variável `numero` os valores de 1 até 100 e executa a conteúdo do `for`, que termina quando usamos a palavra reservada `end`. O detalhe mais importante do código é `(1..100)`, que cria um range de número de 1 até 100, que é exatamente o número de vezes que desejamos executar a impressão de um número.

While

Podemos iterar de 1 até 100 imprimindo cada um dos números, usando a estrutura de repetição `while`, que, assim como em linguagens tradicionais como Java e C, executa um bloco de código até que uma determinada condição seja falsa (conteúdo que também é delimitado pela palavra reservada `end`).

```
numero = 0
while numero <= 100
  puts "Numero: #{numero}"
  numero += 1
end
```

Quando `for` executado, esse código executará a impressão da mensagem *'Numero: x'*, imprimindo de 1 até 100, quando a condição do `while` for `false`.

Until

Ao contrário do `while`, que termina sua execução quando uma condição falsa é alcançada, a estrutura de repetição `until` executa um determinado bloco de código até que uma condição verdadeira seja encontrada:

```
numero = 0
```

```
until numero == 100
  puts "Numero: #{numero}"
  numero += 1
end
```

A diferença é que esse código executará a impressão da mensagem 'Numero: *x*' de 1 até 100, até que o valor da variável `numero` seja 100 e ocorra o término na execução do `until`.

2.9 AS OUTRAS FORMAS DE DECLARAR STRINGS

Aprendemos e discutimos duas formas de declarar `String` :

```
aspas_simples = 'linguagem_ruby'
aspas_duplas = "linguagem_ruby"
```

Mas como proceder caso precisemos declarar `String` com aspas duplas e aspas simples dentro, por exemplo: *"Isso é "normal" e 'útil' no mundo Ruby"*? Nenhuma das duas abordagens anteriores resolvem o problema sem o uso do caractere de escape `\`.

```
string_especial_usando_aspas_simples =
  'Isso é "normal" e \'util\' no mundo Ruby'
string_especial_usando_aspas_duplas =
  "Isso é \"normal\" e 'util' no mundo Ruby"

puts string_especial_usando_aspas_simples
# => 'Isso é "normal" e \'util\' em Ruby'

puts string_especial_usando_aspas_duplas
# => "Isso é \"normal\" e 'util' em Ruby"
```

Existe uma notação na linguagem Ruby, inspirada no Perl, que permite declararmos este tipo de `String` especial. Basta fazê-lo da seguinte forma:

```
string_especial = %{Isso é "normal" e 'util' no mundo Ruby}
```

```
puts string_especial  
# => "Isso é \"normal\" e 'util' em Ruby"
```

Na verdade, qualquer caractere não alfanumérico pode ser usado após o %, por exemplo:

```
string_especial = %[Isso é "normal" e 'util' no mundo Ruby]  
puts string_especial  
# => "Isso é \"normal\" e 'util' em Ruby"
```

```
string_especial = %?Isso é "normal" e 'util' no mundo Ruby?  
puts string_especial  
# => "Isso é \"normal\" e 'util' em Ruby"
```

```
string_especial = %~Isso é "normal" e 'util' no mundo Ruby~  
puts string_especial  
# => "Isso é \"normal\" e 'util' em Ruby"
```

Obviamente, o caractere usado para delimitar a String deve ser escapado com \, caso apareça dentro do texto que está sendo declarado:

```
string_especial = %{Isso é "normal" e \{util no mundo Ruby}  
puts string_especial  
# => "Isso é \"normal\" e {util em Ruby"
```

Entretanto, se você utilizar como delimitador os caracteres (parenteses) , [colchetes] , {chaves} ou <menor e maior> , eles podem aparecer dentro da String sem serem escapados, caso sejam usados em pares (diferente do exemplo dado anteriormente):

```
string_especial = %{Isso é "normal" e {util} no mundo Ruby}  
puts string_especial  
# "Isso é \"normal\" e {util} em Ruby"
```

Esta forma de declaração de String possui algumas variações, que podem, por exemplo, adicionar a capacidade de interpolar variáveis.

- %q : não permite interpolação;
- %Q : permite interpolação.

Existem algumas outras variações que vamos aprender ao longo do livro. Esta maneira de declarar `String` permite que sejam elas também criadas com múltiplas linhas:

```
string_especial = %{Isso é "normal" e {util} no mundo Ruby
                  e a partir de agora veremos a 'todo' momento}
puts string_especial
```

Repare que uma quebra de linha (`\n`) é inserida exatamente no lugar onde quebramos a linha em nosso código. Essa característica é muito útil quando precisamos criar `String` grandes e precisamos deixá-las mais legíveis.

2.10 PRÓXIMOS PASSOS

Neste capítulo, aprendemos boas convenções de código nas declarações de variáveis, alguns outros tipos de dados existentes na linguagem Ruby, estruturas de controle como `if` e `unless` e as melhores formas de usá-las, a fim de diminuir ruídos na sintaxe. Vimos também as conhecidas e úteis estruturas de repetição `for` , `while` e `until` .

Aprendemos também as diferentes formas de declararmos `String` , com variações e suporte a múltiplas linhas. Vimos o interessante valor `nil` , que será muito utilizado durante todo o restante do livro.

O importante é lembrar de que estes conceitos são a base que precisamos para seguir em frente e aprendermos recursos mais avançados da linguagem. Portanto, não deixe dúvidas para trás: se

preciso, volte e leia novamente partes que você julga necessário antes de prosseguirmos.

O COMEÇO DA NOSSA APLICAÇÃO

Nos capítulos anteriores, aprendemos algumas funcionalidades básicas da linguagem Ruby. E, como já vimos, uma das suas principais características é ser orientada a objetos. Neste capítulo focaremos na criação de classes, métodos e objetos usando todo o poder que a linguagem nos proporciona.

Começaremos construindo uma aplicação que servirá de base para os próximos capítulos do livro. Portanto, é interessante que você faça os códigos e entenda os conceitos explicados neste capítulo.

Existem vários assuntos que poderíamos abordar e, neste livro, construiremos uma aplicação para controlar uma loja de livros. Controlaremos o seu estoque, quais são os clientes da loja, faremos algumas vendas e guardaremos todas estas informações em disco, utilizando a API de arquivos do Ruby.

A grande pergunta que sempre fica quando criamos uma aplicação é: por onde começar? Temos muitas opções. Podemos iniciar o desenvolvimento pelo domínio (criando classes), pela interface do usuário, pelo banco de dados e assim por diante. No

nosso caso, começaremos desenvolvendo as classes de domínio da aplicação.

3.1 A DEFINIÇÃO DA CLASSE LIVRO

Se vamos construir uma aplicação que gerencia uma loja de livros, precisamos manter os dados dos nossos livros em algum lugar. Pensando brevemente neles, teremos:

```
nome = "Linguagem Ruby"  
isbn = "123-45678901-2"  
numero_paginas = 245  
preco = 69.90
```

Todos os livros da nossa loja estão com um desconto pré-definido:

```
desconto = 0.1
```

Agora precisamos saber qual é o preço do nosso livro com desconto. Uma conta bem simples pode nos devolver qual o valor com o desconto:

```
nome = "Linguagem Ruby"  
isbn = "342-65675756-1"  
numero_paginas = 245
```

```
preco = 69.90  
desconto = 0.1
```

```
preco_com_desconto = preco - (preco * desconto)
```

O problema deste código, todo espalhado, é que, quando precisarmos calcular o desconto de qualquer livro, teremos de repetir a mesma operação. Se em algum momento esta lógica precisar ser alterada, vamos ter de mudar em N lugares do nosso código.

Podemos voltar um pouco no tempo e imitar as antigas e clássicas funções que temos em linguagens como C.

```
def preco_com_desconto(preco, desconto)
  preco - (preco * desconto)
end
```

Agora, toda vez que precisarmos efetuar o cálculo do preço com desconto, chamamos a função `preco_com_desconto` passando o `preco` e o `desconto` :

```
nome = "Linguagem Ruby"
isbn = "342-65675756-1"
numero_paginas = 245

preco = 69.90
desconto = 0.1

puts preco_com_desconto(preco, desconto) # => 62.90
```

Caso haja necessidade de alterar algum outro livro, devemos criar as variáveis referentes às informações deste outro livro, e então chamar o método `preco_com_desconto` para sabermos qual o novo valor:

```
nome = "Test Driven Development: Teste e Design no Mundo Real"
isbn = "342-65675751-1"
numero_paginas = 212

preco = 89.90
desconto = 0.1

puts preco_com_desconto(preco, desconto) # => 80.90
```

O que temos aqui é um típico caso em que os dados e a sua manipulação estão separados, quando deveriam estar juntos. Esse é um dos princípios por trás da Orientação à Objetos, unir dados e a manipulação dos mesmos, encapsulando o acesso. A maneira de fazermos isso é colocando este código dentro de uma classe e

representar cada um dos livros da nossa loja como uma instância desta classe.

Para criar classes usando Ruby, basta usar a palavra reservada `class` e delimitar o final da sua classe com a palavra reservada `end`.

```
class Livro  
end
```

CONSTANTES

O nome de uma classe deve obrigatoriamente começar com uma letra maiúscula e as outras minúsculas. Por exemplo: `Livro`. Repare que o nome `Livro` não é uma `String`, já que não está entre aspas ou dentro do delimitador de `String`. O nome `Livro` é uma constante, que definimos quando criamos a nossa classe.

Todos os tipos existentes no Ruby como `String`, `Integer`, `Time` etc. são constantes. Caso o nome da sua constante seja composto, usamos a notação *camel case*, por exemplo: `UnidadeDeEstoque`.

O que criamos foi uma abstração de um `Livro`, ou seja, um *template* que dirá do que um livro é composto. A partir da classe, que é o *template*, precisamos criar um novo livro, no qual poderemos dizer quais são suas características, como título, ISBN, autor e assim por diante. Para criarmos cada um, usaremos a palavra `new`:

```
teste_e_design = Livro.new
web_design_responsivo = Livro.new
```

Quando usamos o `new` em uma classe, estamos criando uma instância dela (dizemos também que estamos criando um objeto daquele tipo). No código anterior, criamos duas instâncias e as referenciamos através das variáveis `teste_e_design` e `web_design_responsivo`. Nós não provemos nenhum tipo de informação sobre estes livros, como o nome do autor, ISBN, quantidade de páginas etc. Esse é um dos princípios da Orientação a Objetos, guardar dados sobre uma determinada entidade do nosso sistema.

A melhor maneira de provermos estas informações é no momento que criamos nossos objetos do tipo `Livro`. Para isso, existe o método `initialize` que é chamado toda vez que executamos o método `new` a fim de criar um objeto.

```
teste_e_design = Livro.new("Mauricio Aniche", 247, "123454")
web_design_responsivo = Livro.new("Tárcio Zemel", 189, "452565")
```

Podemos até mesmo remover os parênteses:

```
teste_e_design = Livro.new "Mauricio Aniche", 247, "123454"
web_design_responsivo = Livro.new "Tárcio Zemel", 189, "452565"
```

Quando criamos uma classe, ganhamos automaticamente um método `initialize default`, para que o objeto possa ter suas informações inicializadas. Como queremos passar dados para inicializar nosso objeto de uma maneira diferente, no caso passando o nome do autor, número de páginas e ISBN, devemos implementar nosso próprio método `initialize` em nossa classe `Livro`:

```
class Livro
  def initialize(autor, numero_de_paginas, isbn)
```

```
end  
end
```

Sobrescrevemos o método `initialize` default, criando um **método** cujo nome deve ser `initialize`. O `initialize` é um método especial do Ruby, invocado pelo `Livro.new` para criar um novo objeto. Nesse momento, o Ruby aloca espaço em memória, e depois invoca o método `initialize` passando os parâmetros necessários para criar o objeto.

Agora vamos imaginar que existem alguns livros que não possuem o atributo `isbn`, o natural seria omitir o valor do ISBN:

```
teste_e_design = Livro.new "Mauricio Aniche", 247
```

Isso não funciona, pois, para a linguagem Ruby, a aridade do método é importante, ou seja, devemos passar exatamente a quantidade de argumentos definida no método `initialize`.

Se pensarmos em linguagens como Java, podemos definir um outro método `initialize` que recebe apenas o nome do autor e o número de páginas, recurso conhecido como *sobrecarga*:

```
class Livro  
  def initialize(autor, numero_de_paginas, isbn)  
    end  
  
  def initialize(autor, numero_de_paginas)  
    end  
end
```

Mas isso também não funcionará, pois o interpretador do Ruby considera apenas o último método definido na classe. Ruby não suporta sobrecarga. Para essa situação, a linguagem suporta o uso de argumentos com valores padrão:

```
class Livro
```

```

def initialize(autor, numero_de_paginas, isbn = "1")
end
end

```

Quando declaramos o argumento `isbn = "1"` , definimos que no momento da inicialização dos objetos do tipo `Livro` , podemos omitir o último argumento:

```
Livro.new "Lucas Souza", 198
```

O valor do argumento `isbn` é "1".

Agora vamos inverter a ordem dos argumentos do método `initialize` , colocando o atributo `isbn` antes do atributo `numero_de_paginas` . Também imprimiremos os valores de cada um dos argumentos:

```

class Livro
  def initialize(autor, isbn = "1", numero_de_paginas)
    puts "Autor: #{autor}, ISBN: #{isbn},
        Pág: #{numero_de_paginas}"
  end
end

Livro.new "Lucas Souza", 200

```

E o resultado é: Autor: Lucas Souza, ISBN: 1, Páginas: 200 . Veja que o Ruby é esperto e consegue atribuir o valor 200 ao atributo `numero_de_paginas` , e não ao atributo `isbn` . Isso ocorre porque a linguagem Ruby possui três tipos de argumentos:

- Obrigatórios;
- Com valores padrão;
- Opcionais.

Vimos dois deles até o momento: obrigatório e com valores padrão. Veremos sobre os atributos opcionais nos próximos

capítulos. Mas vamos ao que interessa, como o Ruby consegue descobrir para qual atributo deve atribuir os valores da chamada ao método `initialize` ?

Quando um método recebe a chamada, o interpretador divide a atribuição dos valores em alguns passos. Primeiro ele procura todos argumentos obrigatórios e associa os valores a estes, e caso não encontre valor para algum destes atributos, um erro do tipo `ArgumentException` acontece. Em seguida, se ainda existirem valores sobrando, estes são atribuídos aos argumentos *default* que existirem no método. Caso sobre apenas 1 valor e existirem dois argumentos com valores default, o Ruby atribuirá o valor restante para o primeiro argumento com valor default.

3.2 CRIE A ESTRUTURA DO PROJETO

O código final do projeto que será construído foi disponibilizado no meu GitHub: <https://github.com/lucasas/projeto-ruby>

Agora, é importante que seja criada a estrutura básica do projeto, que facilitará a alteração das classes que serão criadas ao longo dos capítulos. O ideal é que somente os testes sejam feitos utilizando o IRB, e que todas as outras classes e códigos fiquem dentro do projeto.

Crie uma pasta chamada `loja_virtual` em um diretório de sua preferência e, dentro deste diretório `loja_virtual`, crie uma pasta chamada `lib`. Dentro do diretório `lib`, serão criados os arquivos que representam as classes do sistema, como por exemplo, a classe `Livro` que podemos transferir para lá agora

mesmo.

Crie um arquivo chamado `livro.rb` dentro da pasta `lib` , com o conteúdo atual que definimos no IRB:

```
class Livro
  def initialize(autor, isbn = "1", numero_de_paginas)
    puts "Autor: #{autor}, ISBN: #{isbn},
        Pág: #{numero_de_paginas}"
  end
end
```

Agora que a classe `Livro` foi colocada em um arquivo separado, precisamos alterar a maneira que os testes desta classe serão feitos. Quando um IRB é aberto, o arquivo `lib/livro.rb` não é carregado automaticamente. Você pode validar tentando criar um objeto `Livro` abrindo um novo terminal:

```
Livro.new "Lucas Souza", 200
# => NameError: uninitialized constant Livro
```

Para carregar o conteúdo do arquivo `lib/livro.rb` , é necessário utilizar o método `require` da classe `Kernel` . O método `require` recebe como parâmetro uma `String` que pode ser o nome do arquivo `.rb` que você deseja carregar. Se o parâmetro não for o caminho absoluto do arquivo `.rb` , ele será procurado em diretórios que estão definidos na constante `$LOAD_PATH` .

O arquivo `lib/livro.rb` não está em nenhum dos diretório listados em `$LOAD_PATH` , que em geral contém os diretórios onde estão os arquivos `.rb` das classes core do Ruby. Sendo assim, temos dois caminhos para fazer com que o Ruby consiga carregar o arquivo que contém a classe `Livro` . O primeiro é adicionar na constante `$LOAD_PATH` , que é um objeto `Array` , o diretório

lib do projeto loja_virtual :

```
$LOAD_PATH << "caminho relativo do projeto loja_virtual/lib"
```

Ao executar este código, o `require` pode ser feito apenas com o nome do arquivo da pasta `lib` que desejamos carregar:

```
require 'livro'
```

A segunda opção (e a melhor na minha opinião) é carregar o arquivo passando o seu caminho absoluto. Porém, isso pode ser um pouco custoso, já que você pode simplesmente resolver alterar o nome da pasta de `loja_virtual` para `loja`. O ideal é utilizar o método `expand_path` da classe `File`, que retorna o caminho absoluto de um nome de arquivo passado como parâmetro, levando em consideração o diretório no qual a chamada do método é executada.

Supondo que o arquivo `livro.rb` esteja dentro da pasta `lib`, que por sua vez está dentro do diretório `/home/lucas/loja_virtual`, ao executar a chamada ao método `expand_path` dentro deste diretório passando a String `lib/livro`, o retorno do método será `/home/lucas/loja_virtual/lib/livro`.

```
# executando a partir do diretório /home/lucas/loja_virtual
puts File.expand_path("lib/livro")
# => "/home/lucas/loja_virtual/lib/livro"
```

Exatamente o diretório onde se encontra o arquivo `livro.rb`.

Agora para testar a classe `Livro` como fizemos na seção anterior, basta abrir um novo IRB no diretório `loja_virtual` e invocar o método `require`, passando o caminho absoluto do arquivo `livro.rb`:


```
# executando a partir do diretório /home/lucas/loja_virtual
require File.expand_path("lib/livro")

# objeto Livro criado com sucesso
Livro.new "Lucas Souza", 200
```

A partir de agora, todas as novas classes serão colocadas dentro de arquivos `.rb` na pasta `lib`. E os testes serão feitos no IRB, lembrando de que deve ser sempre aberto a partir do diretório `loja_virtual`. E claro, é necessário carregar, utilizando o método `require`, todas as classes que serão usadas no teste.

Outra dica importante: quando for necessário criar classes novas, haverá uma indicação dos procedimentos que deverão ser feitos para que esta possa ser testada. Quando houver alteração em classes já existentes, basta editar o conteúdo da classe dentro de seu respectivo arquivo `.rb`.

E lembre-se: o projeto final com a implementação completa pode ser encontrado no GitHub: <https://github.com/lucasas/projeto-ruby>.

ENCODING ARQUIVOS .RB

Os arquivos que contém as classes criadas em nosso sistema ficarão em arquivos `.rb` dentro de um diretório de sua preferência. Porém, é importante ressaltar que arquivos `.rb` possuem um encoding US-ASCII por padrão. Caso seu código contenha qualquer caractere que não seja compatível com o ASCII, a interpretador Ruby será finalizado e acusará o erro: `invalid multibyte char (US-ASCII)`.

Se você quiser alterar o encoding padrão do arquivo `.rb`, basta adicionar a seguinte linha do arquivo:

```
# encoding: utf-8
```

Neste exemplo, ajustamos o encoding do arquivo para UTF-8, que permitirá que você use acentos e outros caracteres.

3.3 DEFINA OS ATRIBUTOS DE INSTÂNCIA

Os parâmetros passados para o método `initialize` são na verdade variáveis locais. Isto é, assim que terminar a execução do método, as variáveis locais simplesmente desaparecem.

Vamos precisar das informações do autor, número de páginas e ISBN para trabalhar com elas mais adiante nos nossos programas. Queremos que a informação dos livros acompanhem o objeto que vai ser instanciado. Para resolver este problema, precisamos copiar o valor destes parâmetros para variáveis de instância de cada objeto. Este comportamento é muito comum

quando criamos métodos construtores.

Então, vamos copiar os valores dos parâmetros do método `initialize` para variáveis de instância:

```
class Livro
  def initialize(autor, isbn = "1", numero_de_paginas)
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
  end
end
```

Repare que as variáveis de instância têm um caractere `@` (arroba) antes do nome. O importante aqui é percebermos que as variáveis podem ter nomes iguais, porém, variáveis com `@` são de instância e compõem o estado interno do objeto que está sendo criado, enquanto variáveis locais possuem um escopo mais curto, elas duram apenas até o término da execução do método.

VISIBILIDADE DAS VARIÁVEIS DE INSTÂNCIA

Em Ruby, todas as variáveis de instância criadas são privadas, ou seja, não possuem acesso externo, nem para leitura, nem para escrita. Se quisermos acessá-las, usaremos os recursos da própria linguagem, que veremos nas próximas seções.

Vamos testar o método `initialize` que agora guarda os parâmetros em variáveis de instância:

```
teste_e_design = Livro.new("Mauricio Aniche", "123454", 247)
web_design_responsivo = Livro.new("Tárcio Zemel", "452565", 321)

p teste_e_design
```

O código anterior produz a seguinte saída:

```
#<Livro:0x007fa526012048 @autor="Mauricio Aniche",  
  @isbn="123454", @numero_de_paginas=247>  
#<Livro:0x007fa526012049 @autor="Tárcio Zemel",  
  @isbn="452565", @numero_de_paginas=321>
```

PUTS OU P?

Quando precisamos imprimir informações no console em Ruby, podemos utilizar os métodos `puts` ou `p`. O primeiro deles já havíamos visto, o método `puts`. No exemplo de código onde testamos nossas variáveis de instância, usamos o método `p`.

O método `puts` imprime o retorno do método `to_s` do objeto que foi passado para ser impresso. Isto é, caso você faça `puts variavel`, é como se ele mostrasse o valor de `variavel.to_s`, método que veremos logo em seguida. Já o método `p` é mais usado quando queremos realizar o *debug* do conteúdo do objeto passado como argumento. Nesse caso, o método `inspect` do objeto é invocado e o retorno é impresso na tela. Quando o conteúdo das variáveis de instância é impresso, é porque o método `inspect` da classe `Livro` retorna os valores de todas as variáveis do objeto criado.

3.4 SOBRESCREVENDO O MÉTODO TO_S

Convenhamos que o resultado impresso não é o ideal. Podemos retornar uma mensagem mais agradável caso algum usuário do nosso código instancie e queira ver informações mais coerentes sobre uma instância de `Livro`.

Aliás, exibir uma mensagem amigável ao 'imprimir' um objeto é considerado uma boa prática. Esse assunto é inclusive capítulo de um livro muito conhecido na comunidade Java, chamado **Effective Java**, escrito por Joshua Bloch, hoje engenheiro da Google.

Em Ruby, podemos aplicar a mesma prática sobrescrevendo o método `to_s`, que é herdado naturalmente por todas as classes, pois faz parte da classe `Object`. Então, fazemos com que ele devolva uma `String`:

```
# coding: utf-8
class Livro
  def initialize(autor, isbn = "1", numero_de_paginas)
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
  end

  def to_s
    "Autor: #{@autor}, ISBN: #{@isbn},
      Páginas: #{@numero_de_paginas}"
  end
end
```

Uma novidade nesse código é que, em Ruby, quando desejamos retornar um determinado valor, não precisamos explicitamente colocar a palavra `return` antes do valor que desejamos retornar. O retorno de qualquer método escrito em Ruby sempre será a última instrução de código (veremos este comportamento com mais detalhes no capítulo *Ruby e a programação funcional*).

Se a última instrução de um método for, por exemplo, a `String` `"Um texto qualquer"`, ao invocarmos este método, receberemos como seu retorno a própria `String` `Um texto qualquer`. Porém, em algumas ocasiões, o uso da palavra `return` é necessário, dependendo da lógica que estamos implementando, mas lembre-se de que ele **nunca** é obrigatório.

Lembre-se de que o método `to_s` é invocado quando usamos o método `puts`. Portanto, vamos alterar o código que testa a criação dos objetos com suas variáveis de instância:

```
teste_e_design = Livro.new("Mauricio Aniche", "123454", 247)
web_design_responsivo = Livro.new("Tárcio Zemel", "452565", 321)

puts teste_e_design
puts web_design_responsivo
```

Agora o código anterior produz a seguinte saída:

```
Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247
Autor: Tárcio Zemel, Isbn : 452565, Páginas: 321
```

A mensagem está muito mais elegante e concisa. Lembre-se: sobrescrever o método `to_s` sempre é uma boa prática quando queremos criar uma mensagem mais elegante que descreva o estado interno dos objetos.

3.5 ALTERAÇÃO E LEITURA DE ATRIBUTOS

Para que nossos livros possam ser vendidos, precisamos de que eles tenham um atributo que contém seu preço. Para isso, adicionaremos um novo argumento no método `inicialize` da classe `Livro`, e guardaremos seu valor em uma variável de instância chamada `@preco`.

```
# coding: utf-8
class Livro
  def initialize(autor, isbn = "1", numero_de_paginas, preco)
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @preco = preco
  end

  def to_s
    "Autor: #{@autor}, ISBN: #{@isbn},
    Páginas: #{@numero_de_paginas}"
  end
end
```

E podemos instanciar novos livros informando o preço:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
```

Agora para todos os objetos do tipo `Livro` que foram criados, desejamos também saber seu preço. Mas lembre-se de que as variáveis de instância são sempre **privadas**, ou seja, só conseguimos acessá-las dentro da classe `Livro`. A solução neste caso é criar um método **público** que retorne o valor da variável `@preco`:

```
# coding: utf-8
class Livro

  # outros métodos

  def preco
    @preco
  end
end
```

Agora, dada qualquer instância de um `Livro`, podemos invocar o método `preco` que retornará o valor da variável `@preco`:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
```

```
puts teste_e_design.preco # => 60.9
```

Repare que fizemos um método com o mesmo nome da variável. Essa é uma convenção utilizada pelos desenvolvedores Ruby, em que o método que acessa a variável e a própria variável possuem o mesmo nome.

O Brasil é um país que possui inflação que gira em torno de 6% a 7% ao ano. Em nossa aplicação, permitiremos que os nossos livros sofram alterações de preços também. Dessa forma, dada uma instância do tipo `Livro`, desejamos alterar seu preço:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
teste_e_design.preco = 79.9
# => NoMethodError: undefined method 'preco=' for
      Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247:Livro
```

Lembre-se novamente de que as variáveis de instância são sempre **privadas**. Então, por enquanto, não conseguimos alterar o preço de um objeto `Livro`. A solução é criarmos um método **público** que recebe o novo preço do livro como argumento e atribua este preço à variável que guarda o valor do nosso livro, `@preco`.

```
# coding: utf-8
class Livro

  # outros métodos

  def preco
    @preco
  end

  def preco=(preco)
    @preco = preco
  end
end
```


Podemos ver outra convenção para criar código Ruby mais legível. Os métodos que alteram o valor de variáveis de instância costumam ter o nome da variável sem o @ , com um sinal de = , e a declaração dos argumentos do método. Mas lembre-se de que isso é uma convenção.

Para alterar o preço de uma instância do tipo `Livro` , podemos agora invocar o método `preco=` :

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
teste_e_design.preco=(79.9)

puts teste_e_design.preco
```

Ao executarmos esse código, veremos que o novo preço do livro é 79.9 . Mas podemos deixar nosso código mais simples. Remover os parênteses é um bom caminho para melhorar a legibilidade:

```
teste_e_design.preco=79.9
```

Quando definimos métodos que possuem sinal de = em sua nomenclatura, podemos adicionar um espaço na chamada do método:

```
teste_e_design.preco = 79.9
```

Diferença sutil, mas que deixa a sensação de que estamos alterando o valor da variável diretamente. Nosso código está legível, mas podemos melhorá-lo utilizando a própria linguagem Ruby.

Como criar métodos que acessam e modificam os valores de uma variável de instância é uma tarefa bastante comum, os criadores da linguagem pensaram em facilitar este processo e

criaram: `attr_writer` e `attr_reader` . Esses métodos podem receber vários argumentos do tipo `Symbol` . Cada símbolo representa o nome da variável que desejamos expor para leitura ou escrita:

```
# coding: utf-8
class Livro
  attr_writer :preco
  attr_reader :preco

  # outros métodos
end
```

Quando chamamos o método `attr_writer` passando o símbolo `:preco` , automaticamente os objetos do tipo `Livro` passam a ter um método `preco=` . O mesmo acontece quando chamamos o método `attr_reader` , neste caso os objetos passam a ter um método `preco` :

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
teste_e_design.preco = 79.9
puts teste_e_design.preco
```

Se você deseja criar um par de métodos, um de leitura e outro de escrita para uma determinada variável de instância, existe uma maneira mais simples do que utilizar os métodos `attr_reader` e `attr_writer` . Com esse intuito, foi criado o método `attr_accessor` :

```
# coding: utf-8
class Livro
  attr_accessor :preco

  # outros métodos
end
```

O `attr_accessor` atua como um substituto para os métodos `attr_writer` e `attr_reader` . Quando o usamos, definimos um

método de leitura e um outro de escrita seguindo a convenção da linguagem.

Com isso, os métodos `preco` e `preco=`, que haviam sido definidos manualmente, podem ser removidos.

Símbolos

Símbolos são palavras que parecem com variáveis, eles podem conter letras, números e `_` (underline). Símbolos são `String` mais leves e geralmente usados como identificadores.

A verdade é que símbolos são strings, como uma diferença importante, eles são imutáveis, ou seja, seu valor interno não pode ser alterado. Por isso, geralmente os utilizamos como identificadores.

```
:um_simbolo_qualquer.object_id == :um_simbolo_qualquer.object_id
# true
"um_simbolo_qualquer".object_id ==
                                "um_simbolo_qualquer".object_id
# false
```

Veja que independente da quantidade de vezes que referenciamos um símbolo, ele sempre será o mesmo objeto (o método `object_id` retorna um identificador do objeto na memória). Já no caso das `String`, mesmo sendo exatamente iguais, são objetos diferentes na memória. Como `String` são imutáveis, cada objeto tem seu próprio espaço em memória; quando elas não são mais referenciadas, poderão ser coletadas pelo coletor de lixo (*Garbage Collector* - GC).

Já os símbolos, por terem um única instância na memória, nunca serão coletados pelo GC. Além disso, símbolos não são

guardados somente na memória, eles também ficam em um dicionário de símbolos otimizado pelo interpretador.

3.6 ATRIBUTOS NEM TÃO PRIVADOS ASSIM

Um das características de variáveis de instância é que elas são privadas, ou seja, seu valor não pode ser alterado fora da classe, apenas com métodos públicos que internamente em sua implementação alteram o valor da variável. Vimos isso quando precisamos alterar o valor da variável `preco`.

Infelizmente, existem maneiras de burlar estas restrições. Uma vez que temos em mãos a instância de um objeto qualquer, podemos ler e **alterar** o valor das variáveis de instância.

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
puts teste_e_design.instance_variable_get "@preco" # => 60.9
```

O método `instance_variable_get` retorna o valor de uma variável de instância qualquer. Basta passar como argumento o nome da variável com o `@`. No código anterior, o resultado será `60.9`. Ler o valor da variável não é tão problemático se compararmos com o fato de que podemos alterar o valor de qualquer variável de instância também:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
teste_e_design.instance_variable_set "@preco", 75.5
puts teste_e_design.preco # => 75.5
```

O método `instance_variable_set` recebe dois argumentos. O primeiro é o nome da variável de instância que você deseja alterar, e o segundo o valor que você deseja atribuir à variável.

Os métodos `instance_variable_set` e `instance_variable_get` são herdados por todos os objetos Ruby. Eles pertencem a classe `Object`, que é a classe pai da hierarquia de classes que temos na linguagem. Esses métodos nos apresentam o começo de um assunto muito discutido e poderoso da linguagem, conhecido como metaprogramação.

3.7 GRANDES PODERES, GRANDES RESPONSABILIDADES

Os métodos que vimos na seção anterior podem parecer muito poderosos em uma primeira impressão. Porém, com o tempo e dependendo do tamanho do projeto, podem nos trazer grandes dores de cabeça. Vamos tomar como exemplo a nossa classe `Livro`:

```
# coding: utf-8
class Livro
  attr_accessor :preco

  def initialize(autor, isbn = "1", numero_de_paginas, preco)
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @preco = preco
  end

  def to_s
    "Autor: #{@autor}, ISBN: #{@isbn},
    Páginas: #{@numero_de_paginas}"
  end
end
```

Você, ao se sentir o Peter Parker da vida real, resolve alterar o valor de alguma variável usando o `instance_variable_set`:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
```

```
teste_e_design.instance_variable_set "@preco", 75.5
puts teste_e_design.preco # 75.5
```

Depois de alguns anos, você decide casar-se com Mary Jane, abandonar sua vida de homem aranha e também o desenvolvimento de software. O projeto é assumido por outro desenvolvedor que resolve alterar o nome da variável `preco` para `valor` :

```
# coding: utf-8
class Livro
  attr_accessor :valor

  def initialize(autor, isbn = "1", numero_de_paginas, valor)
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @valor = valor
  end

  def to_s
    "Autor: #{@autor}, ISBN: #{@isbn},
    Páginas: #{@numero_de_paginas}"
  end
end

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
puts teste_e_design.preco
# => NoMethodError: undefined method 'preco'
      for Autor: Mauricio Aniche, ISBN: 123454,
        Páginas: 247:Livro
```

A chamada ao método `preco` retornará o erro: `NoMethodError` . Obviamente, porque o método agora se chama `valor` .

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9
puts teste_e_design.valor
```

Porém, o antigo desenvolvedor homem aranha deixou em algum lugar do sistema uma chamada ao

```
instance_variable_set :
```

```
teste_e_design.instance_variable_set "@preco", 75.5  
puts teste_e_design.instance_variable_get "@preco"
```

O problema é que, além de tentarmos alterar o valor da uma variável que não existe mais, o código anterior é silencioso e não retorna nenhum erro avisando que aquela variável não existe. Pelo contrário, ele cria uma variável nova chamada `@preco` que não tem relação alguma com nosso domínio, já que quando a variável não existe, ela será criada ao tentarmos acessá-la.

O código tornou-se obsoleto. Se não existirem testes de unidade que validem estes comportamentos, provavelmente ele seria observado somente em produção quando algum cliente usasse o sistema.

A dica é simples aqui é, se a variável de instância é privada, mesmo que a linguagem lhe permita burlar essa característica, não faça, resolva o problema de outra maneira. Certamente existirá algum método na interface pública do seu objeto que faça por você a alteração da variável de uma forma segura e correta.

ESTRUTURAS DE DADOS

Ruby possui três tipos de estruturas de dados que veremos com detalhes neste capítulo. Cada uma destas estruturas possui suas qualidades e defeitos, que vamos explorar daqui em diante.

4.1 TRABALHE COM ARRAYS

Arrays em Ruby são coleções indexadas, ou seja, guardam objetos em uma determinada ordem e disponibilizam métodos que permitem acessar objetos destas coleções através do seu índice. Diferente do que acontece com a linguagem C ou Java, em que precisamos definir arrays com uma quantidade máxima de objetos, em Ruby os arrays não precisam ter seu tamanho pré-definido, eles crescem conforme a necessidade.

Existem várias formas de definir um `Array` em Ruby, sendo que a mais simples é utilizando `[]` :

```
numeros = [1, 2, 3]
puts numeros.class # => Array
```

Criamos um objeto do tipo `Array` utilizando dois colchetes e separando os vários elementos do `Array` com a vírgula. Para acessar os elementos de um `Array`, usamos o método `[índice]` que recebe como argumento a posição do elemento que desejamos

acessar. Lembre-se de que, em Ruby, os índices começam em 0.

```
numeros = [1, 2, 3]
puts numeros[0] # => 1
puts numeros[1] # => 2
puts numeros[2] # => 3
```

Quando precisamos acessar o primeiro ou o último elemento, podemos fazê-lo através de métodos (`first` e `last`) da classe `Array` :

```
numeros = [1, 2, 3]
puts numeros.first # => 1
puts numeros.last # => 3
```

Como Ruby é uma linguagem com tipagem dinâmica, podemos adicionar objetos de qualquer tipo dentro de um mesmo `Array` . Adicionar novos elementos em um array se dá através do método `<<` , bem parecido com um *append*, que adicionará o novo elemento no final do `Array` .

```
numeros = [1, 2, 3]
numeros << "ola"
puts numeros # [1, 2, 3, "ola"]
# => 1, 2, 3, "ola"
```

O problema de adicionar qualquer tipo de objeto dentro de um `Array` é que muitas vezes não sabemos qual objeto estamos lidando. Por exemplo, vamos definir um método que recebe um `Array` como argumento, busca pelo primeiro elemento, o multiplica por 2 e, por fim, imprime o resultado na tela:

```
def multiplica_primeiro_elemento_por_dois(numeros)
  puts 2 * numeros.first
end

multiplica_primeiro_elemento_por_dois [1, 2, 3] # 2
multiplica_primeiro_elemento_por_dois ["abc", 2, 3]
# => TypeError: String can't be coerced into Fixnum
```

Veremos que existem vantagens na tipagem dinâmica das estruturas de dados em Ruby nos próximos capítulos.

A criação de array de strings

Para criar um Array de strings, a sintaxe utilizada pode ser a mesma que usamos para criar um array com outros tipos de elementos:

```
palavras = ['ola', 'mundo']  
p palavras # => ["ola", "mundo"]
```

Existe uma sintaxe mais simples para criar um array de palavras:

```
palavras = %w{ola mundo}  
p palavras # => ["ola", "mundo"]
```

A vantagem de usar o `%w{}` é poder separar as palavras que compõem o Array usando espaço, e não vírgula, assim poluindo pouco o código. Podemos utilizar também o `%W{}`, que permite a interpolação de valores nas palavras que compõem o array:

```
nome = "Lucas"  
palavras = %W{ola #{nome}}  
p palavras # ["ola", "Lucas"]
```

4.2 GUARDANDO NOSSOS LIVROS

Aproveitando que agora conhecemos um pouco de uma estrutura de dados Ruby, vamos guardar as instâncias dos objetos `Livro` que criamos dentro dela.

```
biblioteca = []  
  
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 70.5  
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
```

```

biblioteca << teste_e_design
biblioteca << web_design_responsivo

puts biblioteca
# => Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247
# => Autor: Tarcio Zemel, Isbn : 452565, Páginas: 321

```

Hoje, decidimos que vamos guardar os objetos dentro de um Array , mas futuramente podemos descobrir outra estrutura que funcione melhor, seja mais rápida ou forneça vantagens que o Array não oferece. Se optarmos por essa mudança, teremos de procurar todos os lugares que instanciam objetos do tipo Livro , guardando-os dentro de um array, e adaptar o código para a maneira da nova estrutura.

Para nos protegermos desse problema, podemos isolar o código que faz essa atribuição em um único ponto, que será invocado em outras partes do código. Assim, alteramos em um só lugar e o sistema inteiro está modificado. Essa conceito chama-se *encapsulamento*.

Vamos isolar este comportamento de guardar livros dentro de um array, em uma classe que podemos chamar de Biblioteca . E quando houver a necessidade de mudarmos de Array para alguma outra estrutura, faremos em apenas um lugar. Adicione a definição da classe Biblioteca dentro de um arquivo chamado biblioteca.rb , dentro do diretório lib :

```

class Biblioteca
  def initialize
    @livros = []
  end

  def adiciona(livro)
    @livros << livro
  end
end

```

```
end  
end
```

Agora, para guardar os livros que criamos, não vamos criar um Array e apendar os valores diretamente nele. Criaremos um objeto do tipo `Biblioteca` e chamaremos o método responsável por adicionar novos livros na biblioteca. Porém, para testar a classe `Biblioteca`, é necessário carregar o arquivo que contém sua definição:

```
require File.expand_path('lib/livro')  
require File.expand_path('lib/biblioteca')
```

Ao passar do tempo, os arquivos necessários para efetuar os testes serão um maior número. Para não dificultar os testes, vamos criar um único arquivo que fará o `require` de todos os outros. Vamos chamá-lo de `loja_virtual.rb` e adicioná-lo no diretório `lib` da pasta `loja_virtual`. Dentro dele, faremos o carregamento dos arquivos `biblioteca.rb` e `livro.rb`:

```
require File.expand_path('lib/livro')  
require File.expand_path('lib/biblioteca')
```

Agora para efetuar os testes, basta executar um único `require`, do arquivo `loja_virtual.rb`:

```
require File.expand_path('lib/loja_virtual')
```

E executar os testes da classe `Biblioteca`:

```
biblioteca = Biblioteca.new  
  
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 70.5  
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",  
                                189, 67.9  
  
biblioteca.adiciona teste_e_design  
biblioteca.adiciona web_design_responsivo
```

Se precisarmos acessar quais são os livros que existem dentro da nossa biblioteca, basta expor a variável `@livros` através do `attr_reader` :

```
class Biblioteca
  attr_reader :livros

  def initialize
    @livros = []
  end

  def adiciona(livro)
    @livros << livro
  end
end
```

E para acessar as variáveis, podemos chamar o método `livros` na `Biblioteca` :

```
biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 70.5
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
                                189, 67.9

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

p biblioteca.livros
# => [Autor: Mauricio Aniche, Isbn: 1, Páginas: 247,
      Autor: Tárcio Zemel, Isbn: 1, Páginas: 189]
```

4.3 PERCORRENDO MEU ARRAY

No capítulo *Seu primeiro passo no Ruby: convenções e as diferentes estruturas primitivas*, aprendemos algumas estruturas de controle, entre elas o `for` , que serve para iterarmos sobre ranges, e também coleções como o `Array` . O método `livros` do objeto `biblioteca` retorna todos os livros em formato de `Array` . Para

acessar cada livro deste array, basta iterarmos através do `for` :

```
biblioteca = Biblioteca.new

# popula a biblioteca

for livro in biblioteca.livros do
  p livro.valor
end

# => 70.5
# => 67.9
```

4.4 COMO SEPARAR OS LIVROS POR CATEGORIA: TRABALHE COM HASH

A medida que nossa biblioteca cresce, precisamos separar os livros dentro da biblioteca por categorias. Dessa forma, o primeiro passo para fazermos essa separação é adicionarmos um atributo chamado `categoria` à classe `Livro` :

```
# coding: utf-8
class Livro
  attr_accessor :valor
  attr_reader :categoria

  def initialize(autor, isbn = "1", numero_de_paginas, valor,
                categoria)

    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @valor = valor
    @categoria = categoria
  end

  def to_s
    "Autor: #{@autor}, ISBN: #{@isbn},
    Páginas: #{@numero_de_paginas},
    Categoria: #{@categoria}"
  end
end
```

end

A categoria do livro será um `Symbol` , como: `:ruby` , `:testes` , `:html` , `:javascript` , `:ios` etc. Os livros que criamos anteriormente recebem um argumento a mais, contendo a sua respectiva categoria:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9,
                           :testes
puts teste_e_design.categoria # => :testes
```

Agora que cada livro possui sua categoria, vamos adaptar nossa classe `Biblioteca` para guardá-los. Dessa forma, a classe `Biblioteca` terá a organização de categorias.

Os arrays guardam os elementos sequencialmente e não existe nenhuma divisão destes elementos. Precisamos de uma estrutura que suporte este tipo divisão, um número N de objetos em um determinado container, sendo possível termos vários containers também.

Vamos utilizar um `Hash` , estrutura que guarda sempre uma chave (identificador único, que geralmente é um símbolo) e um valor (que pode ser qualquer tipo de objeto Ruby). Sua inicialização é feita utilizando `{}` :

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247, 60.9,
                           :testes

web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
                                  189, 70.9, :web_design

hash = {"123454" => teste_e_design,
        "452565" => web_design_responsivo}
```

O hash anterior possui dois containers, um cujo identificador é a `String` `"123454"` , que corresponde ao `isbn` do objeto

teste_e_design , e outro cujo identificador é a String "452565" , que corresponde ao isbn do objeto web_design_responsivo .

Cada um destes containers possui um determinado valor. No Hash anterior, o container "123454" possui como valor o objeto teste_e_design , já o container, cujo identificado é "452565", possui como valor o objeto web_design_responsivo .

Quando desejamos acessar o valor que está incluso dentro um Hash , basta invocarmos o método [:chave] passando a chave identificadora do container:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes

web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                                  70.9, :web_design

hash = { "123454" => teste_e_design,
        "452565" => web_design_responsivo }

p hash["123454"]
# => Autor: Tárcio Zemel, Isbn: 452565, Páginas: 189,
    Categoria: web_design
```

Bem parecido com a maneira que acessamos valores dentro de um Array . A diferença é que passamos um objeto que representa a chave identificadora, e não um Integer que representa o índice, como acontece quando acessamos valores de um Array .

Agora vou provar que o encapsulamento que fizemos ao definir a classe Biblioteca nos ajudará a migrar sem sofrimentos de Array para Hash a estrutura que guardava os livros:

```
class Biblioteca
```



```

attr_reader :livros

def initialize
  @livros = {} # Inicializa com um hash
end

def adiciona(livro)
  @livros[livro.categoria] ||= []
  @livros[livro.categoria] << livro
end
end

```

O código que instancia os livros e os guarda dentro da biblioteca continua funcionando.

```

biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", 247, "123454",
                           :testes

web_design_responsivo = Livro.new "Tárcio Zemel", 189, "452565",
                                  :web_design

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

```

O problema agora é o método que usamos para retornar os livros que estão dentro da nossa biblioteca retorna um Hash , e não mais um Array , e a forma de iterar sobre estas duas estruturas de dados é diferente. A primeira opção é fazer com que o método livros continue retornando um Array :

```

class Biblioteca
  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  def livros
    @livros.values.flatten
  end
end

```

Criando nossa própria implementação do método `livros`, conseguimos manter o mesmo comportamento anterior, quando retornávamos um array diretamente. A segunda opção para resolver o problema é retornar um `Hash` e alterar a forma que iteramos sobre o retorno no método `livros`:

```
class Biblioteca
  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  # Ou criando um attr_reader :livros
  def livros
    @livros
  end
end

biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes

web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                                  70.9, :web_design

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

for categoria, livros in biblioteca.livros do
  p categoria

  for livro in livros do
    p livro.valor
  end
end

# => :testes
# => 60.9
# => :web_design
# => 70.9
```

A diferença é que o método `for` recebe dois argumentos, a chave e o valor do hash. Em nosso exemplo, a chave é a categoria da estante de livros (`:testes` , por exemplo), e valor é um array com os livros desta categoria.

Os métodos `values` e `flatten`

Quando precisamos obter todos os valores de um determinado hash, independente do container, utilizamos o método `values` (que retorna todos os valores do hash dentro de um `Array`):

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes

web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                                70.9, :web_design

hash = { testes: teste_e_design,
        web_design: web_design_responsivo }
p hash.values # [teste_e_design, web_design_responsivo]
```

No caso da classe `Biblioteca` , nós guardamos conjuntos de livros dentro de `Array` , sendo um `Array` para cada categoria. Quando os valores de um hash são do tipo `Array` , o método `values` retornará um novo `Array` com vários `Arrays` dentro, o que pode não ser um resultado muito interessante para trabalharmos. Veja:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes
web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                                70.9, :web_design

hash = { testes: [teste_e_design],
        web_design: [web_design_responsivo]}
p hash.values
# => [Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247,
      Categoria: testes],
```

```
[Autor: Tárzio Zemel, Isbn: 452565, Páginas: 189,  
  Categoria: web_design]
```

Precisamos que o retorno do método `livros` seja um único `Array` com todos os livros existentes dentro da `Biblioteca`, independente da categoria deles. Podemos alcançar nosso objetivo utilizando o método `flatten`:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,  
                           60.9, :testes  
web_design_responsivo = Livro.new "Tárzio Zemel", "452565", 189,  
                                  70.9, :web_design
```

```
hash = { testes: [teste_e_design],  
         web_design: [web_design_responsivo]}
```

```
p hash.values.flatten  
# => [Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247,  
      Categoria: testes,  
      Autor: Tárzio Zemel, Isbn: 452565, Páginas: 189,  
      Categoria: web_design]
```

O método `flatten` procura por objetos do tipo `Array`, dentro do `Array` em que o método foi chamado (em nosso exemplo, o `Array` é retornado pelo método `values`), extrai esses valores e retorna um novo `Array` com todos os elementos extraídos.

É importante ressaltar que o método `flatten` é recursivo. Se o invocarmos em um `Array`, que possui vários outros objetos do tipo `Array`, que por sua vez sejam formados por alguns outros objeto que também sejam `Array`, ele recursivamente extrairá os elementos de todos eles e retornará os elementos em um novo objeto `Array`.

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,  
                           60.9, :testes
```

```

web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                                70.9, :web_design
jsf_e_jpa = Livro.new "Gilliard Cordeiro", "543465", 234, 64.9,
                    :frameworks_mvc

hash = {}
hash[:testes] = [ [ teste_e_design ], [ jsf_e_jpa ] ]
hash[:web_design] = [ [ web_design_responsivo ] ]

p hash.values.flatten
# => [Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247,
      Categoria: testes,
      Autor: Gilliard Cordeiro, Isbn: 543465, Páginas: 234,
      Categoria: frameworks_mvc,
      Autor: Tárcio Zemel, Isbn: 452565, Páginas: 189,
      Categoria: web_design]

```

4.5 INDO MAIS A FUNDO: HASHES NO RUBY 1.9

A chave identificadora de um objeto Hash pode ser um objeto de qualquer tipo (String , Integer , Livro etc.). Porém, o mais comum é definirmos as chaves como Symbol , que são comumente usadas para esta finalidade. Por exemplo:

```

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                        60.9, :testes

web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                        70.9, :web_design

hash = { :testes => [teste_e_design],
         :web_design => [web_design_responsivo] }

p hash[:testes] # => ["Autor: Mauricio Aniche, Isbn: 123454,
                    Páginas: 247,
                    Categoria: :testes"]

```

Na versão 1.9 da linguagem Ruby, foi introduzida uma nova sintaxe para declarar elementos em um hash quando a chave

identificadora é um `Symbol` - muito parecida com a sintaxe da linguagem JavaScript. Entretanto, a forma que acessamos o valor de `Hash` continua sendo a mesma:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                          60.9, :testes
web_design_responsivo = Livro.new "Tárcio Zemel", "452565", 189,
                          70.9, :web_design

hash = { testes: [teste_e_design],
        web_design: [web_design_responsivo] }
p hash[:testes] # => ["Autor: Mauricio Aniche, Isbn: 123454,
                    Páginas: 247,
                    Categoria: :testes"]
```

4.6 INDO MAIS A FUNDO: O OPERADOR `||=`

Imagine que queremos setar o valor em uma determinada variável somente se o valor atual dela for `nil`. Caso ela já esteja preenchida, o valor deve ser mantido.

```
idade = nil

idade = 27 unless idade
puts idade # 27

idade = 35 unless idade
puts idade # 27
```

Muito código para uma tarefa trivial. Por isso, os criadores do Ruby fizeram o operador `||=`, que executa o mesmo comportamento que implementamos com o `unless`:

```
idade = nil

idade ||= 27
puts idade # 27

idade ||= 35
```

```
puts idade # 27
```

4.7 INDO MAIS A FUNDO: O TIPO SET

Quando trabalhamos com objeto do tipo `Array`, podemos repetir os mesmos elementos quantas vezes acharmos necessário, por exemplo:

```
numeros = [1, 2, 2, 3, 2, 1]  
p numeros # [1, 2, 2, 3, 2, 1]
```

Não existe nenhuma regra que proíba a duplicidade de elementos. Mas em certas ocasiões, é necessário garantirmos que, dentro de uma estrutura de dados, existam elementos únicos, ou seja, sem repetição. Ruby contempla essa necessidade através de um outro tipo de coleção, chamada `Set`, que guarda valores em um ordem não definida (diferente de arrays) e garante a não duplicidade.

A sintaxe usada para criarmos uma coleção do tipo `Set` é um pouco diferente das que vimos até o momento:

```
require 'set'  
  
numero_sem_repeticao = Set.new [1, 2, 2, 3, 2, 1]
```

Quando precisamos utilizar o tipo `Set`, precisamos carregar o arquivo `set.rb` que foi instalado junto com o interpretador e as outras classes básicas da própria linguagem. O arquivo `set.rb` contém a classe `Set`.

A classe `Set` possui um método `initialize` que recebe um `Array` com os elementos que formarão a coleção. Ao receber este array, são guardados apenas os elementos não repetidos, e descartados os restantes. Podemos verificar este comportamento

iterando no `Set` que foi criado:

```
require 'set'

numero_sem_repeticao = Set.new [1, 2, 2, 3, 2, 1]
for numero in numero_sem_repeticao do
  p numero
end

# => 1
# => 2
# => 3
```

Vimos que a classe `Set` consegue "descobrir" quais são os elementos "iguais" e manter apenas um destes elementos. Isso funciona muito bem com os números, mas e se tentarmos adicionar várias instâncias de um mesmo `Livro` dentro de um `Set`. Vamos ver?

```
require 'set'

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes
teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
                             60.9, :testes

livros = Set.new [teste_e_design, teste_e_design_2]
for livro in livros do
  p livro
end
```

Repare que duas instâncias de livro compõem o `Set`. Isso ocorre porque, para a linguagem Ruby, estes dois objetos são diferentes. Podemos verificar isso invocando e comparando o retorno do método `object_id` (que retorna um identificador único de cada objeto na memória) de cada um dos objetos:

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                           60.9, :testes
```



```
teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
                             60.9, :testes
```

```
p teste_e_design.object_id == teste_e_design_2.object_id
# => false
```

Para garantirmos que dois objetos são iguais por seus valores, devemos compará-los utilizando o método `eq1?` :

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                             60.9, :testes
```

```
teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
                             60.9, :testes
```

```
p teste_e_design.eq1? teste_e_design_2
# => false
```

O retorno continua sendo `false` , e em nenhum momento definimos o método `eq1?` . Na verdade, ele foi um método herdado da classe `Object` , e que em sua implementação original compara os objetos através de seus `object_id` . No momento em que decidirmos que o critério que define se um objeto é igual ao outro levará outro valor em consideração, devemos sobrescrever o método `eq1?` e fazermos a comparação que julgamos ser a adequada:

```
# coding: utf-8
class Livro
  attr_reader :isbn

  def eq1?(outro_livro)
    @isbn == outro_livro.isbn
  end
end
```

```
teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                             60.9, :testes
```

```
teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
```

```
p teste_e_design.eql? teste_e_design_2 # => true
```

Sobrescrevemos o método `eql?` e definimos que, se um outro livro tiver o mesmo `isbn` que o livro que estamos comparando, eles são iguais. É importante salientar que não estamos dizendo que eles são os mesmos objetos na memória.

==, EQUAL? OU EQL?

O método `==` retorna `true` apenas se os dois objetos envolvidos na comparação forem a mesma instância, este é seu comportamento padrão. Ele pode ser sobrescrito a fim de efetuar a comparação de outras maneiras.

O método `equal?` é similar ao `==`. Ele retorna `true` apenas se os dois objetos envolvidos na comparação forem a mesma instância. As bibliotecas existentes na linguagem, quando têm a necessidade de comparar se dois objetos são a mesma instância, usam o método `equal?`. Por esse motivo, não devemos sobrescrever este método, pois o efeito pode ser bastante prejudicial.

Por fim, o método `eql?` compara as instâncias dos objetos também, por padrão. Porém, este método **deve** ser sobrescrito quando desejamos avaliar se dois objetos são iguais por seus valores, como fizemos com a classe `Livro`.

Porém, sobrescrever apenas o método `eql?` não é suficiente para evitarmos duplicidade de objetos do tipo `Livro` dentro de

uma coleção do tipo `Set` . Ainda não é suficiente, porque o `Set` internamente utiliza um `Hash` para guardar os valores, e uma coleção do tipo `Hash` é um conjunto de buckets (container) com um rótulo e uma determinada quantidade de objetos dentro.

O rótulo destes buckets são na verdade o retorno do método `hash` de um determinado objeto. No caso dos objetos do tipo `Livro` , o retorno do método `hash` quase sempre será diferente. Isso porque todo objeto possui uma implementação *default* do método `hash` que faz um cálculo com propriedades específicas do objeto em questão.

Se quisermos evitar duplicidade dos objetos do tipo `Livro` dentro de um `Set` , devemos fazer com que instâncias que possuem o mesmo ISBN (que é nosso critério de igualdade) tenham o mesmo retorno quando invocarmos o método `hash` :

```
# coding: utf-8
class Livro
  def hash
    @isbn.hash
  end
end

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                          60.9, :testes

teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
                          60.9, :testes

p teste_e_design.hash == teste_e_design_2.hash # => true
```

O que fizemos foi retornar o valor do método `hash` do atributo `String isbn` . É sempre importante tomarmos cuidados quando redefinimos o método `hash` , principalmente quando guardarmos vários objetos deste tipo dentro de `Hash` e `Set` . Se o

valor do `hash` for pouco variável, podemos ter várias colisões de `hash`, o que pode ocasionar uma busca muito mais lenta por esses elementos.

Agora podemos testar que, caso instâncias diferentes da classe `Livro` tenham o mesmo `isbn`, eles não serão mais duplicadas dentro de um `Set` :

```
require 'set'

teste_e_design = Livro.new "Mauricio Aniche", "123454", 247,
                          60.9, :testes

teste_e_design_2 = Livro.new "Mauricio Aniche", "123454", 247,
                          60.9, :testes

livros = Set.new [teste_e_design, teste_e_design_2]
for livro in livros do
  p livro
end

# => Autor: Mauricio Aniche, Isbn: 123454,
      Páginas: 247, Categoria: :testes
```

4.8 PRÓXIMOS PASSOS

No próximo capítulo, veremos maneiras mais elegantes de iterarmos coleções, usando características funcionais da linguagem Ruby. Exploraremos diversos métodos úteis que existem nas APIs do Ruby.

RUBY E A PROGRAMAÇÃO FUNCIONAL

Ruby é conhecida por ser uma linguagem relacionada ao paradigma orientado a objetos, porém, também possui suporte ao paradigma funcional. Neste capítulo, mostrarei os conceitos gerais relacionados à programação funcional, e explicarei como a linguagem Ruby suporta estes conceitos e nos ajuda a criar códigos mais legíveis, com manutenção mais fácil e principalmente mais eficiente.

5.1 O QUE É PROGRAMAÇÃO FUNCIONAL

Fundamentalmente, a programação funcional é um paradigma de programação que trata a computação como a avaliação das funções matemáticas e a capacidade de evitar a mutabilidade de estado. O paradigma funcional enfatiza o uso das funções que não alteram estado, ao contrário da programação imperativa. O paradigma foi fundamentado no ano de 1930 com o lambda calculus, um sistema formal desenvolvido para investigar a definição de funções, aplicação delas e também a recursão.

5.2 FUNÇÕES PURAS

A principal diferença entre as funções matemáticas e as funções criadas nas linguagens imperativas é que, na segunda, funções podem causar efeitos colaterais, alterando o valor já calculado anteriormente. Na prática, isso quer dizer que, ao criarmos uma função e a invocarmos, o seu resultado vai depender do estado atual no momento em que ela for executada.

No paradigma funcional, as funções dependem apenas dos argumentos que foram recebidos em sua chamada. Sendo assim, invocar a função N vezes resulta sempre no mesmo valor, por este motivo chamamos estas funções de **puras**.

Eliminar estes efeitos colaterais facilita o entendimento do comportamento do programa e também evita preocupações quando temos código sendo executado paralelamente, pois nosso código naturalmente se tornou **Thread Safe**.

A linguagem Ruby contempla várias características das funções puras. Vamos tomar como exemplo a `String`, que possui várias funções (métodos) puras:

```
nome = "Lucas"  
puts nome.upcase # => LUCAS  
puts nome # => Lucas
```

Nesse código, criamos uma variável `nome` que possui o valor `"Lucas"`. A chamada do método `upcase`, em vez de alterar a variável para guardar o valor `"LUCAS"`, retorna uma **nova** `String` com o valor em caixa alta. Você pode confirmar este comportamento na linha criada após a chamada do método `upcase`, que imprime o valor da variável `nome`, sendo este exatamente o mesmo valor que definimos na declaração da variável.

O método `upcase` é uma função pura, porque não importa quantas vezes seja invocado, retornará sempre o mesmo valor e também não causa efeitos colaterais.

O símbolo !

Ao mesmo tempo em que Ruby contempla o paradigma funcional, ela também é uma linguagem imperativa, ou seja, possui funções que focam em alterar o estado dos dados. Existem vários métodos na API da linguagem que possuem o caractere `!` após seus nomes. O caractere `!` no final do nome do método é uma convenção, que significa que o método deve ser usado com moderação, porque pode causar efeitos colaterais. Por exemplo, veja o método `upcase!` :

```
nome = "Lucas"
puts nome.upcase! # => LUCAS
puts nome # => LUCAS
```

Como você pode notar, ao executar este código, o método `upcase!` não é um método funcional puro, porque possui efeitos colaterais que alteram a `String` dentro da variável `nome`. Por isso, sempre que usarmos algum método com `!` no final, precisamos tomar cuidado.

O lado bom é que, se precisarmos definir um método que vai alterar o estado de um determinado objeto, podemos defini-los com o caractere `!` no final. Assim, os outros desenvolvedores ficarão cientes do cuidado ao utilizar aquele método.

5.3 COMANDOS QUE RETORNAM VALORES

Literais, chamada de métodos, variáveis, estruturas de controle,

todos estes comandos são avaliados como expressões pelo interpretador Ruby. Vamos tomar como exemplo o caso no qual desejamos atribuir um determinado valor a uma variável se uma condição for verdadeira, e outro valor caso a condição seja falsa. Podemos criar esse código em Ruby da seguinte maneira:

```
valor = nil
numero = "dois"

if numero == "um" then valor = 1
elsif numero == "dois" then valor = 2
else valor = 3
end

p valor # => 2
```

Porém, o código anterior pode ficar mais legível se aproveitarmos o poder da linguagem Ruby de avaliar tudo como uma expressão:

```
numero = "dois"
valor = if numero == "um" then 1
        elsif numero == "dois" then 2
        else 3
      end

p valor # => 2
```

Podemos nos aproveitar desta habilidade da linguagem de avaliar tudo como uma expressão no momento que precisamos declarar várias variáveis com o mesmo valor:

```
a = b = c = 0
p a, b, c # => 0 0 0
```

Até mesmo quando utilizamos um `for`, o seu resultado é uma expressão. Se percorrermos um `Array` com três elementos usando um `for` e multiplicarmos cada item por 2, por exemplo,

podemos atribuir seu resultado a uma variável:

```
numeros = [1, 2, 3, 4]
novos_numeros = for n in numeros
  n * 2
end

p novos_numeros # => [1, 2, 3, 4]
```

O método `for` que utilizamos retorna um `Array` com os mesmos valores inseridos no `Array` antigo. Retornar sempre o `Array` original é um comportamento do `for` e também do método `each`, que veremos em breve. Para criar um novo `Array` com os valores retornados por cada iteração, é necessário utilizar o método `map` que também será visto em breve.

Métodos sem return

Todos os métodos Ruby retornam sempre o resultado da última expressão declarada, logo, você não precisa explicitamente adicionar o `return` no final de cada método:

```
def boas_vindas(nome)
  "Bem vindo: #{nome}"
end

p boas_vindas("Lucas") # => "Bem vindo: Lucas"
```

Quando invocado, o método `boas_vindas` retornará a `String` interpolada com o valor do argumento `nome` que foi passado.

5.4 FUNÇÕES DE ALTA ORDEM: HIGHER-ORDER FUNCTIONS

Funções ou métodos *são high-order* quando têm a capacidade

de receber outras funções como argumentos, ou retornar funções como resultado. Em Ruby, isto é feito usando blocos, lambdas e procs.

Blocos, lambdas e procs são um dos aspectos mais poderoso da linguagem Ruby, e também um dos que causam mais confusões para serem entendidos. Isso porque Ruby possui quatro maneiras de lidar com high-order functions.

Blocos

Este é o método mais comum trabalhar com funções high-order em Ruby. Os blocos são muito utilizados e comuns quando percorremos coleções:

```
numeros = [1, 2, 3, 4]

numeros.each { |numero| p numero }
# => 1
# => 2
# => 3
# => 4
```

Mas o que está acontecendo nesse código afinal?

A primeira e **mais importante** parte que devemos entender está na chamada ao método `each` que fizemos na variável `numeros`. Ao invocarmos o método, estamos passando uma função ou bloco de código como argumento. Internamente, o método `each` itera o `Array`, executa o bloco de código recebido como argumento passando o valor da iteração (neste caso, a variável `numero`). O bloco de código que recebe a variável `numero` está imprimindo-a no console utilizando o método `p`.

Existem outros métodos da classe `Array` que recebem um

bloco de código como argumento e o executam a cada iteração. Por exemplo, o método `collect` :

```
numeros = [1, 2, 3, 4]
numeros_ao_quadrado = numeros.collect { |numero| numero ** 2 }
p numeros_ao_quadrado # => [1, 4, 9, 16]
```

O comportamento do método `collect` é similar ao método `each` , e cada item do `Array` onde o método foi invocado é passado como argumento para o bloco recebido na chamada. Ao invocar o método `collect` , obtemos um novo `Array` com todos os números ao quadrado. Desta maneira, quando imprimimos a variável `numeros_ao_quadrado` , o resultado é `[1, 4, 9, 16]` .

Tanto o método `each` quanto o método `collect` fazem parte da API pública da classe `Array` . Existem vários outros métodos bastante úteis na API dos arrays, você deve sempre consultá-la para não reimplementar comportamentos que já existem. Lembre-se de que um dos principais atrativos da linguagem Ruby é a sua legibilidade, conhecer bem a API da linguagem é um dos passos mais importantes para conseguir tal vantagem.

5.5 CRIE SEU PRÓPRIO CÓDIGO QUE USA UM BLOCO DE CÓDIGO

Mas e se quisermos criar nosso próprio método que recebe um bloco de código? Como podemos implementá-lo?

Vamos criar um método que filtra os livros por uma determinada categoria, itera cada um destes livros e executa um bloco de código que será passado para este método. Nosso primeiro passo será criar o método dentro da classe `Biblioteca` :

```

class Biblioteca
  def initialize
    @livros = {} # Inicializa com um hash
  end

  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  def livros
    @livros.values.flatten
  end

  def livros_por_categoria(categoria)
    @livros[categoria]
  end
end

```

O filtro por categoria é bem simples, o argumento esperado será o `Symbol` identificador do tipo dos livros que desejamos. Agora precisamos iterar estes livros e executar um bloco de código passado como argumento na chamada do método `livros_por_categoria`:

```

class Biblioteca
  def initialize
    @livros = {} # Inicializa com um hash
  end

  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  def livros
    @livros.values.flatten
  end

  def livros_por_categoria(categoria)
    @livros[categoria].each do |livro|
      yield livro
    end
  end
end

```

```

        end
    end
end

```

Quando filtramos os livros por categoria (`@livros[categoria]`), recebemos como retorno um `Array`, e agora podemos percorrê-lo utilizando o método `each` que recebe um bloco de código, e o executa `N` (número de livros que contém dentro do `Array`) vezes. A cada iteração, o bloco é executado recebendo o item (no caso um objeto do tipo `Livro`) na variável `livro` que declaramos entre os caracteres `||`.

Outra novidade é que, diferentemente dos atributos, blocos não precisam ser declarados na assinatura de método. Para executá-los, basta chamar o método `yield` passando os argumentos que serão recebidos pelo bloco declarado na chamada do método `livros_por_categoria`. O método `yield` executará automaticamente o bloco que for passado na chamada do método.

Agora basta invocar o método passando a `categoria` pela qual desejamos efetuar o filtro e também o bloco de código que será executado para cada um dos objetos do tipo `Livro` encontrados no `Array` (adicione também o `attr_reader` para o atributo `autor` na classe `Livro`):

```

biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454",
                           247, 69.9, :testes
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
                                  189, 69.9, :web_design

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

biblioteca.livros_por_categoria :testes do |livro|
  p livro.autor
end

```

```
end
```

```
=> "Mauricio Aniche"
```

Quando executamos o método `livros_por_categoria`, passamos um `Symbol` `:testes` que representa a categoria de livros que desejamos filtrar e também um bloco de código que recebe na variável `livro` declarada dentro dos caracteres `||` cada um dos objetos existentes para a categoria filtrada.

A grande vantagem desta abordagem é que possuímos uma maneira flexível de interagir com o método, ou seja, o bloco passado na chamada do método pode decidir qual comportamento executar com cada um dos objetos recebidos.

Você sempre deve pensar em blocos como uma maneira de flexibilizar os métodos da sua API. Nós podemos, por exemplo, em vez de imprimir o nome do autor no console, imprimi-lo em uma impressora:

```
biblioteca = Biblioteca.new
impressora = Impressora.new

teste_e_design = Livro.new "Mauricio Aniche", "123454",
                           247, 69.9, :testes
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
                                  189, 69.9, :web_design

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

biblioteca.livros_por_categoria :testes do |livro|
  impressora.imprime livro.autor
end
```

A classe `Impressora` poderia ser uma implementação que envia dados para uma impressora conectada via USB ou mesmo via rede sem fio. O que quero lhe mostrar é que, trocando apenas

uma linha, mudamos o comportamento e a maneira que interagimos com o método `livros_por_categoria`. Ele é um método flexível, que recebe um bloco de código onde podemos customizar o comportamento que desejamos.

Evite erros quando um bloco não é passado

O método `livros_por_categoria` espera que um bloco de código seja passado como argumento. Porém, se este bloco for utilizado internamente, mas não for passado como argumento, ocorrerá um erro informando que este método deveria ter sido informado:

```
biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454",
                           247, 69.9, :testes

biblioteca.adiciona teste_e_design
biblioteca.livros_por_categoria :testes
# => LocalJumpError: no block given (yield)
```

Podemos resolver este problema de forma defensiva, evitando um erro caso nenhum bloco de código seja informado e utilizando o método `block_given?` que está disponível em todos os objetos criados em Ruby:

```
class Biblioteca
  def livros_por_categoria(categoria)
    @livros[categoria].each do |livro|
      yield livro if block_given?
    end
  end
end

biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454",
```

```
biblioteca.adiciona teste_e_design  
biblioteca.livros_por_categoria :testes
```

O método `block_given?` verifica se algum bloco foi passado como argumento na chamada do método e retorna um valor booleano. Em nosso exemplo, caso algum bloco seja passado, o executamos; caso contrário, o método `yield` não será invocado.

Esta técnica defensiva é muito útil quando criamos frameworks que serão utilizados por várias bases de código Ruby e onde não temos muito controle sobre os clientes destas APIs que estamos criando.

5.6 EXPLORANDO A API ENUMERABLE

As classes `Array` e `Hash` que vimos até o momento possuem métodos comuns entre elas, que são disponibilizados por um módulo (veremos módulos em breve) chamado `Enumerable`. Estes métodos permitem executar tarefas com coleções, com apenas uma ou duas linhas de código Ruby.

Método inject

Vamos criar uma classe chamada `Relatorio` dentro de um novo arquivo chamado `relatorio.rb`, que ficará dentro da pasta `lib` do projeto. Esta classe terá uma série de métodos para fornecer dados importantes referentes aos livros que temos cadastrados. Para começar, vamos criar um método que retorna a soma total dos preços de todos os livros que temos cadastrados na biblioteca:


```

class Relatorio
  def initialize(biblioteca)
    @biblioteca = biblioteca
  end

  def total
    soma = 0.0

    @biblioteca.livros.each do |livro|
      soma += livro.valor
    end

    soma
  end
end

```

Antes de criar os testes para esta nova classe, não se esqueça de adicionar o `require` dentro do arquivo `lob/loja_virtual`:

```

require File.expand_path('lib/livro')
require File.expand_path('lib/biblioteca')
require File.expand_path('lib/relatorio')

```

Agora basta adicionar alguns livros dentro da biblioteca e, após isso, criar um objeto do tipo `Relatorio` passando como dependência o objeto `Biblioteca`. Depois, invocamos o método `total` que acabamos de criar:

```

biblioteca = Biblioteca.new

teste_e_design = Livro.new "Mauricio Aniche", "123454",
                             247, 69.9, :testes
web_design_responsivo = Livro.new "Tárcio Zemel", "452565",
                                   189, 69.9, :web_design

biblioteca.adiciona teste_e_design
biblioteca.adiciona web_design_responsivo

relatorio = Relatorio.new biblioteca
p relatorio.total # => 139.8

```

Podemos melhorar esse código utilizando e conhecendo um

pouco mais da API Enumerable, que possui um método chamado `inject` , que simplifica muito o código:

```
class Relatorio
  def initialize(biblioteca)
    @biblioteca = biblioteca
  end

  def total
    @biblioteca.livros.inject(0)
    { |tot, livro| tot += livro.valor }
  end
end
```

O método `inject` recebe como primeiro argumento um valor que será um atributo da variável acumuladora, geralmente inicializado em 0. O segundo argumento é um bloco que recebe outros dois argumentos: o primeiro é a variável acumuladora, que foi inicializada anteriormente, no exemplo, 0; já o segundo argumento se refere a cada um dos livros existentes no `Array` de livros retornados pelo objeto `Biblioteca` .

A cada iteração, somamos o valor do objeto `Livro` à variável acumuladora que chamamos de `total` . No final, o valor desta variável é retornado como resultado do método.

Método map

A classe `Relatorio` precisa agora de um método que retorne o título de todos os livros que possuímos no objeto `Biblioteca` . Porém, a classe `Livro` não possui um atributo `titulo` , sendo assim, nosso primeiro passo será adicionar este atributo e logicamente um método acessor para visualizar o título do livro, já que o atributo será privado:

```
# coding: utf-8
```

```

class Livro
  attr_accessor :valor
  attr_reader :categoria, :autor, :titulo

  def initialize(titulo, autor, isbn = "1", numero_de_paginas,
                valor, categoria)

    @titulo = titulo
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @categoria = categoria
    @valor = valor
  end

  def to_s
    "Autor: #{@autor}, Isbn: #{@isbn},
      Páginas: #{@numero_de_paginas},
      Categoria: #{@categoria}"
  end

  def eql?(outro_livro)
    @isbn == outro_livro.isbn
  end

  def hash
    @isbn.hash
  end
end

```

O próximo passo é criar o método `titulos` na classe `Relatorio` que será responsável por retornar o título dos livros presentes na biblioteca:

```

class Relatorio
  def initialize(biblioteca)
    @biblioteca = biblioteca
  end

  def total
    @biblioteca.livros.inject(0)
    { |tot, livro| tot += livro.valor }
  end
end

```

```

def titulos
  titulos = []

  @biblioteca.livros.each do |livro|
    titulos << livro.titulo
  end

  titulos
end

biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                              247, 69.9, :testes

biblioteca.adiciona Livro.new "Design Responsivo",
                              "Tárcio Zemel", "45256", 189, 69.9, :web_design

relatorio = Relatorio.new biblioteca
p relatorio.titulos # => ["TDD", "Design Responsivo"]

```

O método `titulos` possui o mesmo problema do método `total` em sua primeira versão: muito código para uma tarefa extremamente simples. Novamente, se conhecermos a API `Enumerable`, podemos simplificar o código usando um método chamado `map`:

```

def titulos
  @biblioteca.livros.map { |livro| livro.titulo }
end

```

O método `map` itera sobre um `Array` e, para cada elemento (neste caso um objeto `Livro`), executa um bloco de código passado como argumento. O resultado da execução deste bloco é guardado dentro de um `Array` acumulador, que é retornado no final da iteração de todos os livros.

No código anterior, o resultado da execução do bloco é o

titulo do Livro. Como possuímos dois objetos Livro dentro da biblioteca, o retorno do método map será um Array contendo os títulos dos dois livros.

O método map possui o mesmo comportamento do método collect que foi visto anteriormente. Essencialmente o método collect foi criado para satisfazer programadores Smalltalk que possui um método similar. Já o método map possui o mesmo nome de métodos similares de linguagens como Scala.

Método map com notação simplificada

Existe uma maneira mais simples de passar um bloco na chamada ao método map :

```
class Relatorio
  def initialize(biblioteca)
    @biblioteca = biblioteca
  end

  def total
    @biblioteca.livros.inject(0)
      {|tot, livro| tot += livro.valor}
  end

  def titulos
    @biblioteca.livros.map &:titulo
  end
end
```

A expressão &:titulo cria um bloco como este: { |livro| livro.titulo }. O caractere & invoca um método to_proc no objeto, e passa este bloco para o método map .

Simplificando uso do método inject

Aprendemos que é possível simplificar o uso do método map

utilizando a notação `&:method`. O método `inject` possui uma simplificação quando, por exemplo, desejamos que os valores de todos os livros sejam somados. O código atual do método `total` é assim:

```
class Relatorio
  def total
    @biblioteca.livros.inject(0)
      {|tot, livro| tot += livro.valor}
  end
end
```

Podemos usar o método `map` para obter um novo `Array` apenas com os valores de todos os livros existentes:

```
class Relatorio
  def total
    @biblioteca.livros
      .map(&:valor).inject(0) {|tot, valor| tot += valor}
  end
end
```

O método `inject` possui uma variação, na qual é possível passar um `Symbol` em sua chamada, e não um bloco como argumento. Esta variação entende que o símbolo passado como argumento refere-se ao método que será chamado na variável acumuladora - no código anterior, seria o método `+`.

```
class Relatorio
  def total
    @biblioteca.livros.map(&:valor).inject(:+)
  end
end
```

O método `+` então será invocado automaticamente na variável acumuladora para cada uma das iterações feita, recebendo como argumento a variável `valor` que representa o item iterado. Outro detalhe é que não foi necessário inicializar a variável

acumuladora, automaticamente ela é criada com o valor 0.

5.7 PARA SABER MAIS: OUTRAS MANEIRAS DE CRIAR BLOCOS

Procs

Na seção anterior, executamos blocos de código através do método `yield`. Uma segunda maneira é recebermos o bloco de código como um argumento do tipo `Proc`. Vamos ver o código e depois discutiremos as diferenças:

```
class Biblioteca
  def initialize
    @livros = {} # Inicializa com um hash
  end

  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  def livros
    @livros.values.flatten
  end

  def livros_por_categoria(categoria, &bloco)
    @livros[categoria].each do |livro|
      bloco.call livro
    end
  end
end

biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                              247, 69.9, :testes

biblioteca.adiciona Livro.new "Design Responsivo",
```

```

        "Tárcio Zemel", "45256", 189, 69.9, :web_design

biblioteca.livros_por_categoria :testes do |livro|
  p livro.autor
end

=> "Mauricio Aniche"

```

O código é bastante parecido, porém possui duas importantes diferenças. A primeira é que passamos um argumento chamado `&bloco` para o método `livros_por_categoria`. O caractere `&` indica que estamos recebendo uma instância de `Proc` que, na realidade, é o bloco de código que vamos executar. A segunda diferença é que não se invoca o bloco de código chamando o método `yield`, agora invocamos o método `call` no argumento recebido na declaração do método (`bloco`).

O que ainda não fica muito claro é por que usaríamos uma `Proc` em vez de um simples `yield`?

Às vezes, precisamos executar o mesmo bloco de código várias vezes. Um objeto do tipo `Proc` guarda um bloco de código, e pode ser passado como parâmetro várias vezes para efetuar a chamada de um mesmo método, por exemplo:

```

class Biblioteca
  def initialize
    @livros = {} # Inicializa com um hash
  end

  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
  end

  def livros
    @livros.values.flatten
  end
end

```



```

    def livros_por_categoria(categoria)
      @livros[categoria].each do |livro|
        yield livro if block_given?
      end
    end
  end
end

biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                              247, 69.9, :testes

imprime_livro_no_console = Proc.new do |livro|
  p livro.autor
end

biblioteca.livros_por_categoria :testes, imprime_livro_no_console
# => ArgumentError: wrong number of arguments (2 for 1)

```

O que fizemos foi definir um objeto `Proc` na variável `imprime_livro_no_console` que guarda o bloco de código que havíamos criado anteriormente. Com isso, podemos passá-lo para qualquer método que recebe um bloco como argumento.

O único problema é que o código anterior não funciona, e a mensagem de erro parece bem estranha. Ela nos diz que estamos invocando um método que **deve** receber um argumento passando dois. Isso que dizer que o bloco não conta como argumento do método?

Exatamente. O método `livros_por_categoria` recebe apenas um argumento explícito chamado `categoria` e o outro é um bloco, que como na antiga implementação, não precisa necessariamente ser passado. Mas isso ainda não explica o erro retornado, afinal, tudo bem que o bloco pode ser ou não passado, mas no meu exemplo eu desejo passá-lo.

O que acontece é que a variável `imprime_livro_no_console`

que está aguardando a `Proc` que foi criada é um objeto, ou seja, quando invocamos o método `livros_por_categoria` passando o objeto `imprime_livro_no_console` como argumento, é como se estivéssemos passando qualquer outro argumento na chamada. Porém, sabemos que o método recebe apenas um argumento, obrigatoriamente.

O que precisamos fazer é transformar o objeto do tipo `Proc` `imprime_livro_no_console` em um bloco convencional e passar este bloco como argumento na chamada do método. Essa transformação é feita utilizando o caractere `&`:

```
biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                             247, 69.9, :testes

imprime_livro_no_console = Proc.new do |livro|
  p livro.autor
end

# transforma objeto proc em um bloco convencional
biblioteca.livros_por_categoria :testes,
  &imprime_livro_no_console
```

Ao adicionar o caractere `&` antes da variável que guarda o bloco que desejamos passar como argumento para o método `livros_por_categoria`, ele será automaticamente 'convertido' para um bloco convencional novamente.

Desta maneira, o método `livros_por_categoria` continua recebendo e executando um bloco através da chamada ao método `yield`. A vantagem é que o bloco em um objeto do tipo `Proc` pode ser reutilizado em outras partes do código.

Recebendo dois blocos como argumento

Caso seja necessário em algum momento receber dois blocos como argumento de um método, a utilização do `yield` torna-se obsoleta, já que não seria possível descobrir qual dos dois blocos deveria ser executado.

Neste caso, a solução é receber explicitamente os dois blocos como argumentos do método. Porém, como objetos do tipo `Proc` e não blocos convencionais.

```
class Biblioteca
  def livros_por_categoria(categoria, bloco_com_p,
                           bloco_com_puts)
    @livros[categoria].each do |livro|
      # Sem o &, transformamos o bloco em um objeto
      bloco_com_p.call livro
      bloco_com_puts.call livro
    end
  end
end

biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                              247, 69.9, :testes

imprime_livro_no_console = Proc.new do |livro|
  p livro.autor
end

imprime_livro_no_console_com_puts = Proc.new do |livro|
  puts livro.autor
end

biblioteca.livros_por_categoria :testes,
  imprime_livro_no_console,
  imprime_livro_no_console_com_puts
```

Como recebemos dois objetos do tipo `Proc`, não é possível invocá-los através do método `yield`, como foi dito. Devemos tratá-los como `Proc`s. Por isso, para executá-los, é necessário

invocar o método `call` em cada um deles.

Se o bloco passado como argumento receber mais de um argumento, basta invocar o método `call` passando-os separados por `,` (vírgula), assim como seria feito na chamada utilizando `yield`.

Lambdas

Existe outra maneira de guardar um bloco de código em uma variável e passá-lo como argumento na chamada de um método, são os lambdas, conhecidos também como funções anônimas:

```
biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                             247, 69.9, :testes

imprime_livro_no_console = lambda do |livro|
  p livro.autor
end

biblioteca.livros_por_categoria :testes,
  &imprime_livro_no_console
=> "Mauricio Aniche"
```

Os lambdas se parecem muito com procs, entretanto, existem duas diferenças. A primeira é que, quando utilizamos lambdas, ao contrário das procs, existe uma checagem da quantidade de parâmetros passados:

```
biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                             247, :testes

imprime_livro_no_console = lambda do
  p 'executou lambda'
end
```

```

imprime_livro_no_console_proc = Proc.new do
  p 'executou proc'
end

biblioteca.livros_por_categoria :testes,
  &imprime_livro_no_console_proc
=> NameError: undefined local variable or method 'livro'

biblioteca.livros_por_categoria :testes,
  &imprime_livro_no_console
=> ArgumentError: wrong number of arguments (0 for 1)

```

Quando invocamos o método `livros_por_categoria` passando um objeto do tipo `Proc`, na invocação do bloco nenhum argumento é esperado, mas mesmo assim o `p` colocado é executado. Quando o bloco passado é do tipo `Lambda`, acontece um erro: `ArgumentError` informando que o bloco deveria esperar pelo menos 1 argumento, mas nenhum foi declarado.

A segunda diferença acontece quando utilizamos `return` dentro de um `lambda`. Quando declaramos um `return` dentro de um `lambda`, o método que invocou o bloco receberá o retorno deste bloco e continuará com sua execução normalmente. Se usarmos um `return` dentro de uma `proc`, o método que invocou o bloco será interrompido e seu retorno será o valor retornado na sua execução.

Vamos ver um pequeno exemplo para deixar as coisas mais claras:

```

def proc_com_return
  Proc.new { return "retornando algo de uma proc" }.call
  "Proc finalizada"
end

def lambda_com_return
  lambda { return "retornando algo de um lambda" }.call

```

```

    "Lambda finalizado"
end

puts proc_com_return
puts lambda_com_return

# => retornando algo de uma proc
# => Lambda finalizado

```

Procs são como procedures, apenas parte de código que serão executados como parte do comportamento de um método. Lambdas se parecem mais com métodos, já que existe checagem do número de parâmetros e não sobrescrevem o `return` definido no método que os invocaram. Existe apenas um caso no qual é **necessário** obrigatoriamente invocar um método passando uma instância de `lambda` e não um instância de `proc` :

```

def metodo_que_recebe_um_bloco(bloco)
  bloco.call
  "retornando algo do método"
end

p metodo_que_recebe_um_bloco(
  Proc.new { return "Retorno da Proc" })
p metodo_que_recebe_um_bloco(
  lambda { return "Retorno do Lambda" })

# => LocalJumpError: unexpected return
# => retornando algo do método

```

O interpretador Ruby não permite que um argumento passado para um método contenha um `return` . Em nosso exemplo, a `Proc` possui um `return` explícito, por este motivo, acontece o erro: `LocalJumpError` . Como lambdas se comportam como métodos, elas podem conter um `return` explícito.

5.8 PARA SABER MAIS: CURRYING

Currying é uma técnica muito usada em linguagens funcionais, que consiste em transformar uma função que recebe múltiplos argumentos em uma sequência de funções que recebem um único argumento. A definição parece um pouco complicada, mas pode ser simplificada com um pouco de código, vamos ver:

```
executa_comando =  
    lambda { |conexao, comando| conexao.executa comando }  
executa_comando.call Conexao.new, Update.new
```

O código anterior simula a execução de um comando qualquer (no exemplo um `Update`) em um banco de dados através da chamada de um método `executa` , em um objeto do tipo `conexao` . Criamos um lambda que recebe dois argumentos: a conexão com o banco e o comando que desejamos executar. Depois, executamos este lambda passando os argumentos necessários.

Em determinado momento, pode ser útil executar o lambda `executa_comando` passando um comando `Insert` , como demonstrado no código a seguir:

```
executa_comando =  
    lambda { |conexao, comando| conexao.executa comando }  
executa_comando.call Conexao.new, Update.new  
executa_comando.call Conexao.new, Insert.new
```

Repare que sempre que vamos executar o lambda `executa_comando` precisamos passar a conexão; o único parâmetro que mudou foi o `comando` . Para melhorar este código, podemos utilizar a técnica de currying:

```
executa_comando =  
    lambda { |conexao, comando| conexao.executa comando }  
                                .curry  
executa_comando_com_conexao = executa_comando.call Conexao.new
```

```
executa_comando_com_conexao.call Update.new  
executa_comando_com_conexao.call Insert.new
```

Quando invocamos o método `curry` no objeto `Lambda`, nós o particionamos em duas funções, pois o `lambda` recebe dois argumentos. A primeira função que receberá o objeto `Conexao`, quando invocamos esta primeira função, retorna a segunda função, que neste exemplo, receberá o argumento referente ao comando que desejamos executar: `Insert` ou `Update`.

Ao invocar esta segunda função, o conteúdo definido inicialmente no objeto `lambda` criado será finalmente executado e a operação será efetuada utilizando a conexão e o comando que foram passados.

A grande vantagem do uso de `currying` é quando vamos executar o mesmo bloco de código, seja ele um `lambda` ou uma `proc`, e alguns parâmetros são sempre os mesmos. Usando esta técnica, podemos fixar alguns argumentos com o mesmo valor, independente da quantidade de vezes que o bloco for executado. Também é muito útil quando queremos deixar o código mais claro e conciso, por exemplo:

```
multiplicador = lambda { |x, y| x * y }  
p multiplicador.call 2, 13  
# => 26
```

O código anterior é bem simples: ele cria um `lambda` que multiplica dois argumentos passados `x` e `y`, e retorna o resultado. No exemplo, foi calculado o dobro do valor 13. Agora desejamos calcular o dobro do número 43, então fazemos isto com uma simples chamada ao `multiplicador`:

```
multiplicador = lambda { |x, y| x * y }
```



```
p multiplicador.call 2, 13
p multiplicador.call 2, 43
```

```
# => 26
# => 86
```

Utilizando currying, podemos deixar o código um pouco mais legível:

```
multiplicador = lambda { |x, y| x * y }.curry
dobro = multiplicador.call 2

p dobro.call 13
p dobro.call 43
```

```
# => 26
# => 86
```

Particionamos o lambda em duas funções, a primeira recebe o argumento `x` . E ao invocar esta primeira função passando como argumento o valor `2` , a chamamos de `dobro` , afinal, não importa qual seja o valor passado na chamada da segunda função, o valor de `x` está fixo em `2` .

5.9 PARA SABER MAIS: CLOSURE

Quando criamos um bloco de código (lambda, proc ou bloco convencional), ele possui acesso às variáveis visíveis do escopo onde foram criadas, e qualquer alteração nestas variáveis será refletida no escopo original. Ou seja, os blocos possuem uma espécie de link com as variáveis definidas no escopo de origem. Por exemplo:

```
soma = 0

[1, 3, 5, 6, 9].each do |numero|
  soma += numero
end
```

```
p soma # => 24
```

Repare que o bloco de código que definimos e passamos na chamada ao método `each` possui acesso à variável `soma`, que foi definida no escopo onde o bloco foi criado. Esta capacidade de fazer **bind** com as variáveis que foram definidas no escopo no qual o bloco foi criado é conhecida como **closure**.

Este comportamento pode ser confirmado ao executarmos o seguinte exemplo:

```
def imprime_numero
  numero = 134
  yield
end

def chama_metodo_imprime_numero
  numero = 42
  imprime_numero do
    puts "O número aqui é: #{numero}"
  end
end

chama_metodo_imprime_numero # => 42
```

O número impresso é 42 porque, no escopo onde o bloco foi criado, a variável `numero` possui o valor 42. No contexto em que o bloco foi **invocado**, existe uma outra variável `numero` com o valor 134, porém esta segunda variável foi definida no contexto do método `imprime_numero` e não é visível ao bloco.

5.10 PRÓXIMOS PASSOS

Neste capítulo, aprendemos conceitos importantes de um outro paradigma de programação: o funcional. Existem vários conceitos deste paradigma espalhados pelos mais importantes

códigos escritos em Ruby mundo afora. Portanto, posso afirmar que você verá e utilizará estes recursos, principalmente os blocos, que na minha opinião são os mais importantes e que podem contribuir para que você crie códigos elegantes e legíveis.

Também aprendemos a utilizar um pouco da API Enumerable, que possui vários métodos úteis para trabalharmos com coleções em Ruby. Conhecer esta API é um excelente caminho para criar códigos mais legíveis e funcionais quando lidamos com coleções, então, estude-a. O link da API, <http://ruby-doc.org/core-1.9.3/Enumerable.html>, estará sempre aberto quando estiver trabalhando ou estudando.

No próximo capítulo, veremos um pouco da API de File do Ruby. Ela será muito importante para criarmos um espécie de framework que salva e recupera dados dos livros criados utilizando arquivos em disco.

EXPLORANDO API FILE

Agora que conhecemos e aprendemos a utilizar blocos, temos todo o conhecimento necessário para partirmos para API do Ruby que lida com arquivos e diretórios. Usaremos essa API para salvarmos os objetos que representam `Livro` em nosso sistema. Veremos também como trabalhar com serialização e deserialização de objetos, que será a maneira que salvaremos nossos objetos em disco.

6.1 UM POUCO DA CLASSE FILE

A classe `File` da API do Ruby é uma abstração de objetos que são criados para representar um arquivo. Podemos, por exemplo, descobrir qual o tamanho, em bytes, de um arquivo salvo em disco:

```
arquivo_temporario = File.new("/tmp/arquivo")  
p arquivo_temporario.size # => 564
```

A API de arquivos possui uma grande variedade de métodos que utilizam blocos como forma de interação. Podemos criar um arquivo e incluir dados dentro dele com apenas uma linha de código:

```
File.open("/tmp/arquivo", "w") do |arquivo_temporario|  
  arquivo_temporario.puts "primeira linha do meu arquivo"  
end
```

ESCREVENDO COM O MÉTODO WRITE

Podemos escrever dentro de um arquivo utilizando o método `write` em vez do método `puts`. A principal diferença é que o método `puts` adiciona uma quebra de linha `\n` no final da `String` que foi incluída no arquivo, enquanto o método `write` não o faz.

Dado que temos o arquivo texto `/tmp/arquivo` salvo, podemos abri-lo e imprimir cada uma de suas linhas:

```
arquivo_temporario = File.open "/tmp/arquivo", "r"
arquivo_temporario.each do |linha|
  p linha # => "primeira linha do meu arquivo\n"
end
```

MODOS DE ABRIR UM FILE

O segundo parâmetro passado quando invocamos o método `open` é o modo que desejamos abrir o arquivo. As maneiras mais comuns são:

- `r` - abre o arquivo somente para leitura.
- `w` - abre o arquivo somente para escrita (sobrescreve todo o conteúdo do arquivo se este existir).
- `w+` - abre o arquivo tanto para leitura quanto para escrita (sobrescreve todo o conteúdo do arquivo se este existir).
- `a` - abre o arquivo somente para escrita (começa a escrita no final da última linha existente se o arquivo já existir).

A API `File` é muito rica em detalhes, e por isso poderíamos gastar algumas páginas explicando todos os métodos e suas utilidades. Porém, vamos fazer algo mais útil, vamos começar a salvar os objetos da classe `Livro` que estamos criando e salvá-los em arquivos para posteriormente poder ler estes dados e recuperar os objetos.

Entretanto, é necessário antes criarmos a classe que fará este trabalho.

6.2 SERIALIZAÇÃO DE OBJETOS

Serialização nada mais é do que o processo de salvar um objeto

utilizando um sistema de armazenamento qualquer, como por exemplo, um arquivo, ou até mesmo transmiti-lo pela rede. Estes dados são salvos em formato binária ou formato texto, como XML, JSON, YML etc., e podem ser usados posteriormente para recriar um objeto em memória com o mesmo estado em que ele foi armazenado.

O Ruby possui dois mecanismos de serialização de objetos nativos da própria linguagem. Um deles serializa os objetos em um formato que é fácil de ser lido por um ser humano, enquanto o outro serializa em um formato binário.

Vamos utilizar e explorar o formato humano de serialização que é representando pelo formato YAML (<http://ruby-doc.org/stdlib-2.1.2/libdoc/yaml/rdoc/YAML.html>). Qualquer objeto pode ser serializado para o formato YAML sem o mínimo esforço, gastando apenas algumas linhas de código:

```
require 'yaml'

teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",
                           247, 69.9, :testes

objeto_serializado = YAML.dump teste_e_design
p objeto_serializado
```

A saída desse código será algo como:

```
 "--- !ruby/object:Livro \nautor: Mauricio Aniche\nisbn: 247\n\nnumero_de_paginas: \"123454\"\n\ntitulo: TDD\nvalor: :testes\n"
```

A String que foi impressa, e que parece um pouca estranha, é a representação do objeto `teste_e_design` no formato texto YAML. Ela pode ser facilmente adicionada dentro de um arquivo.

Podemos deserializar o `YAML` que foi criado, e criar uma outra instância de `Livro`, com as mesmas informações e estado do objeto `teste_e_design`:

```
outro_teste_e_design = YAML.load objeto_serializado
p outro_teste_e_design # => Autor: Mauricio Aniche, Isbn: 123454,
                        Páginas: 247, Categoria: testes
```

6.3 SALVANDO OBJETOS EM ARQUIVOS

Nos capítulos anteriores, guardamos os objetos do tipo `Livro` dentro de um `Hash`, separando-os por `categoria`. Porém, agora vamos salvá-los também em arquivos, logo a nossa estratégia será criar um arquivo com todos os objetos do tipo `Livro` que forem criados e adicionados na biblioteca.

O ideal é deixar a lógica que lida com a API `File` separada da lógica existente na classe `Biblioteca`, por isso vamos criar uma nova classe chamada `BancoDeArquivos` dentro do arquivo `lib/banco_de_arquivos.rb`, já definindo o método `salva`:

```
require 'yaml'

class BancoDeArquivos
  def salva(livro)
    File.open("livros.yml", "a") do |arquivo|
      arquivo.puts YAML.dump(livro)
      arquivo.puts ""
    end
  end
end
```

Para testar este código, precisamos fazer o `require` deste novo arquivo, adicionando-o dentro do arquivo `lib/loja_virtual`:

```
require File.expand_path('lib/livro')
```



```
require File.expand_path('lib/biblioteca')
require File.expand_path('lib/relatorio')
require File.expand_path('lib/banco_de_arquivos')
```

E agora sim podemos testar o comportamento implementado no método `salva` :

```
teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",
                           247, 69.9, :testes
```

```
BancoDeArquivos.new.salva teste_e_design
```

Abrimos o arquivo `livros.yaml` em modo de escrita, respeitando o conteúdo já existente e inserindo novos conteúdos no final do arquivo. É importante ressaltar que separamos os objetos serializados através de duas linhas. O conteúdo do arquivo é similar ao conteúdo a seguir:

```
--- !ruby/object:Livro
titulo: TDD
autor: Mauricio Aniche
isbn: '123454'
numero_de_paginas: 247
categoria: :testes
valor: 69.9
```

Na classe `Biblioteca` , guardamos os dados em um `Hash` e agora vamos também invocar o método `salva` para guardar os dados dentro do arquivo `livros.yaml` :

```
class Biblioteca
  def initialize
    @livros = {} # Inicializa com um hash
    # Inicializa banco de arquivos
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(livro)
    @livros[livro.categoria] ||= []
    @livros[livro.categoria] << livro
    @banco_de_arquivos.salva livro
  end
end
```

```

end

def livros
  @livros.values.flatten
end

def livros_por_categoria(categoria)
  @livros[categoria].each do |livro|
    yield livro if block_given?
  end
end

end

teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",
                           247, 69.9, :testes

biblioteca = Biblioteca.new
biblioteca.adiciona teste_e_design

```

Podemos deixar o código ainda mais elegante utilizando blocos:

```

class Biblioteca
  def initialize
    @livros = {}
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(livro)
    salva livro do
      @livros[livro.categoria] ||= []
      @livros[livro.categoria] << livro
    end
  end

  def livros
    @livros.values.flatten
  end

  def livros_por_categoria(categoria)
    @livros[categoria].each do |livro|
      yield livro if block_given?
    end
  end
end

```

```

private

def salva(livro)
  @banco_de_arquivos.salva livro
  yield
end
end

```

MÉTODOS PRIVADOS

O método `salva` pode ser privado, pois não existe necessidade de invocá-lo de fora da classe `Biblioteca`. Para criar métodos privados em Ruby, basta invocar o método `private` e, abaixo dele, declarar os métodos que deseja que sejam privados. O ideal é deixá-los sempre no final do arquivo onde a classe está definida.

6.4 RECUPERANDO OBJETOS SALVOS

Vamos alterar um pouco a classe `Biblioteca`. O primeiro passo será guardar os objetos em um `Array`. Com isso, será necessário alterar o método `livros_por_categoria` para que ele continue retornando um `Array` dos livros da categoria que é passada como argumento:

```

class Biblioteca
  attr_reader :livros

  def initialize
    @livros = [] # Inicializando com Array
    @banco_de_arquivos = BancoDeArquivos.new
  end
end

```

```

def adiciona(livro)
  salva livro do
    @livros << livro
  end
end

def livros_por_categoria(categoria)
  @livros.select { |livro| livro.categoria == categoria }
end

private

def salva(livro)
  @banco_de_arquivos.salva livro
  yield
end
end

```

Agora o método `livro_por_categoria` usa o método `select` existente na classe `Array`. O método `livros` que consolidava todos os livros existentes dentro do `Hash` foi removido e agora existe apenas um método leitor, o `livros`, que retorna a variável `@livros`.

Vamos agora inicializar o `Array @livros` com os dados existentes dentro do arquivo `livros.yml`. Para isso, será preciso deserializar os objetos existentes dentro do arquivo e transformá-los novamente em instâncias de `Livro`, usando a classe `YAML`:

```

class BancoDeArquivos
  def salva(livro)
    File.open("livros.yml", "a") do |arquivo|
      arquivo.puts YAML.dump(livro)
      arquivo.puts ""
    end
  end

  def carrega
    $/ = "\n\n"
    File.open("livros.yml", "r").map do |livro_serializado|
      YAML.load livro_serializado
    end
  end
end

```

```

        end
    end
end

```

O método `carrega` configura em sua primeira linha o separador de linhas do arquivo para trabalhar com duas quebras (`\n\n`), depois o arquivo `livros.yml` é aberto em modo leitura. A cada iteração, o conteúdo serializado é transformado em uma instância de `Livro` através do método `load` da classe `YAML` . Como utilizamos o método `map` , o retorno do método `carrega` é um `Array` , como todos os objetos que foram deserializados.

Podemos então carregar os objetos serializados presentes no arquivo e adicioná-los no `Array` de objetos `Livro` da classe `Biblioteca` , representado pela variável `@livros` :

```

class Biblioteca
  def initialize
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(livro)
    salva livro do
      livros << livro
    end
  end

  def livros_por_categoria(categoria)
    livros.select { |livro| livro.categoria == categoria }
  end

  def livros
    @livros ||= @banco_de_arquivos.carrega
  end

  def salva(livro)
    @banco_de_arquivos.salva livro
    yield
  end
end

```

Usamos a variável `@livros` com carregamento **lazy**, ou seja, só buscamos os objetos no `BancoDeArquivos` quando a variável for utilizada pela primeira vez, por isso removemos a inicialização da variável no construtor. Assim, as chamadas seguintes retornam o conteúdo carregado pela primeira vez. Os métodos `adiciona` e `livros_por_categoria` são os responsáveis por invocar o método privado `livros`, por isso não acessam mais diretamente a variável `@livros`.

Método select

O método `select` da API `Enumerable` percorre todos elementos existentes na coleção onde o método foi chamado, e retorna um `Array` contendo os elementos iterados cuja a execução do bloco passado seja `true`. Ele funciona como uma espécie de filtro dos objetos que compõe o `Array`.

No exemplo anterior, se existirem três objetos do tipo `Livro` (dois pertencentes à categoria `:web` e outro, à categoria `:testes`) e invocarmos o método `select` definindo que o retorno do bloco é a comparação da categoria do `Livro` com uma categoria `:testes`, o retorno será um `Array` contendo apenas livros cuja categoria for `:testes`:

```
biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche", "123454",
                             247, 69.9, :testes
biblioteca.adiciona Livro.new "Web Design Responsivo",
                             "Tárcio Zemel", "45256", 240, 69.9, :web
biblioteca.adiciona Livro.new "Web com JSF e JPA",
                             "Gilliard Cordeiro", "543232", 270, 69.9, :web

biblioteca.livros_por_categoria :testes
# => Autor: Mauricio Aniche, Isbn: 123454, Páginas: 247,
```

6.5 PRÓXIMOS PASSOS

Neste capítulo, vimos um pouco sobre a API `File` do Ruby, aprendemos a criar e ler arquivos do disco. Também revimos alguns conceitos de serialização que estavam perdidos em nossa memória e aprendemos que, em Ruby, existem duas maneiras nativas de serializar objetos: `YAML` que é uma serialização mais fácil de ser lida por seres humanos; e `Marshal` que serializa os dados em um formato binário.

O próximo assunto será o compartilhamento de código entre as classes e objetos Ruby. Veremos maneiras de aplicar o princípio DRY (*Dont Repeat Yourself*) em nossos códigos, utilizando técnicas existentes nos principais frameworks e bibliotecas do mundo Ruby.

COMPARTILHANDO COMPORTAMENTOS: HERANÇA, MÓDULOS E MIXINS

Uma das principais característica de um bom design de código é a eliminação de duplicidades desnecessárias. O princípio DRY (*Don't Repeat Yourself*) criado por Andy Hunt e Dave Thomas, no excelente livro *The Pragmatic Programmer*, propõe que cada pequena quantidade de código deve possuir somente uma representação em todo o sistema.

Em sistemas orientados a objetos, as classes são abstrações que permitem que determinados comportamentos sejam isolados juntos com os dados que representam uma entidade no sistema. Na nossa aplicação, a classe `Livro` cumpre este papel, de maneira que todos os métodos criados estão disponíveis entre todas as instâncias criadas a partir da classe `Livro`.

No futuro, podemos decidir vender além de livros, outras mídias como DVDs, CDs etc. Estes objetos certamente possuem dados bastante parecidos, como por exemplo, o preço, já que

estamos lidando com uma aplicação de vendas. Podemos até mesmo criar relatório dos produtos que temos disponíveis, e precisamos criar uma classe que seja genérica para, quando criarmos algum outro tipo de mídia, não precisarmos alterar a classe de relatório, ou mesmo outros pontos do sistemas que lidam com Produto .

Neste capítulo, veremos formas de compartilhar comportamentos em Ruby. O primeiro deles é bem conhecido pelos desenvolvedores do mundo da Orientação a Objetos: **herança**. Também conheceremos uma maneira que é bem mais utilizada e conhecida pela comunidade Ruby, mas que não substitui o uso de herança, os **mixins**.

Discutiremos as vantagens de cada uma destas abordagens e veremos quando devemos utilizar uma ou outra.

7.1 HERANÇA: COMPARTILHANDO COMPORTAMENTOS COM CLASSES

Nos capítulos anteriores, usamos bastante o método `p` para imprimir dados na saída padrão. Aprendemos que o método `p` sempre invoca um método chamado `inspect` no objeto que pedimos para ser impresso. O fato é que nós nunca implementamos esse método em nenhuma das classes que criamos até agora.

Quando invocamos qualquer método em um objeto, na verdade estamos enviando uma mensagem para ele e solicitando que seja executado algum comportamento. A linguagem Ruby procura qual método deve ser executado e, quando o encontra,

executa-o. Por exemplo, quando invocamos o método `inspect` em uma instância de `Livro`, o interpretador Ruby vai procurar o método na instância que representa e possui todos os métodos definidos na classe `Livro`. Caso não encontre, o interpretador tenta encontrá-lo em alguma **superclasse** da classe `Livro`:

```
p Livro.superclass # => Object
```

E encontra o método `inspect` definido na classe `Object`, que é a superclasse **default** de todos os objetos que não estendem explicitamente de outra classe.

HERANÇA SIMPLES

Em Ruby, podemos fazer com que uma classe herde apenas de uma **única** outra classe, o que caracteriza o que chamamos de **herança simples**.

```
p Livro.superclass.methods # => [..., :inspect, ...]
```

LISTANDO OS MÉTODOS DE UMA CLASSE

O método `methods` retorna todos os métodos disponíveis para os objetos que são criados a partir da classe ou de **subclasses**. Podemos, por exemplo, listar os métodos da classe `Object` e assim concluirmos que ela disponibiliza o método `inspect` para os objetos do tipo `Livro`, já que `Livro` herda de `Object`.

Vamos começar a lidar com outros objetos em nossa loja virtual. Agora vamos criar DVD s, e já conseguimos pensar em abstrações para estas duas classes, por exemplo, todo Livro e DVD possuem um titulo , valor e uma categoria . Portanto, criaremos uma classe Midia , que ficará dentro do arquivo lib/midia.rb , e que também será a superclasse das classes Livro e DVD . Esta também não foi criada e deve ser adicionada no arquivo lib/dvd.rb .

```
class Midia
end

# coding: utf-8
class DVD < Midia
  attr_accessor :valor
  attr_reader :titulo

  def initialize(titulo, valor, categoria)
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Titulo: #{@titulo}, Valor: #{@valor} }
  end
end

# coding: utf-8
class Livro < Midia
  attr_accessor :valor
  attr_reader :categoria, :autor, :titulo

  def initialize(titulo, autor, isbn = "1", numero_de_paginas,
                valor, categoria)
    @titulo = titulo
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @categoria = categoria
    @valor = valor
  end
end
```

```

end

def to_s
  "Autor: #{@autor}, Isbn: #{@isbn},
    Páginas: #{@numero_de_paginas},
    Categoria: #{@categoria}"
end
end

```

Não se esqueça de fazer o `require` destas duas novas classes no arquivo `lib/loja_virtual.rb` :

```

require File.expand_path('lib/midia')
require File.expand_path('lib/dvd')
require File.expand_path('lib/livro')
require File.expand_path('lib/biblioteca')
require File.expand_path('lib/relatorio')
require File.expand_path('lib/banco_de_arquivos')

```

Quando criamos uma classe que herda comportamento de outra, utilizamos o caractere `<` . Definimos, por exemplo, que a classe `Livro < Midia` herda todos os comportamentos existentes na classe `Midia` e em todas as superclasses também.

As classes `Livro` e `DVD` definem um método acessor para o atributo `valor` e um método de leitura para o atributo `titulo` . Quando invocamos o método `attr_reader :titulo` , estamos definindo um método `titulo` para os objetos da classe `Livro` e `DVD` , no caso do método `attr_accessor :valor` definimos dois métodos: `valor` e `valor=(novo_valor)` . Já que este código está sendo repetido, vamos extraí-lo para dentro da classe `Midia` , e assim as classes `Livro` e `DVD` as herdarão:

```

class Midia
  attr_accessor :valor
  attr_reader :titulo
end

# coding: utf-8

```

```

class DVD < Midia
  def initialize(titulo, valor, categoria)
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end

# coding: utf-8
class Livro < Midia
  attr_reader :categoria, :autor

  def initialize(titulo, autor, isbn = "1", numero_de_paginas,
                valor, categoria)

    @titulo = titulo
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @categoria = categoria
    @valor = valor
  end

  def to_s
    "Autor: #{@autor}, Isbn: #{@isbn},
    Páginas: #{@numero_de_paginas},
    Categoria: #{@categoria}"
  end
end

teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",
                           247, 69.9, :testes

p teste_e_design.valor # => 69.9
p teste_e_design.titulo # => "TDD"

windows = DVD.new "Windows 7 for Dummies", 98.9,
                  :sistemas_operacionais
p windows.valor # => 98.9
p windows.titulo # => Windows 7 for Dummies

```

7.2 HERANÇA E VARIÁVEIS DE INSTÂNCIA

As mídias da nossa loja virtual possuem um desconto particular de cada uma delas que é aplicado ao seu valor. No método `initialize` da classe `Livro`, vamos definir uma variável de instância que guardará o valor do desconto, e faremos o mesmo no método `initialize` da classe `DVD`. Após isso, definiremos um método `valor_com_desconto` diretamente na classe `Midia`, já que este será um comportamento padrão das duas mídias existentes:

```
class Midia
  attr_accessor :valor
  attr_reader :titulo

  def valor_com_desconto
    @valor - (@valor * @desconto)
  end
end

# coding: utf-8
class DVD < Midia
  def initialize(titulo, valor, categoria)
    @titulo = titulo
    @valor = valor
    @categoria = categoria
    @desconto = 0.1
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end

# coding: utf-8
class Livro < Midia
  attr_reader :categoria, :autor

  def initialize(titulo, autor, isbn = "1", numero_de_paginas,
                valor, categoria)
```

```

        @titulo = titulo
        @autor = autor
        @isbn = isbn
        @numero_de_paginas = numero_de_paginas
        @categoria = categoria
        @valor = valor
        @desconto = 0.15
    end

    def to_s
        "Autor: #{@autor}, Isbn: #{@isbn},
        Páginas: #{@numero_de_paginas},
        Categoria: #{@categoria}"
    end
end

teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",
                          247, 69.9, :testes

p teste_e_design.valor_com_desconto # => 59.41

windows = DVD.new "Windows 7 for Dummies", 98.9,
                  :sistemas_operacionais
p windows.valor_com_desconto # => 89.01

```

O método `valor_com_desconto` existe nas instâncias de `Livro` e `DVD` como esperado. Porém, existe algo interessante no código anterior: o método `valor_com_desconto` acessa duas variáveis de instância, `@valor` e `@desconto`, que estão definidas nas subclasses. Em outras linguagens, este comportamento não funcionaria, exceto se as variáveis estivessem também definidas na classe `Midia`.

Este comportamento é bastante particular da linguagem Ruby, quando falamos de variáveis de instância e herança. Você vai entender isso melhor nas próximas linhas.

As classes `Livro` e `DVD`, assim como outras mídias que surgirão, terão um valor de desconto pré-definido em 10%. Sendo

assim, vamos criar uma variável de instância chamada `@desconto` dentro da classe `Midia`, e inicializá-lo com o valor `0.1`:

```
class Midia
  attr_accessor :valor
  attr_reader :titulo

  def initialize
    @desconto = 0.1
  end

  def valor_com_desconto
    @valor - (@valor * @desconto)
  end
end
```

Podemos assim eliminar a variável `@desconto` da classe `DVD`:

```
# coding: utf-8
class DVD < Midia
  def initialize(titulo, valor, categoria)
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end
```

Agora podemos testar novamente o método `valor_com_desconto` e o comportamento deve ser obviamente o mesmo:

```
windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_com_desconto
# => TypeError: nil can't be coerced into Float
```

Aconteceu um erro que não esperávamos, e ele nos indica que a variável `@desconto` está nula. Isso aconteceu porque não

invocamos o método `initialize` da superclasse que inicializa a variável `@desconto`. Resolvemos isso invocando o método `super` dentro do `initialize` da classe `DVD`, que invocará por sua vez o `initialize` da classe `Midia`:

```
class DVD < Midia
  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_com_desconto # => 89.01
```

Porém, ainda não ficou claro como o método `valor_com_desconto` possui acesso à variável `@valor`. Todo o objeto Ruby possui um conjunto de variáveis de instância, e estas variáveis não são definidas na classe, mas sim quando invocamos algum método que as cria. Este, na maioria dos casos, acaba sendo o próprio método `initialize`, que é um simples método como outro qualquer.

Pelo motivo de não serem definidas na classe, variáveis de instância **não são herdadas** por subclasses. Quando invocamos o método `initialize` da superclasse (através do método `super`), uma variável de instância é criada no escopo onde foi chamada. No exemplo, a variável é criada dentro do objeto `DVD`.

Por este motivo, o método `valor_com_desconto` funciona.

Ao criarmos um objeto do tipo `Livro` ou `DVD`, ele possui uma variável de instância chamada `@valor` e outra chamada `@desconto`. Quando invocamos o método `valor_com_desconto`, as variáveis referenciadas pelo método são as do objeto no qual ele foi chamado: `Livro` ou `DVD`.

```
teste_e_design = Livro.new "TDD", "Mauricio Aniche", "123454",  
                           247, 69.9, :testes  
  
p teste_e_design.valor_com_desconto # => 59.41  
  
windows = DVD.new "Windows 7 for Dummies", 98.9,  
                  :sistemas_operacionais  
p windows.valor_com_desconto # => 89.01
```

Podemos provar essa característica da linguagem se tentarmos invocar o método `valor_com_desconto` em uma instância de `Midia`:

```
midia = Midia.new  
midia.valor_com_desconto  
# => undefined method '*' for nil:NilClass
```

O erro indica que a variável `@valor` está nula. E faz bastante sentido, pois em nenhum momento nós criamos uma variável `@valor` no conjunto de variáveis de um objeto do tipo `Midia`.

Para resolver este problema, basta inicializar objetos do tipo `Midia` com um valor padrão, por exemplo:

```
class Midia  
  attr_accessor :valor  
  attr_reader :titulo  
  
  def initialize  
    @desconto = 0.1  
    @valor = 10.0  
  end  
end
```

```

    def valor_com_desconto
      @valor - (@valor * @desconto)
    end
  end
end

```

```

midia = Midia.new
midia.valor_com_desconto # => 9.0

```

Agora que o método `initialize` da classe `Midia` define uma variável `@valor`, e o método `initialize` da classe `DVD` define a mesma variável com outro valor, qual será o seu valor no momento da chamada ao método `valor_com_desconto` em uma instância de `DVD`? Vamos conferir:

```

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_com_desconto # => 89.01

```

Exatamente o mesmo valor que antes. Isso acontece porque não herdamos a variável `@valor` no momento da chamada ao método `super`, apenas foi criada uma variável `@valor` valendo `10.0` no conjunto de variáveis de instância do objeto `windows`. Como logo em seguida redefinimos o valor da variável `@valor` para `98.9` (valor do atributo recebido no método `initialize`), este será o valor no momento da execução do método `valor_com_desconto`.

Esta é uma característica importante de herança em Ruby, pois não herdar variáveis de instância exclui a possibilidade de que uma delas seja utilizada em uma subclasse, sombreando uma variável que foi definida na superclasse.

Como invocar o método `initialize` da superclasse

O primeiro ponto que devemos ressaltar é que Ruby não possui

construtores. Existe um método `initialize` que é executado quando criamos um objeto ao invocar o método `new` em uma constante definida.

Como Ruby possui apenas métodos, quando desejamos invocar o método `initialize` da superclasse, devemos invocá-lo por meio da palavra chave `super`. Porém existe uma pequena armadilha.

Quando invocamos o `initialize` da superclasse apenas com `super` sem os parênteses, o interpretador Ruby tentará invocá-lo passando os mesmos parâmetros recebidos pelo método `initialize` da subclasse, por exemplo:

```
# coding: utf-8
class DVD < Midia
  def initialize(titulo, valor, categoria)
    super # invocando super sem os parâmetros
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
# => ArgumentError: wrong number of arguments (3 for 0)
```

Os parâmetros `titulo`, `valor` e `categoria` são repassados para o método `initialize` da classe `Midia` que não recebe nenhum argumento, por este motivo o erro.

Para resolver, basta adicionar os parênteses na chamada `super()` que o método `initialize` da superclasse será

chamado sem a passagem de nenhum argumento. Se o método `initialize` da superclasse recebe argumentos, você deve passá-los - neste caso, o uso dos parênteses é opcional.

7.3 OS CUSTOS NO USO DA HERANÇA

A maioria dos programadores Ruby já projetaram classes que utilizam herança para modelar as classes que escrevem no dia a dia. De fato, esta maneira de projetar classes é muito comum e quase nunca pensamos nos problemas que essa abordagem pode nos trazer.

Não existe encapsulamento entre os objetos

Utilizamos herança geralmente para modelar classes nas quais desejamos compartilhar comportamentos, mas o que muita gente não sabe é que muitas vezes acabamos compartilhando implementação. Entre outras coisas, isso significa que não importa de quantas classes você herde, todos os métodos e o estado do seu objeto são definidos em um único *namespace*. E se você não tomar o devido cuidado, essa perda de encapsulamento entre os objetos podem trazer grandes dores de cabeça.

Para testar esse comportamento, vamos refatorar o método `valor_com_desconto` :

```
class Midia
  attr_accessor :valor
  attr_reader :titulo

  def initialize
    @desconto = 0.1
    @valor = 10.0
  end
end
```

```

    def valor_com_desconto
      @valor - desconto
    end

    private

    def desconto
      @valor * @desconto
    end
  end
end

```

Apenas extraímos parte do código para um método privado chamado `desconto`, e o comportamento continua o mesmo. Mas é desta refatoração que vem o problema. Hoje, apenas as classes `Livro` e `DVD` estendem a classe `Midia`, porém, no futuro outras mídias podem ser criadas. E em alguma destas mídias, o desenvolvedor resolve criar, por exemplo, um método de leitura que retorna o valor do desconto definido, sem saber que existe um outro método com o mesmo nome, definido na classe `Livro`:

```

class CD < Midia
  attr_reader :desconto

  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
    @desconto = 0.3
  end
end

windows = CD.new "Windows 95", 239.9, :sistemas_operacionais
p windows.valor_com_desconto # => 239.6

```

Perceberam o problema? O método `desconto` definido na classe `Livro` foi invocado. Mas por quê?

Lembre-se de que, quando invocamos um método em Ruby,

estamos na verdade enviando uma mensagem para um objeto, e no momento em que invocamos o método `valor_com_desconto`, ele percebe que tem de invocar um outro método chamado `desconto` para completar sua operação. Neste momento, o interpretador Ruby, procura por este método na classe `CD`. Ao encontrá-lo, o método é executado e o seu retorno é enviado para o método `valor_com_desconto` que efetua a subtração com a variável `@valor` e retorna este resultado final.

O principal problema aqui é que, em Ruby, não existe o conceito de encapsulamento entre os objetos envolvidos no uso da herança. O mesmo problema acontece com variáveis de instância (que por padrão são sempre privadas), que podem ser alteradas nas superclasses e afetar algum outro comportamento das subclasses que usam as variáveis alteradas.

Reúso e customização podem ser armadilhas

Algumas classes ancestrais ou superclasses fornecem métodos que são projetados para serem substituídos pelas classes descendentes ou subclasses. Este comportamento, quando bem definido e explicado, pode ser muito útil, já que na superclasse podemos definir a maior parte dos comportamentos, deixando apenas alguma parte dele para ser customizada pelas subclasses. Este tipo de comportamento é tão comum que virou um Design Pattern chamado **Template Method**.

Porém, estas customizações podem nos trazer alguns problemas. O código escrito até o momento possui uma falha devido às customizações disponíveis em algumas APIs do Ruby. Vamos ver um exemplo de código que demonstra o problema:

```

class CD < Midia
  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end
end

windows = CD.new "Windows 95", 239.9, :sistemas_operacionais
p windows
# => #<CD:0x007ff86b91ec20 @desconto=0.1, @valor=239.9,
      @titulo="Windows 95",
      @categoria=:sistemas_operacionais>

puts windows # => #<CD:0x007ff86b91ec20>

```

Internamente, sabemos que o método `p` chama o método `inspect` do objeto `windows`, e o método `puts` por sua vez chama o método `to_s`. Ao executarmos esse código, o resultado exibido no terminal é o retorno da implementação destes métodos.

O método `inspect` herdado da classe `Object` fornece uma saída para depurarmos o conteúdo do objeto. Já o método `to_s`, também da classe `Object`, é um método que na maioria das vezes deve ser sobrescrito a fim de tornar-se mais útil quando o invocamos. Foi o que fizemos com os objetos `Livro` e `DVD`, agora vamos fazer o mesmo com os objetos do tipo `CD`:

```

class CD < Midia
  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Titulo: #{@titulo}, Valor: #{@valor} }
  end
end

```


end

```
windows = CD.new "Windows 95", 239.9, :sistemas_operacionais  
p windows # => Titulo: Windows 95, Valor: 239.9
```

Aparentemente, não existe nenhum problema com este código. Entretanto, o método `inspect` da classe `Object` funciona de uma maneira um pouco deselegante no Ruby 1.9, definindo sua própria implementação do método `to_s` e deixando alguns efeitos colaterais em nosso código:

```
windows = CD.new "Windows 95", 239.9, :sistemas_operacionais  
puts windows # => Titulo: Windows 95, Valor: 239.9
```

Temos exatamente a mesma saída do método `inspect`. A própria documentação do `inspect` fala sobre esta decisão em relação ao design no código. Resumidamente, quando determinado objeto não sobrescreve o método `to_s`, o método `inspect` possui seu retorno padrão que vimos anteriormente; caso contrário, ele apenas delega a chamada ao `to_s`.

A maneira de resolver este problema é sobrescrever o método `inspect` e definir sua própria implementação, que pode ser parecida com a original. Indiretamente, isso pode afetar a hierarquia de classes criadas.

Se por exemplo, a classe `Midia` sobrescrever o método `to_s`, todas as classes que a estendem herdarão o método `to_s`. Por consequência, se quisermos uma implementação do método `inspect` parecida com a original, vamos ter de criá-la nas subclasses.

Com esse conhecimento sobre a herança em Ruby, temos embasamento o suficiente para conhecermos um novo recurso, chamado **módulos**. Eles são a base para uma alternativa da

herança, conhecida como *mixings*.

7.4 MÓDULOS

Namespace

A classe `Biblioteca`, que criamos nos capítulos anteriores, armazena um `Array` de objetos do tipo `Livro`. Já que esta classe apenas armazena objetos, podemos batizá-la com um nome mais sugestivo, como chamá-la apenas de `Set`, ou seja, um conjunto, neste caso de livros:

```
class Set
  def initialize
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(livro)
    salva livro do
      livros << livro
    end
  end

  def livros_por_categoria(categoria)
    livros.select { |livro| livro.categoria == categoria }
  end

  def livros
    @livros ||= @banco_de_arquivos.carrega
  end

  private

  def salva(livro)
    @banco_de_arquivos.salva livro
    yield
  end
end
```

```
conjunto_de_livros = Set.new
conjunto_de_livros.adiciona Livro.new "TDD", "Mauricio Aniche",
                                     "123454", 247, :testes
```

O que fizemos nesse código é muito perigoso, porque a `corelib` do Ruby já possui uma classe chamada `Set`, sendo assim, o código abre a classe e altera completamente os comportamentos definidos pelos engenheiros que a projetaram. Se quisermos mesmo que a classe que guarda os objetos `Livro` tenha o nome `Set`, o único jeito é adicionar um *namespace* para diferenciar a classe `Set` do Ruby com a classe `Set` do sistema de venda de livros.

É importante ressaltar que as bibliotecas que você utilizará quando programar em Ruby podem possuir classes cujos nomes conflitam com as próprias classes já existentes no código-fonte que escrevemos. Os módulos permitem que classes e outros módulos possuam o mesmo nome, evitando colisões. Podemos pensar nos *namespaces* como diretórios no projeto em que separamos os arquivos que representam cada classe ou módulo.

Algo muito comum acontece em código Java, que utilizam os pacotes para distribuir e organizar código. Além de evitar possíveis conflitos com nomes de classes já existentes nas bibliotecas existentes.

Vamos então criar um módulo cuja função é criar um namespace, que neste caso pode ser algo que identifique que a classe pertence ao sistema de vendas. Vamos batizar o sistema como `VendaFacil`:

```
module VendaFacil
end
```

Módulos em Ruby seguem as mesmas regras de definição de nomes de classes, a diferença é o uso da palavra `module` .

Agora basta colocar a classe `Set` no *namespace* do módulo `VendaFacil` :

```
module VendaFacil
  class Set
    def initialize
      @banco_de_arquivos = BancoDeArquivos.new
    end

    # Outros métodos ...
  end
end
```

Usamos o operador *Constant lookup* `<::` para acessarmos uma classe ou constante que foi definida dentro de namespace:

```
# acessando classe Set do namespace VendaFacil
conjunto_de_livros = VendaFacil::Set.new

conjunto_de_livros.adiciona Livro.new "TDD", "Mauricio Aniche",
  "123454", 247, :testes
```

Você pode utilizar o *Constant lookup* para encontrar classes ou até mesmo outros módulos no nível que desejar. Se a classe ou módulo que você deseja acessar estiver em um namespace `VendaFacil`, `Util` ou `Set`, podemos acessá-lo utilizando:

```
VendaFacil::Util::Set
```

Utilizar módulos como namespaces é a forma padrão de organizar bibliotecas e classes em Ruby. Esta é uma excelente prática que recomendo que seja seguida conforme seu código for crescendo e as possibilidades de conflitos aumentarem.

Mixins

Precisamos formatar o atributo `valor` dos objetos `Livro`, `CD` e `DVD` respeitando as regras de formatação do Real. Podemos definir o método formatador na classe `Midia` que é a superclasse de todas as classes citadas anteriormente:

```
class Midia
  def valor_formatado
    "R$ #{@valor}"
  end
end
```

O problema de criar o método `valor_formatado` dentro da classe `Midia` é que, se precisarmos utilizar este comportamento em classes que não fazem parte da hierarquia, seria necessário duplicar o código. Isso seria extremamente ruim, pela necessidade de manter o mesmo código em várias partes do sistema.

Os módulos tem outra utilidade, muito importante e bastante usada nas principais bibliotecas e frameworks existentes, como por exemplo, o Rails. Os *mixings* eliminam a necessidade de herança para compartilhar código, além de permitir uma espécie de herança múltipla, já que podemos usar vários módulos em uma mesma classe.

Módulos podem definir métodos de classe e de instância. Então isso quer dizer que podemos instanciar módulos? Não. Módulos não podem ser instanciados, suas únicas utilidades são fornecer *namespaces* e compartilhar código. O que fazemos é um `include` do módulo criado em uma ou várias classes, e assim as classes que incluíram aquele módulo passam a ter em sua interface todos os **métodos de instância** definidos no módulo incluído. Nós misturamos os métodos na classe que incluiu o módulo, daí vem o nome *mixings*.

Podemos criar o método `valor_formatado` dentro de um módulo e utilizá-lo em vários lugares do sistema:

```
# lib/formatador_moeda.rb
module FormatadorMoeda
  def valor_formatado
    "R$ #{@valor}"
  end
end
```

Repare que, no método `valor_formatado` que foi criado, nos referimos a uma variável de instância chamada `@valor`. Lembre-se de que os métodos criados dentro de módulos, assim como na herança, assumem o escopo onde foram invocados. Por exemplo:

```
# coding: utf-8
class Livro < Midia

  # todos os métodos de instância são
  # incluídos nos objetos Livro

  include FormatadorMoeda

  def initialize(titulo, autor, isbn = "1", numero_de_paginas,
                valor, categoria)

    @titulo = titulo
    @autor = autor
    @isbn = isbn
    @numero_de_paginas = numero_de_paginas
    @categoria = categoria
    @valor = valor
    @desconto = 0.15
  end

  def to_s
    "Autor: #{@autor}, Isbn: #{@isbn},
      Páginas: #{@numero_de_paginas},
      Categoria: #{@categoria}"
  end
end

tdd = Livro.new "TDD", "Mauricio Aniche", "123454", 247,
```

```
:testes  
tdd.valor_formatado # => R$ 247
```

O objeto `tdd` possui o método `valor_formatado` definido no módulo que incluímos na classe `Livro` e, ao invocarmos, a variável `@valor` usada é a definida no objeto `Livro` que invocamos o método.

RESOLVENDO AMBIGUIDADE DE MÉTODOS

Uma das dúvidas mais comuns para quem está começando a utilizar *mixings* é como é feita a procura do método que precisa ser executado. Ou seja, se definirmos um método `valor_formatado` na classe `Livro` ou dentro de algum outro *mixing* feito na classe `Livro`, qual método será executado?

O interpretador Ruby primeiro procura o método dentro da classe do objeto, depois nos *mixings* incluídos na classe, e só então na superclasse e seus *mixings*. Se a classe possuir múltiplos módulos incluídos, no último módulo incluído será feita a primeira busca, depois no penúltimo e assim por diante.

É válido ressaltar que, quando fazemos o `include` do módulo `FormatadorMoeda`, fazemos apenas uma referência que a classe `Livro` tem os comportamentos definidos no módulo. O interpretador Ruby não copia os métodos de instância para dentro da classe, em vez disso, da classe `Livro` para o módulo `FormatadorMoeda`. Isso implica que se o módulo for incluso em

várias classes, ao alterarmos algum comportamento definido nele, todas as classes que incluem o módulo terão seus comportamentos alterados também.

Variáveis de instância e mixings

Módulos não possuem variáveis de instância, pois não podem ser instanciados. Precisamos nos lembrar de como as variáveis de instância funcionam em Ruby: a primeira menção para uma variável precedida com `@` cria um variável de instância no objeto atual.

Para os *mixings*, isso significa que o módulo que você inclui em uma classe pode definir variáveis de instância no objeto da classe que está sendo instanciado. Entretanto, este comportamento pode trazer erros inesperados, já que as variáveis de instância definidas no módulo podem conflitar com variáveis de instância definidas na classe que o incluiu, ou pode até mesmo conflitar variáveis definidas em outros módulos que foram incluídos na classe.

A dica para evitar este tipo de problema é utilizar algum tipo de prefixo nos nomes das variáveis a fim de criar um nome único. Mas este é um problema muitas vezes inevitável quando utilizamos *mixings*, já que podemos incluir quantos módulo quisermos. Portanto, tenha cuidado e veja se realmente faz sentido você incluir um módulo só para não repetir código.

Incluindo módulos em objetos existentes

Se você precisa incluir comportamentos existentes em um módulo apenas em alguns objetos, pode fazer isso utilizando a palavra chave `extend` em vez de `include`. Por exemplo, dado

que temos dois objetos do tipo `DVD` criados, e desejamos que apenas um deles possua os comportamentos definidos no módulo `FormatadorMoeda` :

```
windows = DVD.new "Windows 7 for Dummies", 98.9,
             :sistemas_operacionais
linux = DVD.new "Linux for Dummies", 13.9,
          :sistemas_operacionais

windows.extend FormatadorMoeda
windows.valor_formatado # => R$ 98.9

linux.valor_formatado
# => NoMethodError: undefined method 'valor_formatado'
for "linux":DVD
```

O método `extend` inclui os métodos de instância do módulo apenas para o objeto `windows` ; o outro objeto `linux` , apesar de ser da mesma classe, não possui o método.

Utilizando módulos da biblioteca padrão do Ruby

Existe uma grande variedade de módulos na biblioteca padrão do Ruby. Alguns deles, muito poderosos, nos permitem escrever menos código e executar tarefas complexas usando apenas um `include` . O principal motivo é que eles interagem com código da classe que os inclui.

As APIs de coleção do Ruby (por exemplo, `Array` , `Set` e `Hash`) possuem uma variedade de método como: `sort` , `include?` , `select` etc. Se olharmos o código da classe `Biblioteca` , podemos visualizar o quão legal seria termos todos estes métodos e podemos invocá-los diretamente em uma instância de `Biblioteca` em vez de retornar um `Array` através do método `livros` , e daí então fazer as operações desejadas.

Todos estes métodos podem ser incluídos na classe Biblioteca , incluindo o módulo Enumerable . A única coisa necessária será criar um método each que itera e retorna todos os elementos do Array de Livro .

```
class Biblioteca

  include Enumerable

  def initialize
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(livro)
    salva livro do
      livros << livro
    end
  end

  def livros_por_categoria(categoria)
    livros.select { |livro| livro.categoria == categoria }
  end

  def livros
    @livros ||= @banco_de_arquivos.carrega
  end

  # método each que possibilita que os outros métodos
  # do módulo Enumerable funcionem em uma
  # instância de Biblioteca
  def each
    livros.each { |livro| yield livro }
  end

  private

  def salva(livro)
    @banco_de_arquivos.salva livro
    yield
  end
end
```

Agora, se precisarmos saber a somatória total de todos os livros existentes dentro da `Biblioteca`, basta usarmos o método `inject` que itera sobre os elementos de um `Array` e guarda a somatória em uma variável:

```
biblioteca = Biblioteca.new

biblioteca.adiciona Livro.new "TDD", "Mauricio Aniche",
                             "123454", 247, 69.9, :testes
biblioteca.adiciona Livro.new "Web Design Responsivo",
                             "Tárcio Zemel", "45256", 240, 69.9, :web

p biblioteca.inject(0) { |tot, livro| tot += livro.valor }
# => 139.8
```

A classe `Biblioteca` se comporta como um `Enumerable`, e possui todos os métodos que um `Array`, por exemplo. Mas neste caso, todos os métodos do módulo `Enumerable` utilizam o método `each` que itera pela variável `@livros`.

7.5 INDO MAIS A FUNDO: CONSTANT LOOKUP DE DENTRO PARA FORA

É muito importante entendermos que as constantes (classes, módulos etc.) são procuradas do namespace mais interno, partindo para o mais externo, até chegar ao namespace global. Isso é muito confuso às vezes, e por sorte, aparecerá no momento da execução do código.

Por exemplo, vamos alterar a classe `VendaFacil<::Set` para inicializar a variável `@livros` com um `Array` vazio utilizando outra sintaxe:

```
module VendaFacil
  class Set
```

```

    def initialize
      @livros = Array.new
      @banco_de_arquivos = BancoDeArquivos.new
    end

    # Outros métodos ...
  end
end

```

Se executarmos este código no `irb` e imprimirmos de qual tipo é a variável `@livros`, veremos que continua sendo do tipo `Array`:

```

conjunto_de_livros = VendaFacil::Set.new
p conjunto_de_livros.livros.class # => Array

```

Este comportamento parece óbvio, porém, se existe um pequeno pedaço de código como o seguinte, as coisas podem ser alteradas de forma inesperada:

```

module VendaFacil
  module Array
    # alguns métodos aqui
  end
end

```

O desenvolvedor que escreveu o código anterior se preocupou com o fato de já existir uma constante `Array` definida, e organizou o código adicionando o namespace `VendaFacil` na criação desta nova constante. Porém, esta mudança nos causa pequenas dores de cabeça. Ao tentar chamar o método `new` na classe `Array` dentro da classe `VendaFacil<::Set`, um erro será lançado:

```

VendaFacil::Set.new
# => NoMethodError: undefined method 'new' for
VendaFacil::Array:Module

```

Existem diversas maneiras de solucionarmos este problema.

Uma delas é voltar a inicialização do Array para o código anterior `[]` . A outra é alterar o nome do módulo `VendaFacil<::Array` para algum nome que não conflite com classes da *core standard library* do Ruby, o que na maioria das vezes é a melhor solução. Se nenhuma das opções anteriores for possível, podemos utilizar explicitamente o *constant lookup* do namespace global:

```
module VendaFacil
  class Set
    def initialize
      @livros = ::Array.new
      @banco_de_arquivos = BancoDeArquivos.new
    end

    # Outros métodos ...
  end
end
```

Adicionar `<::` antes da chamada ao `Array.new` forçará o interpretador Ruby a ignorar os namespaces internos e externos, e procurar diretamente no namespace global, onde estão as constantes que não possuem namespace.

No geral, quando utilizamos *lookup absolute* de constantes, é um sinal que deveríamos usar outros nomes para as classes que conflitam.

7.6 DUCK TYPING: O POLIMORFISMO APLICADO NO RUBY

Sistemas de tipos é uma parte fundamental em todas as linguagens de programação, e a maneira que cada linguagem o implementa influencia o design das classes do sistema.

Linguagens estaticamente tipadas como C++ e Java fazem com

que pensemos em objetos como abstração de estrutura de dados e comportamentos. E não existe de fato uma grande diferença entre os objetos criados e o seu tipo, eles são intimamente ligados.

Porém, não são todas as linguagens que se comportam deste maneira. Ruby, por exemplo, é uma linguagem que possui tipagem dinâmica, que permite que as classes sejam modeladas e lidem com um estilo de tipo conhecido como *Duck Typing*.

Duck Typing considera o que um objeto pode fazer e não de qual tipo ele é, quebrando a ligação com a classe do objeto criado. Agora quando escrevemos um método, ou iteramos um `Array`, temos de pensar no que o objeto pode fazer, ignorando seu tipo.

Vamos tomar como exemplo a classe `Biblioteca`, que atualmente guarda vários objetos do tipo `Livro`. Devemos lidar com objetos que sejam mídias, ou seja, `Livro`, `DVD` e `CD`. Mas será que nosso código atualmente não suporta esses outros "tipos"? Vamos ver:

```
biblioteca = Biblioteca.new

windows = DVD.new "Windows 7 for Dummies", 98.9,
               :sistemas_operacionais

biblioteca.adiciona windows

biblioteca.each do |midia|
  p midia.titulo # => Windows 7 for Dummies
end
```

Repare que não houve necessidade de alterarmos sequer uma linha de código no método `adiciona` e também no método `each`. Isso acontece primeiro porque o interpretador Ruby não se importa com o tipo de objeto passado como argumento; segundo

porque os métodos não esperam nenhum comportamento específico dos objetos que estão sendo incluídos. Isso significa que podemos passar **qualquer** tipo de objeto na chamada do método:

```
class Revista
  attr_reader :titulo

  def initialize(titulo)
    @titulo = titulo
  end
end

biblioteca = Biblioteca.new

mundo_j = Revista.new "MundoJ"

biblioteca.adiciona mundo_j

biblioteca.each do |qualquer_objeto|
  p qualquer_objeto.titulo # => MundoJ
end
```

Porém, para o método `livros_por_categoria`, que está com um nome defasado, o que importa não é o tipo, mas o que os objetos guardados na variável `@livros` podem fazer.

```
biblioteca = Biblioteca.new

mundo_j = Revista.new "MundoJ"

biblioteca.adiciona mundo_j

biblioteca.livros_por_categoria :testes
# => NoMethodError: undefined method 'categoria' for
      "mundo_j":Revista
```

O método `livros_por_categoria` espera que os objetos do Array `@livros` tenham um método `categoria`, porém, o recém-criado objeto `Revista` possui apenas um método `modelo`. Ou seja, é um objeto que definitivamente não deveria

estar dentro da variável `@livros` , pois não possui os comportamentos necessários.

Verifique manualmente os tipos dos objetos

Caso precise executar comandos apenas para determinados tipos, é possível verificar o tipo de um objeto de alguns maneiras. A primeira forma foge um pouco do estilo de tipagem imposto pelo *Duck Typing*, porque faz uma verificação explícita pela classe do objeto. Vamos aproveitar para renomear as variáveis e métodos ligados ao nome **livro**:

```
class Biblioteca

  include Enumerable

  def initialize
    @banco_de_arquivos = BancoDeArquivos.new
  end

  def adiciona(midia)
    salva midia do
      midias << midia
    end if midia.kind_of? Midia
  end

  def midias_por_categoria(categoria)
    midias.select { |midia| midia.categoria == categoria }
  end

  def midias
    @midias ||= @banco_de_arquivos.carrega
  end

  def each
    midias.each { |midia| yield midia }
  end

  private
```



```

def salva(midia)
  @banco_de_arquivos.salva midia
  yield
end
end

```

O método `kind_of?` retorna `true` se o objeto for um tipo ou subtipo da constante passado como argumento, em nosso exemplo, da constante `Midia`. Com a verificação incluída no método `adiciona`, são aceitos apenas objetos do tipo `Midia`, `Livro`, `DVD` e `CD`.

```

biblioteca = Biblioteca.new

windows = DVD.new "Windows 7 for Dummies", 98.9,
               :sistemas_operacionais

biblioteca.adiciona windows

mundo_j = Revista.new "MundoJ"

biblioteca.adiciona mundo_j # não inclui o objeto mundo_j

biblioteca.each do |midia|
  p midia.titulo # => Windows 7 for Dummies
end

```

Caso desejemos incluir objetos de outros tipos (por exemplo, `Revista`) que não são subtipos de `Midia`, devemos alterar o método `adiciona` e fazer a verificação por estes outros tipos. Isto é, perdemos todo o poder do *Duck Typing* que é justamente evitar este tipo de verificação e importar-se apenas com o que o objeto pode fazer.

Podemos permitir que sejam incluídos qualquer tipo de objeto e fazer a verificação apenas pelos métodos:

```

class Biblioteca

```

```

include Enumerable

def initialize
  @banco_de_arquivos = BancoDeArquivos.new
end

def adiciona(midia)
  salva midia do
    midias << midia
  end
end

def midias_por_categoria(categoria)
  midias.select do |midia|
    midia.categoria == categoria if
      midia.respond_to? :categoria
    end
  end
end

def midias
  @midias ||= @banco_de_arquivos.carrega
end

def each
  midias.each { |midia| yield midia }
end

private

def salva(midia)
  @banco_de_arquivos.salva midia
  yield
end
end

```

Com esta mudança, ao executarmos o método `adiciona`, podemos passar qualquer objeto como parâmetro. Porém, se executarmos o método `midias_por_categoria`, apenas os objetos que possuem (`respond_to?`) o método `categoria` são considerados na comparação, independente do seu tipo.

A vantagem do uso do *Duck Typing* pode ser constatada ao

criar testes de unidade para a classe `Biblioteca` , porque poderíamos passar *mocks*, que possuem uma implementação *fake* do método `categoria` , que permitiria que os comportamentos fossem testados isoladamente.

7.7 HERANÇA OU MIXING? QUAL DEVO USAR?

Módulos e herança no geral servem para compartilharmos código e evitarmos assim a duplicidade de partes do sistema. Mas a pergunta principal é: qual delas devo escolher?

Esta é uma das típicas questões, como muitas outras, que a resposta **deve** ser: depende. Com a experiência que ganhamos ao longo do anos como desenvolvedor, tendemos a escolher uma das alternativas. Mas o importante é sempre lembrar de que não existe *bala de prata*, e não se esquecer quais são os pontos fracos e fortes de cada uma das abordagens.

Classes em Ruby estão relacionados a ideia de tipos. Quando criamos um objeto "texto", dizemos que ele é do tipo `String` ; ou quando criamos um objeto `[1, 3]` , dizemos que ele é do tipo `Array` . Quando criamos classes, estamos definindo novos tipos, e quando usamos herança, criamos um subtipo de algum outro tipo já existente. Se temos um objeto `carro` , podemos dizer que ele é *um* `Veículo` , ou seja, tudo que um `Veículo` pode fazer, o `carro` faz.

Portanto, quando criamos subtipos, estamos definindo relacionamentos *é um*. Porém, nos domínios da maioria das aplicações, não existem este tipo de relacionamento. O mais

comum é dizer que um objeto *tem um* ou *usa um* outro objeto para executar seus comportamentos, ou seja, na maioria das vezes utilizamos composição, e não herança.

Além disso, herança nos traz graves problemas de acoplamento. Qualquer alteração na superclasse pode quebrar todas as subclasses, que muitas vezes não estão sob nosso controle. Comportamentos definidos na superclasse podem vazar para quem utiliza as subclasses, e espalhar estas regras por todos os lados na aplicação, o que inibe ainda mais as mudanças e evolução do sistema.

Mixings possuem basicamente os mesmos problemas que herança. Ao incluirmos um módulo em uma classe, ela passa a ter todos os comportamentos definidos no módulo. E se quisermos alterar algum destes métodos, vamos quebrar todos os códigos que invocam os métodos através de instâncias de classes que mixaram os comportamentos deste módulo.

Porém, eu vejo algumas vantagens de utilizar *mixings* e não herança:

- Módulos podem ser gerenciados em tempo de execução através do método `include`, enquanto herança é definida no momento da escrita da classe;
- Módulos são mais fáceis de testar unitariamente de maneira isolada;
- Módulos nos permite utilizar metaprogramação (capítulo *Metaprogramação e seus segredos*) para definir *Domain Specific Languages*;
- Módulos são classes que não podem ser instanciadas, eles existem apenas para adicionar funcionalidades para classes

já existentes.

Hoje eu usaria herança apenas em casos esporádicos nos quais o relacionamento *é um* realmente for comprovado. No caso de querer apenas compartilhar comportamentos entre várias classes, eu usaria *mixings*. Porém, é preciso lembrar dos problemas que ambas provocam no design da aplicação, trazendo grande acoplamento entre os componentes.

A melhor solução é usar composição para reutilizar acoplamento. Mas a comunidade Ruby não utiliza muito esta prática, exceto alguns casos raros, já que a maioria dos códigos usam *mixings*.

METAPROGRAMAÇÃO E SEUS SEGREDOS

A metaprogramação permite que você não fique limitado às abstrações que a linguagem lhe oferece, permite que você vá além e crie novas abstrações sobre a linguagem Ruby. Isso permite basicamente que você crie *Domain Specific Languages* (DSL), a fim de facilitar a leitura e escrita do código da sua aplicação.

A grande maioria dos bons programadores Ruby utilizam técnicas de metaprogramação para criar essa simplicidade de código. Porém, esta é uma técnica que leva um bom tempo para ser entendida, pois é necessário conhecimento da linguagem Ruby como um todo. Neste capítulo, vou mostrar como usar metaprogramação a favor do seu código, utilizando as suas principais técnicas.

Metaprogramação é um conteúdo muito extenso que exigiria um único livro só para contar todos os seus detalhes. Com o conteúdo que você verá a partir de agora, será capaz de criar suas próprias DSLs e poupar várias linhas de código.

8.1 ENTENDA O SELF E METHOD CALLING

Ruby possui um conceito de *current object*, representado pela variável `self`. Esta variável possui dois papéis importantes em todo código Ruby, inclusive na metaprogramação.

Variáveis de instância

O `self` é responsável por armazenar as variáveis de instância de um objeto. Quando tentamos acessar uma variável de instância, o Ruby vai procurar por ela dentro do objeto `self`:

```
class Revista
  def initialize(titulo)
    @titulo = titulo
  end

  def titulo
    @titulo
  end
end

mundo_j = Revista.new "MundoJ"
p mundo_j.titulo # => MundoJ
```

A objeto referenciado por `mundo_j` possui uma variável de instância `@titulo`, que está associado ao objeto `self`. Quando invocamos o método `titulo`, a variável acessada está dentro do `self`.

Method calling

A variável `self` possui um papel importante nas chamadas de método que fazemos nos objetos. Quando invocamos um método em algum objeto, dizemos que estamos enviando uma mensagem para que aquele objeto execute alguma ação. Chamamos esse objeto de "*receiver* da chamada" (receptor). Quando executamos o código `mundo_j.titulo`, o objeto `mundo_j` é o *receiver* e

`titulo` é a ação que queremos que seja executada.

Quando executamos a chamada a algum método sem explicitar qual é o *receiver*, o interpretador Ruby assume que o *receiver* é o *current object* (`self`). Por exemplo:

```
# coding: utf-8
class Revista
  def initialize(titulo)
    @titulo = titulo
  end

  def titulo
    @titulo
  end

  def titulo_formatado
    "Título: #{titulo}"
  end
end

mundo_j = Revista.new "MundoJ"
p mundo_j.titulo_formatado # => Título: MundoJ
```

Nesse código, na chamada ao método `titulo`, o *receiver* é `self`. Mas quem é o `self` neste momento?

```
# coding: utf-8
class Revista
  def initialize(titulo)
    @titulo = titulo
  end

  def titulo
    @titulo
  end

  def titulo_formatado
    puts self # => #<Revista:0x007f93aa18cd70>
    "Título: #{titulo}"
  end
end
```



```

mundo_j = Revista.new "MundoJ"
p mundo_j.titulo_formatado # => Título: MundoJ

```

O código `puts self` imprimiu que o `self` é o próprio objeto `mundo_j`, ou seja, o método `titulo` foi invocado no próprio objeto `mundo_j`.

Podemos invocar o método `titulo`, especificando qual o receiver: `self.titulo`. Porém, isso só deixaria nosso código mais poluído, porque o `self` neste caso não faria diferença, já que implicitamente o método será invocado em `self`.

Quando o *receiver* é explícito, o comportamento é bastante parecido, sendo a única e importante mudança que o `self` é alterado durante a chamada do método para representar o *receiver* explícito. Quando o Ruby termina a execução do método, o `self` volta a ter seu estado anterior. Por exemplo:

```

# coding: utf-8
class Revista
  def initialize(titulo)
    @titulo = titulo
  end

  def titulo
    titulo_upcase = @titulo.upcase
    "Título: #{titulo_upcase}"
  end
end

mundo_j = Revista.new "MundoJ"
p mundo_j.titulo # => Título: MUNDOJ

```

Quando invocamos o método `upcase` em uma `String`, então o `self` neste momento será o objeto `String`. Logo após a execução do `upcase`, o `self` volta a ser o objeto `mundo_j`.

8.2 O IMPACTO DO SELF NA DEFINIÇÃO DE CLASSES

Quando definimos classes em Ruby, `self` também é alterado para representar o objeto que guarda as informações da classe que estamos definindo. Isso acontece porque definir classes em Ruby é como executar um outro código qualquer, onde criamos objetos e invocamos métodos. Para mostrar como isso funciona, vamos colocar um `puts` no meio da declaração da classe `Revista` :

```
# coding: utf-8
class Revista
  puts "O self aqui é: #{self}"
  puts "O self aqui é do tipo: #{self.class}"

  def initialize(titulo)
    @titulo = titulo
  end

  def titulo
    titulo_upcase = @titulo.upcase
    "Título: #{titulo_upcase}"
  end
end
```

Quando executamos este código, o resultado é:

```
# => O self aqui é: Revista
# => O self aqui é do tipo: Class
```

Na definição de classes, o `self` é um objeto do tipo `Class` que guarda todas as informações da classe que estamos criando. Ele pode, inclusive, guardar variáveis de instância:

```
# coding: utf-8
class Revista

  @id = 0
```

```

def self.id
  @id += 1
end

def initialize(titulo)
  @titulo = titulo
end

def titulo
  titulo_upcase = @titulo.upcase
  "Título: #{titulo_upcase}"
end
end

```

Há muita coisa diferente nesse código que ainda não vimos, então, vamos por partes e com bastante calma.

Definimos uma variável `@id` com o valor `0` dentro do objeto `self` da definição da classe `Revista`, e já sabemos que `self` neste momento é o objeto do tipo `Class` que guarda não só os métodos definidos na classe, mas também variáveis de instância que não são dos objetos `Revista`, mas sim do objeto `Class` que guarda informações da classe `Revista`. Lembre-se de que em Ruby **tudo** é objeto.

Logo após, definimos um método `id` dentro de `self`, ou seja, um método dentro do objeto `Class`. Este método **deve** ser executado a partir da objeto `Class` que representa a classe `Revista`:

```

p Revista.id # => 1
p Revista.id # => 2
p Revista.id # => 3

p Revista.new('MundoJ').id
# => NoMethodError: undefined method 'id'
for #<Revista:0x007fcb7c86dd60>

```

Repare que o método não existe para instâncias do tipo

Revista , mas apenas na classe `Revista` . Complicado essa parte, não? Vamos entendê-la com mais detalhes na próxima seção, estudando os *Singletons*.

8.3 SINGLETON CLASS E A ORDEM DA BUSCA DE MÉTODOS

Em Ruby, quando invocamos um determinado método em um objeto, o interpretador procura pelo método no objeto `class` que representa a classe do objeto onde o método foi invocado, por exemplo:

```
munido_j = Revista.new "MundoJ"
p munido_j.titulo # => Título: MUNDOJ
```

O objeto do tipo `Class` que representa a classe `Revista` guarda referência para o método `titulo` e, quando invocamos este método, o interpretador Ruby procura por ele no objeto `Class` . Tudo isso já sabíamos, mas a novidade agora é que o interpretador Ruby procura por este método em outro lugar primeiro:

```
windows = DVD.new "Windows 7 for Dummies", 98.9,
               :sistemas_operacionais
linux = DVD.new "Linux for Dummies", 13.9,
        :sistemas_operacionais

def windows.desconto_formatado
  "Desconto: #{@desconto * 100}%"
end

p windows.desconto_formatado
# => Desconto: 10.0%

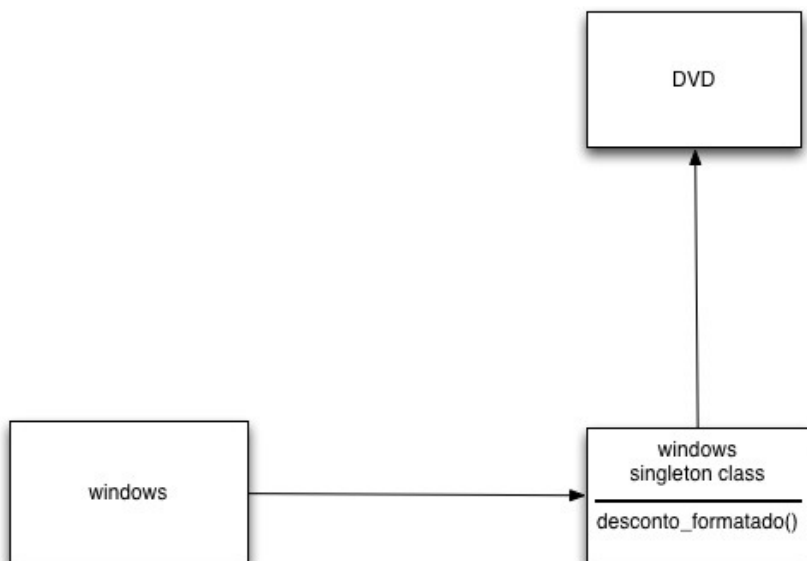
p linux.desconto_formatado
# => NoMethodError: undefined method 'desconto_formatado'
for "linux":DVD
```

Quando invocamos o método `desconto_formatado` no objeto `windows`, o Ruby define `self` como o objeto `windows`, e então procura pelo método e o encontra. Mas porque quando invocamos o mesmo método no objeto `linux`, ele não é encontrado? O método não foi definido para todos os objetos DVD ?

O Ruby faz alguma *magia negra* para definir o método `desconto_formatado` apenas para o objeto `windows` ? A resposta é **não**. Lembre-se de que metaprogramação não é *magia negra*, consiste apenas em aprender a linguagem a fundo e saber usá-la a seu favor.

Quando definimos o método `desconto_formatado` no objeto `windows`, o Ruby cria uma nova classe e define o método dentro dela. Esta classe é anônima e é conhecida como *singleton class*, *metaclass* ou *eigenclass*. Vamos seguir chamando-a de *singleton class* até o final do livro.

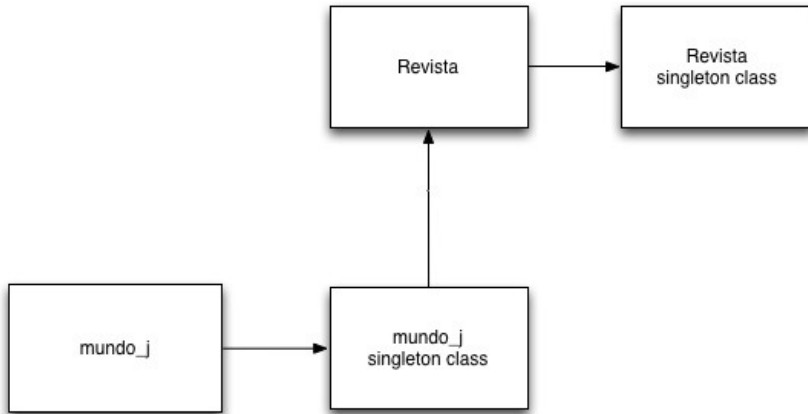
A *singleton class* criada é definida como a `class` do objeto `windows`, e a `superclass` da *singleton class* é a classe `DVD`, conforme pode ser observado na figura:



Ao invocarmos algum outro método que não está definido na *singleton class* do objeto `windows`, o interpretador Ruby primeiro vai procurar pelo método na *singleton class*; caso não encontre, procurará na *superclass* `Revista`, e provavelmente o encontrará.

Quando definimos o método `id` na classe `Revista`, nós criamos o método na *singleton class* do objeto `Class` que representa as definições da classe `Revista`. Quando invocamos o método `Revista.id`, na verdade estamos invocando o método na *singleton class*.

Após a definição, foi criada uma *singleton class* cuja *superclass* é `Revista`, conforme a figura a seguir mostra:



Podemos agora invocar o método `id` no momento da criação de objetos do tipo `Revista` :

```
# coding: utf-8
class Revista

  @id = 0

  def self.id
    @id += 1
  end

  def initialize(titulo)
    @id = self.class.id
    @titulo = titulo
  end

  def id
    @id
  end

  def titulo
    titulo_upcase = @titulo.upcase
    "Titulo: #{titulo_upcase}"
  end
end
```

```

mundo_j = Revista.new "MundoJ"
p mundo_j.id # => 1

mundo_ruby = Revista.new "MundoRuby"
p mundo_ruby.id # => 2

```

Primeira definição que devemos nos lembrar é que a variável de instância `@id`, definida no método `initialize`, não é a mesma que foi definida na criação da classe. Elas estão em contextos (`self`) diferentes: a primeira foi definida no contexto do objeto `Class` que representa as definições da classe `Revista`, já a segunda pertence ao objeto `Revista` que está sendo criado, portanto elas não conflitam. Exatamente por esse motivo, é impossível acessar a variável `@id` da classe `Revista` em métodos de instâncias de `Revista`, por exemplo, dentro do método `initialize`.

A mesma regra vale para os métodos `id`. O primeiro foi criado na *singleton class* do objeto `Class` que representa a classe `Revista`, o segundo foi definido para as instâncias de `Revista` que forem criadas.

O valor da variável `@id` do método `initialize` é inicializado com o valor de retorno do método `id` definido na *singleton class* da classe `Revista`. Como o `self` no método `initialize` representa o objeto que está sendo criado, para acessar o método `id` da *singleton class* da classe `Revista`, invocamos `self.class` e em seguida o método `id`, que incrementa e retorna a variável `@id`.

8.4 INDO MAIS A FUNDO: ACESSANDO A SINGLETON CLASS

Nós aprendemos a definir métodos na *singleton class* de objetos do tipo `DVD`, mas também aprendemos a adicioná-los na *singleton class* do objeto `Class` que representa a classe `Revista`. Existe outra sintaxe para fazermos as mesmas tarefas feitas anteriormente:

```
windows = DVD.new "Windows 7 for Dummies", 98.9,
               :sistemas_operacionais

class << windows
  def desconto_formatado
    "Desconto: #{@desconto * 100}%"
  end
end

p windows.desconto_formatado
# => Desconto: 10.0%
```

Utilizando esta sintaxe, o objeto `self` passa a ser o objeto `windows`, fazendo com que o método `desconto_formatado` seja adicionado apenas neste objeto.

Podemos usar a mesma sintaxe ao definir métodos no contexto de um classe, por exemplo:

```
class Revista
  @id = 0

  class << self
    def id
      @id += 1
    end
  end
end

p Revista.id # => 1
p Revista.id # => 2
```

Neste caso, o `self` dentro da *singleton class* é o próprio objeto do tipo `Class` que representa a classe `Revista`. Uma das

vantagens de utilizar esta sintaxe é poder invocar métodos privados, já que eles não podem ser acessados através de um *receiver* explícito.

Imagine que precisamos adicionar um método para retornar o valor atual da variável `@id`. Podemos implementar isso usando o método `attr_reader`:

```
class Revista
  @id = 0

  class << self
    attr_reader :id
  end
end
```

Se utilizássemos a sintaxe antiga, faríamos: `self.attr_reader :id`. Porém, o interpretador acusaria um erro, informando que métodos privados não podem ser invocados a partir de um `self` explícito.

Eu, particularmente, prefiro a sintaxe `self.nome_do_metodo`. Utilizo a sintaxe `class << objeto` apenas em casos esporádicos como o exemplo que eu citei anteriormente.

8.5 METAPROGRAMAÇÃO E AS DEFINIÇÕES DE UMA CLASSE

Podemos utilizar o método `attr_reader` para criar pares de métodos `get` e `set` que acessam as variáveis de instância dos

objetos criados a partir da classe DVD , por exemplo:

```
# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end
```

Ao utilizarmos bibliotecas externas, nos deparamos com códigos que fazem chamadas a métodos no momento da definição da classe. Por exemplo, a biblioteca Mongoid, que cria uma camada ORM para acessarmos dados em um MongoDB, possui um método `field` que serve para especificarmos quais campos serão mapeados entre o objeto e um documento salvo no banco:

```
class Document
  include Mongoid::Document
  field :name, type: String
end
```

Estes métodos são sempre invocados no momento da definição da classe, e na maioria das vezes geram código por baixo dos panos. Nós também podemos criar métodos que serão utilizados na definição de classes, e que geram código automaticamente. Este é um dos poderes da metaprogramação, escrever código automaticamente sem que seja necessária muita repetição.

No capítulo anterior, nos criamos um `Module` que serve para

formatar uma variável `@valor` da classe onde o módulo foi incluído. Em nossos exemplos, usamos o módulo na classe `Livro`, mas também podemos utilizá-lo na classe `DVD`:

```
# coding: utf-8
class DVD < Midia
  include FormatadorMoeda

  attr_reader :titulo

  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end

  def to_s
    %Q{ Título: #{@titulo}, Valor: #{@valor} }
  end
end

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_formatado # => R$ 98.9
```

Agora precisamos que a variável `@desconto` também seja formatada. Podemos alterar o módulo `FormatadorMoeda` para resolver o problema:

```
module FormatadorMoeda
  def valor_formatado
    "R$ #{@valor}"
  end

  def valor_com_desconto_formatado
    "R$ #{valor_com_desconto}"
  end
end
```

Pronto! Temos também um método

`valor_com_desconto_formatado` (que usou o método `valor_com_desconto` definido na classe `Midia`):

```
windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_formatado # => R$ 98.9
p windows.valor_com_desconto_formatado # => R$ 89.01
```

Mas temos um problema um pouco grave ao adotar a solução proposta. A cada "valor" de atributo ou método dos meus objetos que eu queira formatar adicionado o prefixo "R\$", teremos de adicioná-lo no módulo `FormatadorMoeda`. Solução não muito elegante, concorda? Certo, você concordou. :)

O que podemos utilizar é metaprogramação. Vamos criar um método `formata_moeda` que servirá para definirmos quais variáveis de instância ou métodos devem ter um método similar que retorne o valor formatado. Por exemplo, se houver um método `valor_com_desconto`, existirá um similar `valor_com_desconto_formatado`; se houver uma variável de instância `@valor`, haverá um método `valor_formatado` e assim por diante.

Vamos colocar a mão na massa. O primeiro passo é definir o método `formata_moeda`:

```
# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  def self.formata_moeda
    def valor_formatado
      "R$ #{@valor}"
    end

    def valor_com_desconto_formatado
      "R$ #{valor_com_desconto}"
    end
  end
end
```

```

        end
    end

    formata_moeda

    def initialize(titulo, valor, categoria)
        super()
        @titulo = titulo
        @valor = valor
        @categoria = categoria
    end
end

```

O método `formata_moeda` foi adicionado na *singleton class* do objeto da classe `DVD`. Isso significa que posso invocá-lo logo após sua definição sem explicitamente definir qual o *receiver*, pois o *self* neste caso é o objeto que representa a classe `DVD`.

No momento que invocamos o método `formata_moeda`, ele define os métodos `valor_formatado` e `valor_com_desconto_formatado` para todos os objetos `DVD` que forem criados. Mas isso não ajudou muita coisa, pois o que fizemos foi copiar e colar o código definido no módulo `FormatadorMoeda`.

O que precisamos é definir automaticamente estes métodos. Vamos utilizar um método chamado `define_method`, que recebe o nome do método que desejamos criar e um bloco que representa o corpo do método, ou seja, o conteúdo entre o `def` e o `end`. Lembrando que qualquer parâmetro definido no bloco torna-se parâmetro do método que está sendo criado.

A primeira informação que precisamos passar na chamada do método `formata_moeda` são as variáveis de instância que desejamos formatar e também quais os métodos. Na verdade, estas informações serão passadas na chamada do método, porque

precisamos delas para saber quais serão os métodos que serão gerados automaticamente via `define_method`.

```
# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  def self.formata_moeda(*variaveis_e_metodos)
    def valor_formatado
      "R$ #{@valor}"
    end

    def valor_com_desconto_formatado
      "R$ #{valor_com_desconto}"
    end
  end

  formata_moeda :valor_com_desconto, :valor

  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end
end
```

Agora vamos utilizar o método `define_method` para criar os método `valor_formatado` e `valor_com_desconto_formatado`:

```
# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  def self.formata_moeda(*variaveis_e_metodos)
    variaveis_e_metodos.each do |name|
      define_method("#{name}_formatado") do
        valor =
        respond_to?(name) ?
        send(name) : instance_variable_get("@#{name}")
        "R$ #{valor}"
      end
    end
  end
end
```

```

end

formatar_moeda :valor_com_desconto, :valor

def initialize(titulo, valor, categoria)
  super()
  @titulo = titulo
  @valor = valor
  @categoria = categoria
end

end

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_formatado # => R$ 98.9
p windows.valor_com_desconto_formatado # => R$ 89.01

```

Recebemos os valores que devem ser formatados em um Array contendo `[:valor, :valor_com_desconto]`. Sendo assim, precisamos criar os métodos `valor_formatado` e `valor_com_desconto_formatado`. Por isso iteramos o Array para gerar um método para cada valor.

O método `define_method` recebe no primeiro argumento cada uma dos valores do Array interpolado com `_formatado`, e o segundo argumento é o bloco que define o corpo do método, onde definimos seu comportamento. Repare que o bloco não recebe nenhum parâmetro porque o método que estamos definindo não precisa. Lembre-se de que, se fosse necessário receber algum parâmetro no método, deveríamos explicitá-los no bloco.

O segredo maior do nosso código está na busca pelo valor que deve ser formatado. Um detalhe importante: invocamos os métodos `respond_to`, `send` e `instance_variable_get` dentro do corpo do método que estamos criando. Todos estes métodos são invocados no *default receiver*, que neste caso, é o

objeto `windows` . O método `define_method` está criando um método nas instâncias de `DVD` , e não no objeto `Class` que representa a classe `DVD` .

O método `respond_to` , que já havíamos visto, verifica se o objeto possui um determinado método. Este método recebe um `Symbol` , e se o objeto tiver o método, o retorno é `true` ; senão, `false` . Em nosso exemplo, o primeiro item do `Array` é `:valor_com_desconto` , e quando invocamos o método `respond_to?(:valor_com_desconto)` , o retorno é `true` . Logo em seguida, invocamos o método `valor_com_desconto` utilizando um outro método do core Ruby, chamado `send` .

O método `send` recebe um `Symbol` que é o nome do método que desejamos invocar. Caso o método que vamos invocar exija argumentos em sua chamada, podemos passá-los logo após o nome do método separando-os com `,` :

```
def qualquer_metodo(qualquer_argumento, outro_argumento)
  p qualquer_argumento, outro_argumento
end

send(:qualquer_metodo, 123, 321) # => 123, 321
```

O segundo item do `Array` é o `Symbol` `valor` que é uma variável presente nos objetos do tipo `DVD` . Quando fazemos a verificação se o objeto `DVD` possui o método `valor` , o resultado é `false` , já que este método realmente não está definido. Neste caso, nós precisamos acessar o valor da variável `@valor` , porém, variáveis de instância são privadas.

Podemos trapacear esta regra utilizando o método `instance_variable_get` , passando o nome da variável com `@` . Se a variável existir, seu valor será retornado.

Em ambos os casos, guardamos o valor em uma variável local chamada `valor`, que logo em seguida é interpolada com a `String`: `R$`. Para comprovar que os métodos realmente foram criados nos objetos `DVD`, podemos utilizar o método `methods` que retorna um `Array` com todos os métodos disponíveis:

```
windows = DVD.new "Windows 7 for Dummies", 98.9,  
               :sistemas_operacionais  
p windows.methods # => [..., :valor_formatado,  
                        :valor_com_desconto_formatado, ...]
```

Esta técnica de metaprogramação é muito usada em frameworks como Rails, por exemplo. O comportamento que utilizamos é bem parecido com o comportamento usado no método `attr_accessor` que cria novos métodos para acessarmos variáveis de instância de objetos de uma determinada classe.

Compartilhando o método `formata_moeda`

A implementação do método `formata_moeda`, usado para criar métodos que formatam valores de atributos ou métodos criados anteriormente, está restrita para uso dentro a classe `DVD`. Se quisermos utilizá-lo dentro das classes `Livro` e `CD`, podemos movê-lo para dentro da classe `Midia`, já que esta é a superclasse de `Livro` e `CD`.

Porém, se quisermos usá-lo dentro da classe `Revista`, teríamos de fazer com que ela também estendesse `Midia`, uma opção que não faz muito sentido em nosso modelo de negócio. Além de não fazer sentido métodos que formatam valores serem definidos dentro da classe `Midia`, pois isso foge de suas responsabilidades.

O melhor caminho é utilizar um módulo:

```

module FormatadorMoeda
  def formata_moeda(*variaveis_e_metodos)
    variaveis_e_metodos.each do |name|
      define_method("#{name}_formatado") do
        valor =
        respond_to?(name) ?
        send(name) : instance_variable_get("@#{name}")
        "R$ #{valor}"
      end
    end
  end
end

```

E usar dentro das classes que desejamos este comportamento. Utilizando o `extend` na definição da classe, os métodos definidos no módulo serão incluídos como "métodos de classe", ou seja, serão incluídos no objeto `Class` que representa as definições da classe `DVD`. Eles poderão ser usados exatamente da mesma maneira que eram quando definimos o método diretamente na definição da classe.

```

# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  extend FormatadorMoeda

  formata_moeda :valor_com_desconto, :valor

  def initialize(titulo, valor, categoria)
    super()
    @titulo = titulo
    @valor = valor
    @categoria = categoria
  end
end

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
p windows.valor_formatado # => R$ 98.9
p windows.valor_com_desconto_formatado # => R$ 89.01

```

Varags em Ruby

O método `formata_moeda` recebe um `Array` contendo símbolos que representam os nomes das variáveis e métodos para os quais desejamos criar métodos formataores. Existe uma maneira mais elegante de criarmos métodos que precisam receber argumentos com número variável em Ruby:

```
# coding: utf-8
class DVD < Midia
  attr_reader :titulo

  # adicionado o caractere * antes do nome do argumento
  # define que este argumento recebe N valores
  # separados por vírgula

  def self.formata_moeda(*variaveis_e_metodos)
    variaveis_e_metodos.each do |name|
      define_method("#{name}_formatado") do
        valor =
        respond_to?(name) ?
          send(name) : instance_variable_get("@#{name}")
          "R$ #{valor}"
        end
      end
    end
  end

  formata_moeda :valor_com_desconto, :valor

  # outros metodos
end
```

Basta adicionar o caractere `*` (asterisco) antes do nome do argumento que receberá vários valores. Dentro do método, o argumento é um `Array`, que contém os vários valores passados. Podemos adicionar apenas um argumento que receberá vários valores, porque seria impossível para o interpretador Ruby atribuir os valores corretamente se tivéssemos dois argumentos deste tipo.

Definindo métodos de classe e de instância no módulo

Caso seja necessário definir métodos de instância e de classe dentro do mesmo módulo, existe uma técnica muito usada em frameworks como Rails. Esta técnica faz uso de um *hook method* do Ruby chamado `included`, que é chamado automaticamente quando incluímos um módulo em uma classe. Neste método, recebemos um objeto `Class` que representa a classe que está incluindo o módulo.

Mesmo assim, é necessário separar os métodos de instância dos métodos de classe. Geralmente fazemos isso utilizando um *nested module* que contém apenas os métodos de classe, deixando os métodos de instância na definição do próprio módulo, por exemplo:

```
module FormatadorMoeda
  def metodo_de_instancia
    "um metodo de instancia qualquer"
  end

  # Módulo que guarda os métodos de classe
  module ClassMethods
    def formata_moeda(*variaveis_e_metodos)
      variaveis_e_metodos.each do |name|
        define_method("#{name}_formatado") do
          valor = respond_to?(name) ?
            send(name) :
            instance_variable_get("@#{name}")
          "R$ #{valor}"
        end
      end
    end
  end

  end

  # hook method que é executado quando incluímos o módulo
  # dentro de alguma classe, recebendo no argumento
  # classe_que_incluiu_modulo o objeto Class que
  # representa a classe que incluiu o módulo
```

```

    def self.included(classe_que_incluiu_modulo)
      classe_que_incluiu_modulo.extend ClassMethods
    end
  end

  # coding: utf-8
  class DVD < Midia
    attr_reader :titulo

    include FormatadorMoeda

    formata_moeda :valor_com_desconto, :valor

    def initialize(titulo, valor, categoria)
      super
      @titulo = titulo
      @valor = valor
      @categoria = categoria
    end
  end

  windows = DVD.new "Windows 7 for Dummies", 98.9,
    :sistemas_operacionais
  p windows.valor_formatado
  # => R$ 98.9
  p windows.valor_com_desconto_formatado
  # => R$ 89.01
  p windows.metodo_de_instancia
  # => "um metodo de instancia qualquer"

```

Utilizando esta técnica, podemos criar módulos que possuem métodos que serão adicionados nas instâncias de `DVD`, e também métodos que podem ser usados na definição da classe.

8.6 CRIANDO UM FRAMEWORK PARA PERSISTIR OBJETOS EM ARQUIVOS

A principal camada de persistência difundida na comunidade Ruby é conhecida como *ActiveRecord*, a implementação de um

design pattern de mesmo nome que visa persistir dados de um objeto através de interfaces públicas definidas no próprio objeto. Essa interface geralmente possui os métodos `create` , `update` , `find` e `delete` .

Atualmente o código da classe `BancoDeArquivos` recebe um objeto do tipo `Livro` e o salva em um arquivo chamado `livros.yml` . A partir de agora, faremos uma refatoração, em que a classe deixará de existir e os métodos para inserir, atualizar, buscar e remover um objeto `Livro` do banco de arquivos serão feitos através de uma interface no próprio objeto.

Vamos começar fazendo o código apenas na classe `Revista` , que foi refatorada para ter apenas um atributo `titulo` e um valor :

```
class Revista
  attr_reader :titulo, :id

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
  end
end
```

O próximo passo será definir um método `save` que fará uma implementação dupla. Ele será responsável por criar um arquivo cujo nome será um `id` automático gerado para cada objeto `Revista` e gravar o conteúdo do objeto dentro deste arquivo. Também será responsável por atualizar o conteúdo do arquivo caso ele já exista.

Dentre estes passos, o primeiro é criar um campo `id` e atribuir um valor automático a ele:

```
class Revista
```

```

attr_reader :titulo, :id

def initialize(titulo, valor)
  @titulo = titulo
  @valor = valor
  @id = self.class.next_id
  # Atribui um id ao objeto Revista
end

private

def self.next_id
  Dir.glob("db/revistas/*.yaml").size + 1
end
end

```

Criamos um método de classe (`next_id`) que gera o próximo `id` para o objeto que está sendo criado. Sua implementação é bem simples: ele utiliza a classe `Dir` da API `File` do Ruby, para contar quantos arquivos `.yaml` existem dentro da pasta `db/revistas` onde estão localizados os arquivos referentes aos objetos salvos em disco.

Somando o valor 1 à quantidade de arquivos, sabemos qual é o `id` do próximo arquivo gerado. Definimos também um método para acessarmos o `id` do objeto que foi criado.

Agora basta criarmos o método `save` que será responsável por guardar o conteúdo do objeto serializado dentro de um arquivo cujo nome será o `id` do objeto:

```

class Revista
  attr_reader :titulo, :id

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
  end
end

```



```

def save
  File.open("db/revistas/#{@id}.yaml", "w") do |file|
    file.puts serialize
  end
end

private

def serialize
  YAML.dump self
end

def self.next_id
  Dir.glob("db/revistas/*.yaml").size + 1
end
end

```

A implementação do método `save` abre um arquivo na pasta `db/revistas/#{id_do_objeto}.yaml` (cria a pasta `db/revistas` na raiz do projeto) em modo de escrita e imprime dentro dele o conteúdo do próprio objeto (`self`) serializado em YAML. Podemos comprovar que o comportamento foi implementado com sucesso executando o código a seguir:

```

mundo_j = Revista.new "Mundo J", 10.9
mundo_j.save

```

Ao abrirmos o arquivo `db/revistas/1.yaml` , veremos o seguinte conteúdo:

```

--- !ruby/object:Revista
valor: 10.9
titulo: Mundo J
id: 1

```

Exatamente o conteúdo que existia no objeto `mundo_j` .

O método `save` também será responsável por salvar as atualizações feitas no objeto. Felizmente não será necessário alterar

nenhuma linha de código no método, já que quando abrimos o arquivo utilizando o modo `w`, qualquer impressão feita dentro dele substitui o conteúdo existente pelo novo.

Porém, para testar que a atualização está funcionando, precisamos permitir que os dados do objeto sejam alterados. Vamos fazer com que o `valor` possa ser alterado adicionando um método de escrita:

```
class Revista
  attr_reader :titulo, :id
  attr_accessor :valor # permite escrita no atributo valor

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
  end

  def save
    File.open("db/revistas/#{@id}.yaml", "w") do |file|
      file.puts serialize
    end
  end

  private

  def serialize
    YAML.dump self
  end

  def self.next_id
    Dir.glob("db/revistas/*.yaml").size + 1
  end
end
```

Agora vamos alterar o `valor` de um objeto `Revista` e salvá-lo para verificar que as alterações serão refletidas no arquivo correspondente:

```
mun
do_j = Revista.new "Mundo J", 10.9
mun
do_j.save
```

```
mun
do_j.valor = 12.9
mun
do_j.save
```

Ao abrirmos o arquivo `db/revistas/1.yml` , podemos ver que o conteúdo realmente foi alterado como esperávamos:

```
--- !ruby/object:Revista
valor: 12.9
titulo: Mundo J
id: 1
```

Buscando objetos

Com isso, completamos duas operações principais do CRUD (*Create, Retrieve, Update, Delete*): Create e Update. Vamos agora implementar o método que recebe um `id` , busca pelo arquivo correspondente, e retorna o objeto deserializado:

```
class Revista
  attr_reader :titulo, :id
  attr_accessor :valor

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
  end

  def save
    File.open("db/revistas/#{@id}.yml", "w") do |file|
      file.puts serialize
    end
  end

  def self.find(id)
    YAML.load File.open("db/revistas/#{id}.yml", "r")
  end
end
```

```

private

def serialize
  YAML.dump self
end

def self.next_id
  Dir.glob("db/revistas/*.yaml").size + 1
end
end

```

O código não está muito diferente do encontrado na classe `BancoDeArquivos`, implementada no capítulo *Explorando API File*. A diferença principal é que criamos o método `find` como método do objeto `Class`, referente a classe `Revista`. Isso faz bastante sentido, já que não temos a instância do objeto que estamos procurando. Nós na verdade queremos criá-la a partir do conteúdo do arquivo. Em frameworks como **Active Record**, esse tipo de comportamento é comum.

Outra diferença é que agora nós guardamos apenas 1 objeto serializado por arquivo, por isso, no momento da leitura, precisamos apenas usar o método `open` da classe `File` e passar o seu resultado para ser deserializado pela API de YAML.

Invocando o método `find` passando com argumento um `id` cujo arquivo exista em disco, o objeto existente será deserializado e retornado:

```

mundo_j = Revista.find 1
p mundo_j.valor # => 12.9

```

Como vimos, caso ele exista, um objeto do tipo `Revista` é retornado para o cliente do método `find`. Mas a pergunta que fica é: *quando o arquivo for removido ou não existir em disco, o que o método `find` retornará?*

A decisão de como implementar este comportamento é bem extensa e causa grandes discussões. Uns vão preferir o comportamento que já existe, que consiste em retornar `nil` caso o arquivo referente ao `id` não seja encontrado.

Retornar `nil`, na minha opinião, não funciona muito bem, afinal, não encontrar o arquivo que contém os dados do objeto que está sendo procurado é um erro. Quando retornamos `nil`, não estamos expressando o que realmente aconteceu de fato e, com isso, o cliente do método não sabe como realmente tratar a possível falha.

Alguns desenvolvedores tem o péssimo hábito de inventar códigos de erro, que em geral são número aleatórios de 1 à 1000, para expressar um erro que aconteceu no sistema. Todos nós aqui já nos deparamos com coisas do gênero em algum momento de nossas vidas, o que convenhamos não é nada legal, já que o número, exceto se for decorado, não representa muita informação. Portanto, **nunca** faça isso. No dia que fizer, provavelmente você se lembrará de mim e terá de correr 30 km em uma esteira no sol do meio dia.

Outras preferem retornar exceções, o meu caso, avisando ao cliente do método `find` que ele não deveria invocar o método passando um `id` que não existe mais. Esse comportamento também é seguido pelo Active Record, que retorna uma exceção do tipo `DocumentNotFound`.

Exceções na verdade são objetos que encapsulam o erro. O cliente do método tem a opção de se recuperar do possível erro, ou ela será propagada até o final da stack de execução, estourando um erro para o usuário, provavelmente.

O método `find`, como foi dito anteriormente, precisa retornar uma exceção caso o arquivo referente ao `id` passado não seja encontrado. Porém, exceções são classes, e precisam ser definidas em algum momento. O Ruby possui uma grande hierarquia de exceções que podem ser vistas na primeira figura deste capítulo, e você verá que essa hierarquia nos ajuda muito no momento de criarmos código para tratar as exceções.

8.7 GERENCIANDO EXCEÇÕES E ERROS

Geralmente, quando precisamos disparar exceções, criamos classes que sejam filhas de `StandardError`, uma subclasse de `Exception`. A classe `StandardError` é a superclasse da maioria das exceções que nos deparamos até agora: `NoMethodError`, por exemplo. No caso do método `find`, vamos retornar uma exceção própria que chamaremos de `DocumentNotFound`, que deve ser adicionada no arquivo `lib/document_not_found.rb`:

```
class DocumentNotFound < StandardError
end
```

Agora, basta dispararmos esta exceção customizada quando o arquivo não existir:

```
# coding: utf-8
class Revista
  def self.find(id)
    raise DocumentNotFound,
      "Arquivo db/revistas/#{id} não encontrado.", caller
    unless File.exists?("db/revistas/#{id}.yaml")
      YAML.load File.open("db/revistas/#{id}.yaml", "r")
    end
  end
end
```

Nós disparamos exceções utilizando um método do `Kernel`

chamado `raise` , que neste caso é executado apenas se o método `File.exists?` retornar `false` , ou seja, caso o arquivo não exista em disco. O método `raise` interrompe o fluxo de execução do método, por isso, se o método `raise` for invocado, existe a garantia de que o código `YAML.load # ...` não será executado.

A forma que utilizamos o método `raise` dispara uma exceção do tipo `DocumentNotFound` , com uma mensagem de erro que representa uma explicação do erro retornado. Por último utilizamos mais um método do `Kernel` , o `caller` , que retorna a *Stack Trace* que informa mais organizadamente onde o erro aconteceu e onde poderia ter sido tratado.

RAISE E SUAS VARIAÇÕES

Podemos invocar o método `raise` passando apenas um `String` que representa a mensagem de erro. Esta será inserida na exceção que está sendo lançada.

```
raise "Aconteceu um erro qualquer"
```

Neste caso, o tipo de exceção disparada será: `RuntimeError` .

EXCEÇÕES MAIS INFORMATIVAS

Lembre-se de que as exceções são objetos normais, e possuem uma definição formal escrita em uma classe. Você, assim como em qualquer outro objeto, pode adicionar métodos, inclusive definir seu próprio `initialize`, que receberá os argumentos desejados:

```
class DocumentNotFound < StandardError
  def initialize(mensagem)
    @mensagem = mensagem
  end

  def mensagem_formatada
    "-- #{@mensagem}"
  end
end
```

Como tratar os erros

O que acontecerá quando invocarmos o método `find` passando como argumento o `id` de um arquivo que não existe? Uma exceção será lançada, e o seguinte erro será impresso:

```
Revista.find 42 # => DocumentNotFound: Arquivo db/revistas/42
                                     não encontrado.
```

E também será impressa a *stack trace* com as chamadas e onde realmente o erro estourou. O que não sabemos ainda é como lidar com este erro e não interromper o fluxo de execução do nosso sistema. Vamos adicionar um tratador de exceções, ou melhor, um *exception handler*:

```
begin
```



```

    Revista.find 42
  rescue
    p "O objeto que estava procurando não foi encontrado."
  end

# => O objeto que estava procurando não foi encontrado.

```

Todo o conteúdo existente que está dentro do `begin` está protegido pelo *exception handler* `rescue` que foi definido. Quando não passamos nenhum argumento para o `rescue`, ele trata exceções de tipo ou de subtipos de `StandardError`. Isto é, se o código `Revista.find 42` retornar qualquer exceção destes tipos, o `p` que colocamos será executado.

Caso você queira restringir o poder de tratamento de erro do `rescue`, você pode especificar a exceção ou exceções que deseja tratar:

```

begin
  Revista.find 42
rescue DocumentNotFound
  p "O objeto que estava procurando não foi encontrado."
end

begin
  Revista.find 42
rescue DocumentNotFound, OutraExcecaoAqui
  p "O objeto que estava procurando não foi encontrado."
end

```

Você pode também receber em uma variável local, o objeto que representa o erro que aconteceu:

```

begin
  Revista.find 42
rescue DocumentNotFound => erro
  p erro
end

```

Vários rescues

Você pode definir vários *exception handlers* após um `begin` :

```
begin
  Revista.find 42
rescue DocumentNotFound
  p "Documento nao encontrado"
rescue OutraExcecaoAqui
  p "Outra excecao"
end
```

Bastante parecido com um `switch/case`. Caso a exceção bata com o primeiro `rescue` , a mensagem "Documento nao encontrado" será impressa; caso não, o *match* será feito com a exceção `OutraExcecaoAqui` . Se bater, a mensagem "Outra excecao" será impressa.

Operações obrigatórias com o `ensure`

Algumas vezes é importante que algum processo seja feito caso o código seja executado com sucesso ou caso alguma exceção seja tratada. O exemplo mais clássico é quando estamos lendo um arquivo utilizando a classe `File` , e precisamos garantir que ele seja fechado após a execução do código.

Para garantir este comportamento, usamos a cláusula `ensure` logo após a cláusula `rescue` , onde definimos um pedaço de código que será executado se o código dentro do `begin` executar normalmente, se tratarmos uma exceção utilizando o `rescue` , ou até mesmo se acontecer alguma exceção que não havia sido prevista.

```
file = File.open("/tmp/file")
begin
  p file.read
```

```
rescue
  p "tratando erro"
ensure
  file.close
end
```

Retry

Outras vezes é necessário tentar novamente a execução de uma tarefa definida dentro de um bloco `begin end`. Por exemplo, ao tentar inserir um registro no banco de dados, caso o banco esteja indisponível, podemos tentar novamente. É possível alcançar este objetivo usando o método `retry` dentro da cláusula `rescue`:

```
begin
  db.collection("people").insert({nome: "Lucas Souza"})
rescue
  p "Conexao está indisponível"
  retry # executa o bloco begin/end novamente
end
```

Caso alguma exceção aconteça, a tentativa de inserção no banco de dados será executada novamente. É muito importante salientar que o código anterior pode se tornar um *loop* infinito, caso o banco não esteja disponível em nenhum momento. O ideal é criar *flags* que garantam um número mínimo de tentativas, ou ter realmente certeza do que está fazendo.

8.8 A EXCLUSÃO DE DADOS IMPLEMENTADA COM METAPROGRAMAÇÃO

A última operação que falta para completarmos o CRUD da classe `Revista` é a exclusão de objetos deste tipo. A implementação do método `destroy` será bastante simples,

porém, causará alguns efeitos colaterais. Vamos ao código:

```
# coding: utf-8
require "fileutils"

class Revista
  attr_reader :titulo, :id
  attr_accessor :valor

  # initialize

  # save

  # destroy

  def destroy
    FileUtils.rm "db/revistas/#{@id}.yaml"
  end

  # métodos privados
end
```

Usamos a classe `FileUtils`, que possui uma série de métodos utilitários para lidar com arquivos. Lembre-se de que é necessário fazer um `require` desta classe para poder utilizá-la. A implementação do método `destroy` é bem simples, quando invocado uma tentativa para excluir o arquivo cujo nome é a variável de instância, a `@id.yaml` é feita utilizando o método `rm` da classe `FileUtils`.

O comportamento pode ser testado executando o seguinte código:

```
munido_j = Revista.find 1
munido_j.destroy

Revista.find 1
# => DocumentNotFound: Arquivo db/revistas/1 não encontrado
```

Repare que na segunda vez que tentamos buscar um objeto

Revista através do método `find` , uma exceção é disparada, garantindo que o objeto realmente foi excluído quando invocamos o método `destroy` .

Mas o que acontece caso o método `destroy` seja invocado duas vezes em sequência?

```
mun
do_j = Revista.find 1
mun
do_j.destroy
mun
do_j.destroy # => Errno::ENOENT: No such file or
              directory - db/revistas/1.yml
```

Na segunda tentativa, o arquivo já não existe mais em disco, por este motivo a exceção `No such file or directory` foi disparada. Porém, não faz sentido essa exceção ser disparada quando um usuário está invocando o método `destroy` , que como o próprio nome diz, destrói o objeto em questão e não diz nada sobre lidar com arquivos em disco. Toda esta lógica está encapsulada dentro do método e os seus detalhes não devem ser expostos para os clientes da API.

Para contornar este problema, vamos adicionar uma *flag* chamada `destroyed` que será inicializada com o valor `false` , e que terá seu valor alterado para `true` na primeira chamada ao método `destroy` :

```
# coding: utf-8
require "FileUtils"

class Revista
  attr_reader :titulo, :id, :destroyed
  attr_accessor :valor

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
  end
end
```

```

        @destroyed = false
    end

    def destroy
        unless @destroyed
            @destroyed = true
            FileUtils.rm "db/revistas/#{@id}.yaml"
        end
    end
end
end

```

Ao tentarmos invocar o método `destroy` por duas vezes, a segunda chamada não dispara nenhuma exceção e a variável `@destroyed` possui o valor `true` :

```

mundo_j = Revista.find 1
mundo_j.destroy
mundo_j.destroy # => nil
p mundo_j.destroyed # => true

```

Mas como nem tudo é perfeito, ainda existe um problema em nosso método `destroy` . Caso algum objeto seja criado diretamente pelo método `new` , o valor da variável `@destroyed` será `false` . Assim, será possível invocar o método `destroy` .

O problema é que o objeto acabou de ser criado, e com certeza ainda não existe um arquivo que guarde seus valores. Ao invocarmos o método, caímos no mesmo problema anterior:

```

mundo_j = Revista.new "Mundo Java", 10.9
mundo_j.destroy # => Errno::ENOENT: No such file or directory

```

Podemos resolver este problema adicionando uma outra *flag* que será responsável por guardar a informação se o objeto é um novo registro ou não, por isso, vamos chamá-la de `@new_record` . Esta *flag* será inicializada com `true` e precisa ter seu valor alterado para `false` quando o objeto for salvo através do método `save` :

```

# coding: utf-8
require "FileUtils"

class Revista
  attr_reader :titulo, :id, :destroyed, :new_record
  attr_accessor :valor

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
    @destroyed = false
    @new_record = true
  end

  def save
    # seta objeto como registro existente
    @new_record = false

    File.open("db/revistas/#{@id}.yaml", "w") do |file|
      file.puts serialize
    end
  end

  def destroy
    # verificar se o objeto não foi removido anteriormente
    # e se o objeto não é um novo registro
    unless @destroyed or @new_record
      @destroyed = true
      FileUtils.rm "db/revistas/#{@id}.yaml"
    end
  end
end
end

```

Agora no momento que criarmos um objeto diretamente pelo método `new` e tentarmos removê-lo, nada será executado:

```

mundo_j = Revista.new "Mundo Java", 10.90
mundo_j.destroy # => nil

```

Caso o objeto seja salvo antes de ser removido, o método `destroy` é executado normalmente:

```
# cria e salva objeto revista
mundo_j = Revista.new "Mundo Java"
mundo_j.save
mundo_j.destroy

# verifica que o objeto já não existe mais
Revista.find 1 # => DocumentNotFound
```

Mas você deve estar se perguntando se isso não causaria problemas na execução do método `find`, pois aparentemente o método `YAML.load`, ao deserializar o conteúdo do arquivo, executa o método `initialize`, certo? Errado.

O processo de deserialização não executa o método `initialize`, ele apenas recupera os valores salvos em disco, diretamente para as variáveis de instância. Por este motivo, quando executamos o método `find`, o objeto retornado possui a variável de instância `@new_record` com o valor `false`, já que antes de salvar o objeto, nós definimos o valor desta variável para `false`:

```
mundo_j = Revista.find 1
mundo_j.new_record # => false
```

Código com muitas responsabilidades

O código da classe `Revista` possui muitas responsabilidades. Além de guardar todos os atributos da classe, os métodos responsáveis por refletir o estado do objeto no seu arquivo em disco estão todos definidos na própria classe. Além disso, se quisermos aproveitar estes métodos em outras classes, teríamos de duplicar o código, ferindo o princípio do DRY (*Don't Repeat Yourself*) e tendo muito trabalho no momento em que for necessário alterar algum destes comportamentos.

Podemos isolar o código responsável por lidar com arquivos

em uma classe, e utilizar herança para reaproveitar os comportamentos definidos. Porém, se no futuro existir a necessidade de utilizar herança para um propósito melhor, não podemos herdar de duas classes. Neste caso, seria necessário uma grande refatoração para suportar esta mudança.

A melhor solução neste cenário é utilizar os *mixings* para compartilhar comportamentos, incluindo-os nas classes que desejarmos. Vamos começar a nossa refatoração seguindo *baby steps*, definindo primeiro um módulo com os métodos `save`, `destroy` e `find` de maneira que todos os códigos que usam a classe `Revista` continuem funcionando.

```
# coding: utf-8
require "FileUtils"

module ActiveFile
  def save
    @new_record = false

    File.open("db/revistas/#{@id}.yaml", "w") do |file|
      file.puts serialize
    end
  end

  def destroy
    unless @destroyed or @new_record
      @destroyed = true
      FileUtils.rm "db/revistas/#{@id}.yaml"
    end
  end

  module ClassMethods
    def find(id)
      raise DocumentNotFound,
        "Arquivo db/revistas/#{id} nao encontrado.", caller
      unless File.exists?("db/revistas/#{id}.yaml")
        YAML.load File.open("db/revistas/#{id}.yaml", "r")
      end
    end
  end
end
```

```

    def next_id
      Dir.glob("db/revistas/*.yaml").size + 1
    end
  end

  def self.included(base)
    base.extend ClassMethods
  end

  private

  def serialize
    YAML.dump self
  end
end

```

Lembre-se de criar o módulo anterior dentro do arquivo `lib/active_file.rb` e executar o seu `require` dentro do arquivo `lib/loja_virtual.rb`.

Agora, podemos utilizar o módulo `ActiveFile` e remover boa parte do código que estava sendo definida na classe `Revista`:

```

class Revista
  attr_reader :titulo, :id, :destroyed, :new_record
  attr_accessor :valor

  include ActiveFile

  def initialize(titulo, valor)
    @titulo = titulo
    @valor = valor
    @id = self.class.next_id
    @destroyed = false
    @new_record = true
  end
end

```

O código da classe `Revista` ficou bem mais conciso e enxuto. Mas o método `initialize` ainda possui detalhes em sua implementação que são necessários para que o módulo

`ActiveFile` funcione corretamente, por exemplo, a criação as variáveis `@id` , `@new_record` e `@destroyed` .

Com esta dependência de código, todas as classes que usarem o módulo `ActiveFile` terão de definir exatamente as mesmas variáveis. A melhor maneira de resolver este problema é imitar frameworks, como `ActiveRecord` e `Mongoid` , definindo o método `initialize` dentro do *mixing* e controlando a criação de variáveis através de métodos de classe.

Vamos tomar como exemplo um código utilizando `Mongoid` :

```
class Revista
  include Mongoid::Document

  field :titulo, type: String
end

mundo_j = Revista.new titulo: "MundoJ"
```

A chamada ao método `field` define que os objetos do tipo `Revista` terão uma propriedade `titulo` , cujo valor é do tipo `String` . Vamos implementar nossa solução de maneira bem parecida, exceto pelo tipo, que no momento não é tão importante. Nosso primeiro passo será criar o método `field` :

```
module ActiveFile
  # outros metodos

  module ClassMethods
    # metodo find

    # metodo next_id
    def field(name)
      @fields ||= []
      @fields << name
    end
  end
end
```

```

def self.included(base)
  base.extend ClassMethods
end
end

```

O método é responsável apenas por guardar em um `Array` chamado `@fields` quais serão os atributos dos objetos que incluïrem o módulo `ActiveFile`.

Utilizando `class_eval` para ajudar na definição da classe

Além de guardar quais serão os atributos que formarão o objeto a ser salvo em arquivo, o método `field` cria os métodos acessores: `get` e `set` para cada um dos `fields` definidos. A variável `self` dentro do método `field` é a instância de `Class` referente a classe que está incluindo o módulo, em nosso caso, a classe `Revista`.

Para definir métodos acessores para um determinado `field`, podemos invocar o método `attr_accessor` passando a variável `name`:

```

module ActiveFile
  module ClassMethods
    def field(name)
      @fields ||= []
      @fields << name

      self.attr_accessor name
    end
  end

  def self.included(base)
    base.extend ClassMethods
  end
end

```

Entretanto, esse código não funciona, porque o método `attr_accessor` é privado e não pode ser invocado com um *receiver* explícito. A solução pode ser a utilização do método `send`, que não diferencia se estamos invocando um método `private`, `protected` ou `public`. Mas vamos aprender uma outra maneira de definir métodos de instância, desta vez usando metaprogramação.

O método `class_eval` da classe `Object`, quando utilizado, permite que um bloco de código seja executado como se ele estivesse escrito na definição da classe. Os códigos seguintes possuem o mesmo efeito:

```
class String
  def plural
    "#{self}s"
  end
end

p "cachorro".plural # cachorros

String.class_eval do
  def plural
    "#{self}s"
  end
end

p "cachorro".plural # cachorros
```

Ambos definem o método `plural` para instâncias de `String`, ou seja, ambos abrem a classe `String` e definem um método chamado `plural`.

INSTANCE_EVAL

O `instance_eval` também pertence a classe `Object` e serve para executarmos um bloco de código. Porém, a diferença principal entre ele e o método `class_eval` é que os métodos definidos dentro do bloco passado ao `instance_eval` são incluídos na *singleton class* de `self`. Desta forma, ao invocarmos o método `instance_eval` em uma instância `Class` referente a classe `String`, os métodos definidos se tornarão métodos de classe:

```
String.instance_eval do
  def metodo_de_classe
    "Isso eh um metodo de classe"
  end
end

# Isso eh um metodo de classe
p String.metodo_de_classe
```

Usando o método `class_eval`, é possível definirmos os métodos necessários para acessar o valor de um determinado `field`:

```
module ActiveFile
  module ClassMethods
    def field(name)
      @fields ||= []
      @fields << name

      get = %Q{
        def #{name}
          @#{name}
        end
      }
    end
  end
end
```

```

        set = %Q{
            def #{name}=(valor)
                @#{name}=valor
            end
        }

        self.class_eval get
        self.class_eval set
    end
end

def self.included(base)
    base.extend ClassMethods
end
end

```

Nas variáveis locais `get` e `set`, definimos duas `String` que possuem o conteúdo exato dos métodos criados automaticamente pelo método `attr_accessor`. Após definir estas duas variáveis, cada uma delas é passada ao método `class_eval` que faz o `evaluate` e executa o resultado como um código Ruby qualquer.

O segundo passo da nossa refatoração é definir um método `initialize` genérico para todos os objetos que incluam o módulo `ActiveFile`. Neste método, definiremos os valores das variáveis `@id`, `@new_record` e `@destroyed`:

```

module ActiveFile
    def included(base)
        base.extend ClassMethods
        base.class_eval do
            attr_reader :id, :destroyed, :new_record

            def initialize
                @id = self.class.next_id
                @destroyed = false
                @new_record = true
            end
        end
    end
end

```

end

O método `initialize` será definido na classe que inclui o módulo, quando o *hook method* `included` for executado. A implementação do método `initialize` está parcialmente feita, mas já permite que o método `initialize` seja removido da classe `Revista`. Além disso, podemos definir os `fields` da classe utilizando o método `field`, que cria as variáveis de instância e os métodos para acessá-las:

```
class Revista
  include ActiveFile

  field :titulo
  field :valor
end

revista = Revista.new
p revista.new_record # => true
p revista.id # => quantidade de revistas + 1

revista.titulo = "Veja"
revista.valor = 10.90
revista.save
```

Todas as operações que fazíamos anteriormente continuam funcionando da mesma maneira. A única restrição é que antes era possível inicializar os objetos passando os valores dos atributos através da chamada ao método `new`. Entramos na terceira parte da refatoração, que possibilitará a inicialização das variáveis de instância.

Simulando named parameters

Named parameters é a capacidade de invocar funções ou métodos que recebem os valores dos atributos não por sua ordem, mas seguindo o nome de cada um deles. Em linguagens sem esta

característica, é necessário seguir a ordem de definição dos parâmetros do método, por exemplo:

```
def metodo(primeiro_parametro, segundo_parametro)
  p primeiro_parametro, segundo_parametro
end

metodo 10, "segundo" # => 10, "segundo"
metodo "segundo", 10 # => "segundo", 10
```

Como podemos ver, a ordem realmente é importante. Outra desvantagem é que não fica claro o que representa cada valor que passamos na chamada do método, a única maneira de descobrir estas informações é olhando o código-fonte, o que nem sempre é possível. Ao utilizar *named parameters*, podemos definir o valor de cada atributo na chamada do método:

```
def metodo(primeiro_parametro: 1, segundo_parametro: 2)
  p primeiro_parametro, segundo_parametro
end

metodo primeiro_parametro: 10, segundo_parametro: "segundo"

# => 10, "segundo"

metodo segundo_parametro: "segundo", primeiro_parametro: 10
# => 10, "segundo"
```

Repare que não foi necessário seguir a ordem de definição dos parâmetros do método, podemos passá-los na ordem que desejamos. No Ruby 1.9 e versões anteriores, é possível simular o comportamento dos *named parameters* usando Hash :

```
def metodo(parametros)
  p parametros[:primeiro_parametro],
    parametros[:segundo_parametro]
end

metodo primeiro_parametro: 10, segundo_parametro: "segundo"
# => 10, "segundo"
```

```
metodo segundo_parametro: "segundo", primeiro_parametro: 10
# => 10, "segundo"
```

Recebemos um `Hash` e, ao invocarmos o método, as chaves são os "nomes" dos atributos e os valores são, obviamente, os valores dos atributos relacionados a cada chave. É muito comum encontrarmos códigos escritos em Ruby que usam `Hash` para simular os *named parameters*, por isso, utilizaremos esta técnica para passar os valores dos atributos dos objetos que incluem o módulo `ActiveFile` :

```
module ActiveFile
  def included(base)
    base.extend ClassMethods
    base.class_eval do
      attr_reader :id, :destroyed, :new_record

      def initialize(parameters = {})
        @id = self.class.next_id
        @destroyed = false
        @new_record = true

        parameters.each do |key, value|
          instance_variable_set "@#{key}", value
        end
      end
    end
  end
end
```

Iteramos o argumento `parameters` e, para cada atributo (`key`), definimos seu respectivo valor (`value`) utilizando o método `instance_variable_set` . Existe uma ressalva a ser feita: o argumento `parameters` é inicializado com um `Hash` vazio para, quando o método `new` for chamado sem nenhum argumento, o código funcione normalmente.

Agora ao criar um objeto do tipo `Revista` , podemos passar o

valor dos atributos `@titulo` e `@valor` :

```
revista = Revista.new titulo: "Veja", valor: 10.90
```

```
p revista.titulo # => Veja
```

```
p revista.valor # => 10.90
```

```
revista.save
```

Com isso, eliminamos todo o código existente na classe `Revista` referente a persistência dos dados em arquivos.

8.9 METHOD LOOKUP E METHOD MISSING

Vamos relembrar de alguns conceitos vistos neste capítulo. A primeira coisa importante que temos de lembrar é sobre **variáveis de instância**. Elas **sempre** estão dentro do contexto de um objeto, seja ela uma instância de `Class` ou de qualquer outra classe definida; isso que dizer que o interpretador Ruby sempre vai procurá-las dentro do *current object*.

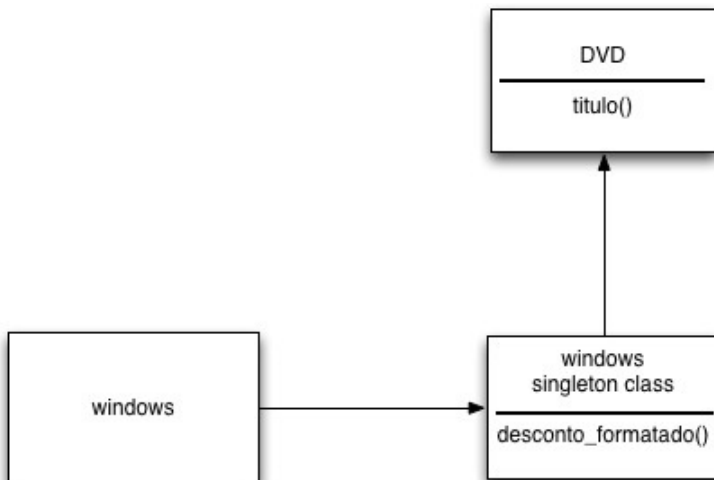
Quando falamos de métodos, a história é um pouco diferente. Existem *containers* nos quais podemos definir métodos: classes, módulos e as *singleton classes*. Relembrando brevemente cada um deles, temos:

- **Classes:** são instâncias de `Class` que representam as definições de um modelo em nosso sistema. Por exemplo o objeto `Class` que guarda as informações referentes à `Revista`.
- **Módulos:** são parecidos com classes, porém, não podem ser instanciados. Servem como *containers* de método que serão compartilhados com várias classes do sistema.
- **Singleton class:** instância criada automaticamente para

armazenar informações referentes a um objeto específico. Através dela, podemos definir comportamentos específicos por objeto.

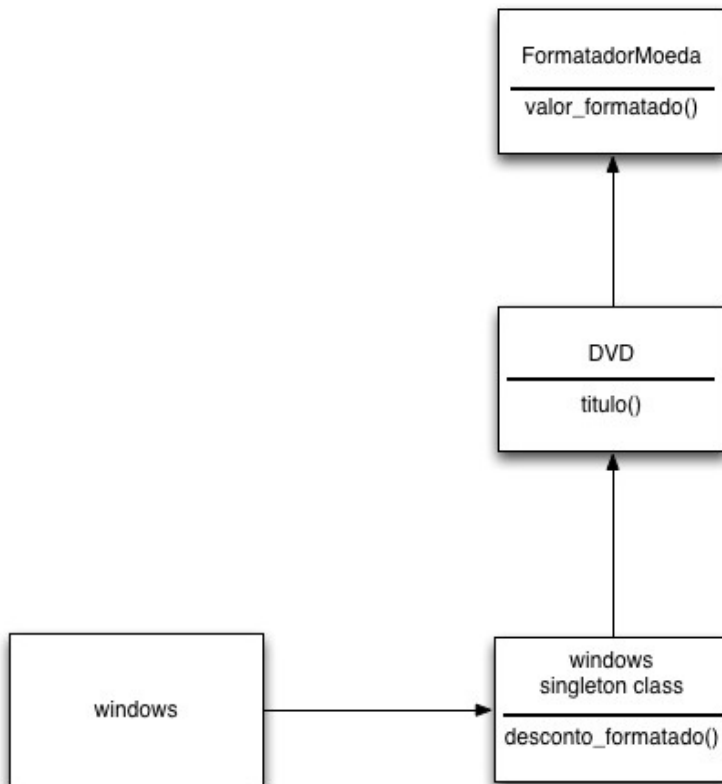
Existe um conceito muito importante que já foi visto em várias partes do capítulo, porém, não vimos concretamente como funciona este processo. O *method lookup* consiste na maneira que o interpretador Ruby procura pelo método que precisa ser executado. Quando invocamos um método em um objeto do tipo DVD , por exemplo, o primeiro lugar onde o interpretador vai procurá-lo será na classe que armazena os métodos específicos dos objetos DVD , ou seja, o objeto do tipo Class referente a classe DVD :

```
windows = DVD.new "Windows 7 for Dummies", 98.9,  
               :sistemas_operacionais  
windows.titulo
```

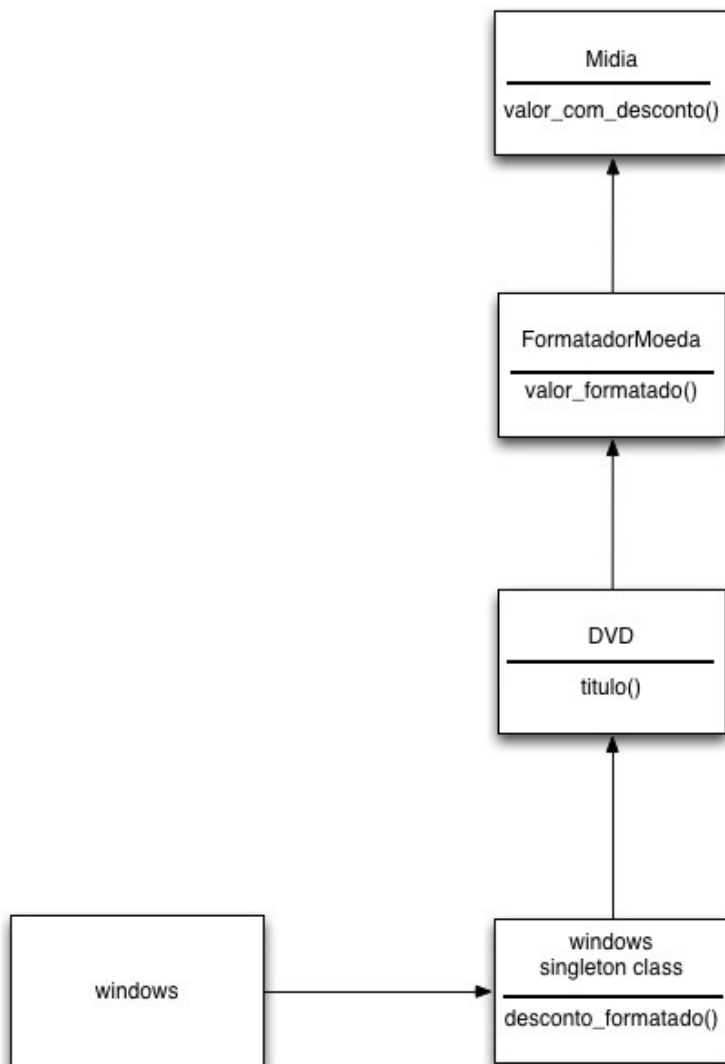


Caso o método que foi invocado não exista dentro da classe

DVD , o interpretador vai procurá-lo nos módulos que foram incluídos na classe DVD , seguindo a ordem inversa da inserção deles. Portanto, tome cuidado com a ordem que você insere os *mixings*; caso eles possuam métodos 'iguais', o último método incluído será executado.

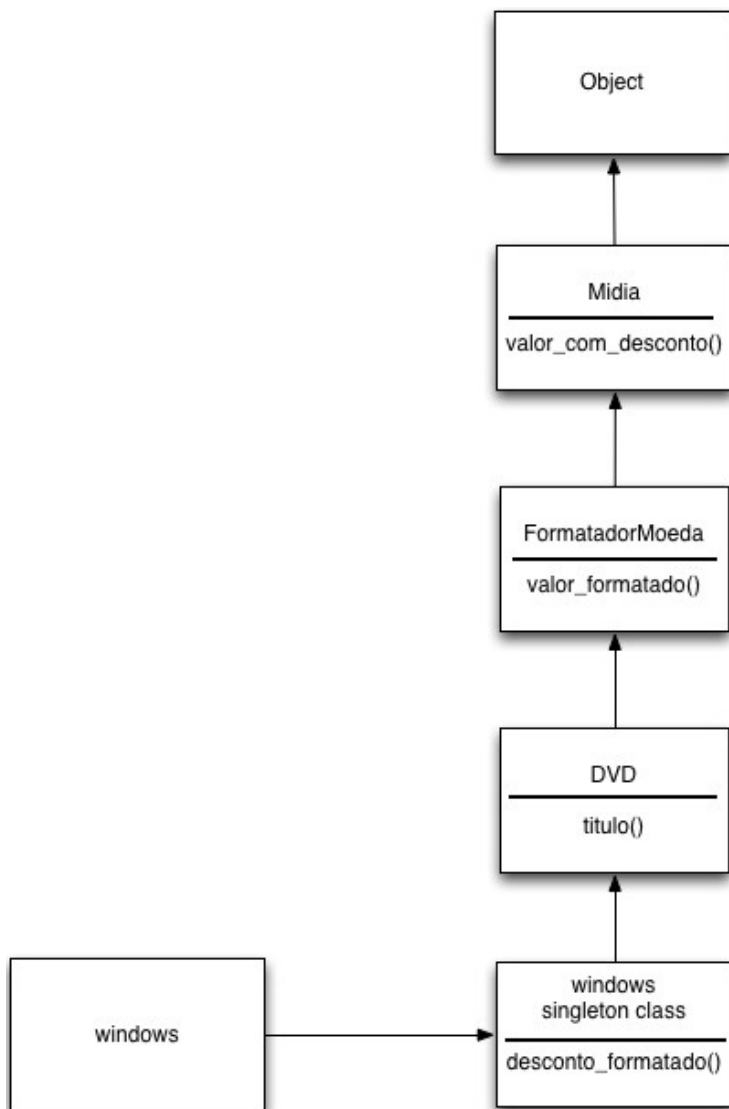


Se o método também não existir dentro dos módulos incluídos na class DVD , o interpretador vai procurá-lo na *superclass* da classe DVD , que neste caso, é a classe Midia .

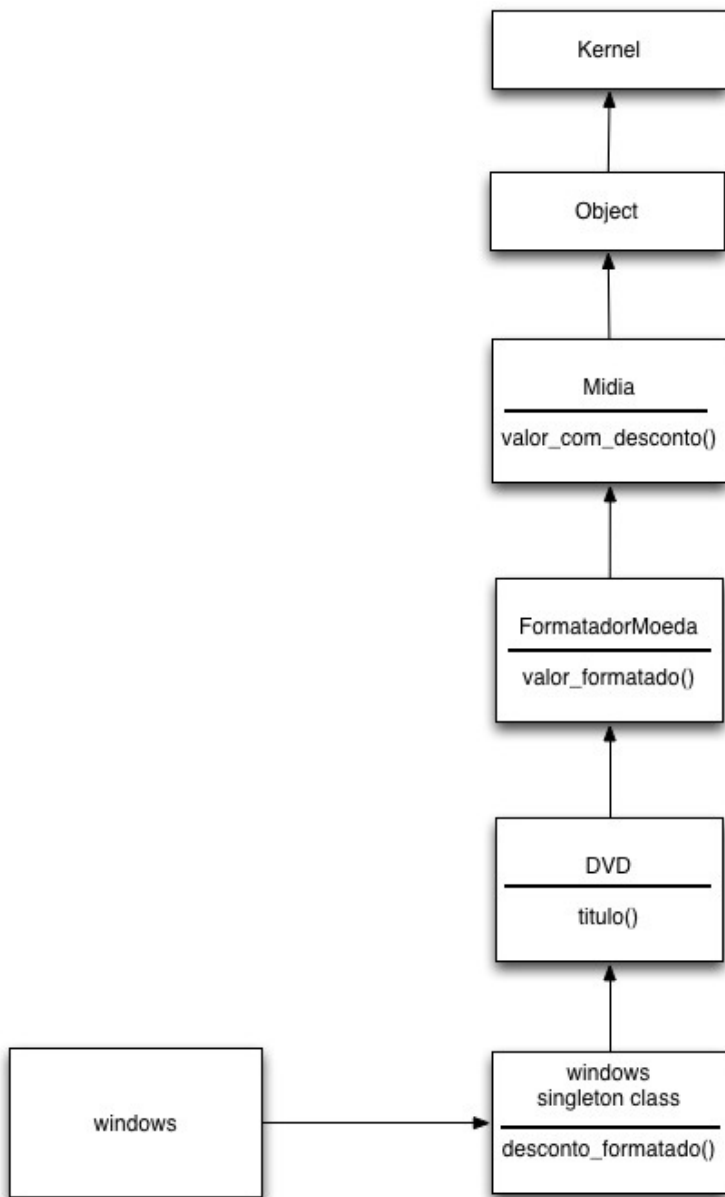


Uma vez que o método também não existir dentro da *superclass* de DVD , o interpretador vai procurá-lo então na *superclass* da classe Midia , que parece não estar definida, mas é a

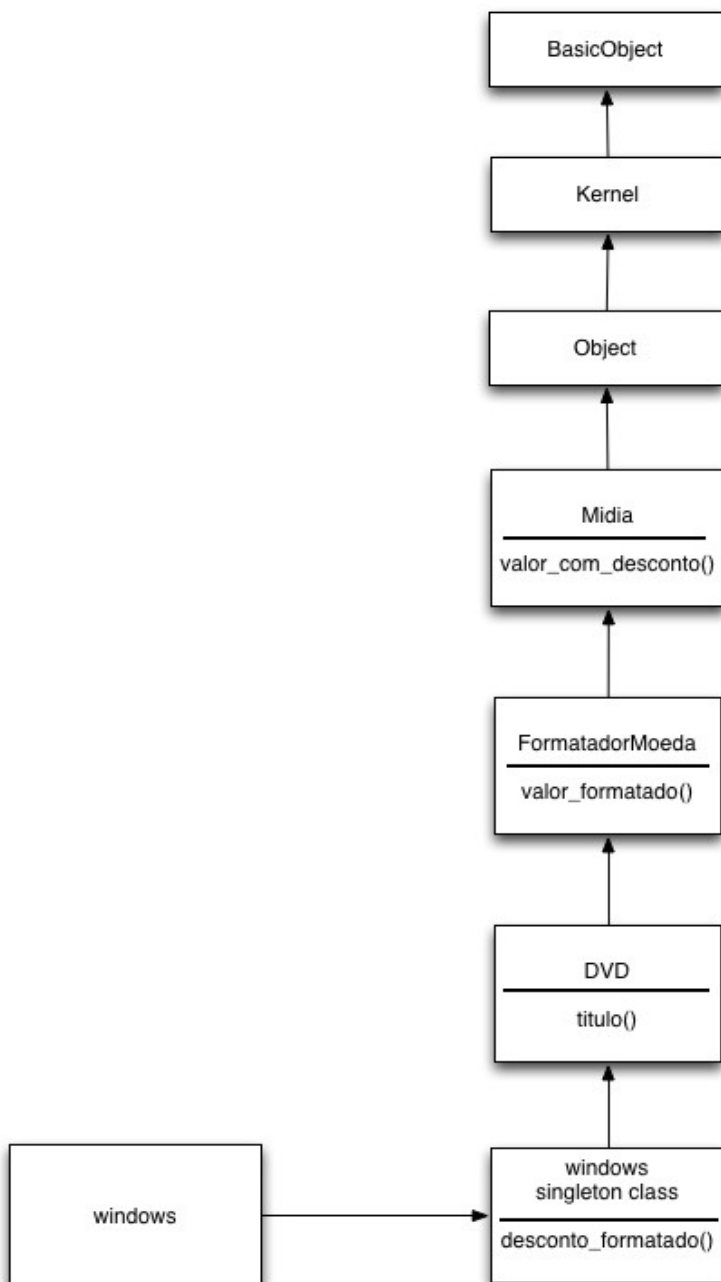
classe `Object` . Lembre-se de que todas as classes que não estendem explicitamente uma outra têm como *superclass* **sempre** a classe `Object` .



A classe `Object` inclui um módulo chamado `Kernel` , onde encontra-se, por exemplo, o método `puts` . O interpretador Ruby procurará pelo método invocado dentro deste módulo também:



A partir do Ruby 1.9, caso o método não exista também na classe `Object`, existe um outro passo a ser percorrido, a classe `BasicObject`, que é a *superclasse* da classe `Object`. A classe `BasicObject` é o topo da hierarquia de classes definidas pela linguagem Ruby, e é o último lugar onde o interpretador procurará por um método a fim de executá-lo. Caso ele não seja encontrado, a exceção `NoMethodError` será lançada.



Este processo executado no *method lookup* do Ruby é conhecido como *one step to right, then up*. Ou seja, o método é buscado na classe do objeto, caso não encontrado, o interpretador sobe na hierarquia e faz a busca até encontrá-lo.

Como saber minha hierarquia de classes?

Para descobrir toda a cadeia hierárquica que compõe uma determinada classe, basta usarmos o método `ancestors` :

```
p DVD.ancestors  
# => [DVD, FormatadorMoeda, Midia, Object, Kernel, BasicObject]
```

Repare que o `Array` retornado representa exatamente o fluxo que foi explicado.

Trabalhando com *method missing*

Vimos na seção anterior que o interpretador Ruby busca pelo método que precisa ser executado, olhando primeiro na *singleton class* do objeto, depois na classe do objeto, na superclasse da classe do objeto, assim por diante, até chegar na classe `BasicObject` . Quando o interpretador Ruby não encontra o método, ele na verdade executa a chamada de um *hook method* chamado `method_missing` .

Quando um *hook method* é invocado, o processo pela busca do método que deve ser executada é exatamente o mesmo de um outro método, e o interpretador percorre toda a hierarquia de classes. Entretanto, neste caso, o interpretador encontra a implementação do método `method_missing` definido na classe

`BasicObject` , desta maneira a busca pelo método termina neste ponto.

O comportamento padrão deste método é lançar uma exceção do tipo `NoMethodError` . Isto é, quando invocamos um método que não existe na hierarquia de classes do objeto, o método `method_missing` é executado e por esse motivo recebemos uma exceção.

O importante de entender todo esse processo é que o `method_missing` é um método como qualquer outro, de modo que podemos sobrescrevê-lo e criar um comportamento próprio para tratar casos nos quais um método não existe. O maior exemplo disso é o *ActiveRecord*, que faz o mapeamento entre o objeto e o banco de dados. Quando utilizado, o *ActiveRecord* mapeia todas as colunas de uma tabela para métodos `get` e `set` em objeto cujo tipo é o nome da própria tabela, porém, não obrigatoriamente:

```
class Revista < ActiveRecord::Base
end
```

Se a tabela `Revista` tiver uma coluna chamada `nome` , ao criarmos um objeto do tipo `Revista` , podemos obter e alterar o valor do campo:

```
revista = Revista.new
revista.nome = "MundoJ"
p revista.nome # => MundoJ
```

Estão disponíveis métodos de busca na classe `Revista` . Se quisermos, por exemplo, buscar por todas as revistas cujo nome for "MundoJ", podemos fazê-lo utilizando um método chamado `find_by_nome` :

```
mundo_j = Revista.find_by_nome "MundoJ"
```

Se tivermos uma coluna chamada `valor`, podemos fazer a busca por `valor`, utilizando um método chamado `find_by_valor`:

```
mundo_j = Revista.find_by_valor 90.8
```

Podemos fazer combinações de busca por dois campos ao mesmo tempo:

```
mundo_j = Revista.find_by_nome_and_valor "MundoJ", 90.8
```

Obviamente, todos estes métodos são criados e manipulados via metaprogramação. Mas neste caso específico, a implementação da classe `ActiveRecord::Base` reimplementa o método `method_missing`. Com esta alteração, todas as chamadas de métodos inexistentes feitas em um objeto do tipo `ActiveRecord::Base` acarretarão na execução deste novo `method_missing`, que possui uma implementação amarrada com as colunas existentes do banco de dados.

Exemplificando, ao invocarmos o método `find_by_nome` na classe `Revista`, o interpretador Ruby buscará por ele na hierarquia de classes normalmente. Como este método não existe, o *hook method* `method_missing` mais próximo da classe `Revista` será executado, ou seja, o método reimplementado na classe `ActiveRecord::Base`.

Esta nova implementação utiliza convenções e verifica primeiramente se o método que desejamos invocar segue o padrão `find_by_alguma_coluna_da_tabela`. Em caso positivo, o segundo passo é verificar se a coluna existe no banco de dados; caso exista, uma busca por este campo e pelo valor passado na

invocação do método é efetuada no banco de dados. Se a coluna não existir ou o método na estiver no padrão de nomes do ActiveRecord, a implementação definida na classe `BasicObject` é invocada pelo método `super`.

Implementando finders

Vamos criar métodos *finders* para efetuar buscas na classe `Revista` através de seus *fields* utilizando `method_missing`. O intuito é permitir que os seguintes métodos sejam invocados:

```
munJo_j = Revista.find_by_titulo "MundoJ"  
munJo_j = Revista.find_by_valor 90.8
```

Vamos utilizar o `method_missing` no módulo `ActiveFile` para suportar estes comportamentos:

```
module ActiveFile  
  module ClassMethods  
    def method_missing(name, *args, &block)  
      load_all.select do |object|  
        field = name.to_s.split("_").last  
        object.send(field) == args.first  
      end  
    end  
  
    private  
  
    def load_all  
      Dir.glob('db/revistas/*.yaml').map do |file|  
        deserialize file  
      end  
    end  
  
    def deserialize(file)  
      YAML.load File.open(file, "r")  
    end  
  end  
end
```

Podemos invocar agora um método chamado `find_by_titulo` :

```
# Retorna todas as revistas cujo título é 'MundoJ'  
revistas = Revista.find_by_titulo "MundoJ"
```

Explicando o código com mais detalhes. O primeiro passo foi redefinir o método `method_missing` , que recebe três argumentos:

1. `name` - nome do método que está sendo invocado e não foi definido em nenhuma classe da hierarquia.
2. `*args` - argumentos passados para o método que foi invocado.
3. `&block` - bloco de código passado na chamada do método inexistente, que pode ser executado com `yield` .

A implementação do método `method_missing` invoca um método `private` chamado `load_all` , que percorre todos os arquivos `.yaml` do diretório onde estão localizados os objetos serializados do tipo `Revista` . Então, ele deserializa e adiciona-os dentro de um `Array` retornado no final do método.

Com todos os objetos `Revista` em mãos, é possível filtrar os que desejamos utilizando o método `select` da API `Enumerable` . Na chamada ao método `select` , nós fizemos o `split` no nome do método que está sendo executado (no exemplo: `find_by_titulo`) pelo caractere `_` (underline), e obtemos a última ocorrência, neste caso a `String` cujo valor é `titulo` .

Com o nome do campo pelo qual desejamos efetuar o filtro em mãos, basta acessar o valor da variável `titulo` usando o método

`send` e comparando o seu resultado com o primeiro argumento passado, que corresponde ao primeiro parâmetro da chamada ao método `find_by_titulo:"MundoJ"`.

Entretanto, nosso código está frágil:

```
# NoMethodError: undefined method 'blah'
for #<Revista:0x007fd0328369e0>
  Revista.blah
```

O problema está na chamada ao método `send` que recebeu a `String` `blah` como argumento. Como os objetos `Revista` não possuem nenhum método acessor chamado `blah`, a exceção `NoMethodError` é lançada. Devemos nos precaver e programar defensivamente **sempre** que redefinirmos o método `method_missing`.

A solução neste caso é verificar se o `field` pelo qual desejamos efetuar a busca existe ou não naquele objeto:

```
module ActiveFile
  module ClassMethods
    def method_missing(name, *args, &block)
      field = name.to_s.split("_").last
      super if @fields.include? field

      load_all.select do |object|
        object.send(field) == args.first
      end
    end

    private

    def load_all
      Dir.glob('db/revistas/*.yaml').map do |file|
        deserialize file
      end
    end

    def deserialize(file)
```

```

        YAML.load File.open(file, "r")
    end
end
end

```

A verificação feita invoca o `method_missing` original (herdado da classe `BasicObject`) caso o `field` que estamos acessando não esteja registrado na variável de instância `@fields` . O comportamento feito para efetuar buscas por objetos somente é válido caso esteja sendo feita por um atributo válido; qualquer outro caso, o `method_missing` original é invocado.

Lembre-se de que é **altamente** recomendável invocar o `method_missing` original utilizando o método `super` , quando alguma situação sair do controle da reimplementação que foi feita.

8.10 UTILIZANDO EXPRESSÕES REGULARES NAS BUSCAS

A maioria dos atributos que trabalhamos até o momento são do tipo `String` . Com a classe `String` , é possível fazer diversas operações e transformações com seus mais de 100 métodos. Porém, algumas vezes é necessário buscar determinado conjunto de caracteres e substituí-los por outros, ou até mesmo criar validações para uma `String` , como verificar se ela é um e-mail válido, por exemplo. Nestas ocasiões, usamos as expressões regulares.

O que uma expressão regular pode fazer por mim

Um expressão regular consiste em um *pattern* que pode ser avaliado junto a uma `String` . Na teoria, isso significa fazer

comparações como: "o texto que está escrito aqui" contém a sequência de caracteres "escrito", por exemplo. Na prática, as expressões regulares servem para:

- Testar se uma `String` *casa* com um determinado *pattern*.
- Extrair partes de uma `String` que *casam* com um determinado *pattern*.
- Alterar uma `String` , substituindo partes que *casam* com um determinado *pattern* por outra `String` .

Criando métodos de busca utilizando expressões regulares

Existem várias maneira de criar expressões regulares em Ruby, mas vou mostrar a mais básica delas, utilizando o caractere `/` . Portanto, em Ruby, ao escrevermos `/windows/` , estamos criando uma expressão regular que usaremos para comparar com `String` s.

Essa simples expressão regular *casa*, por exemplo, com as seguintes `String` s: `"windows xp"` e `"blue windows"` . Mas não *casa* com as `String` s: `"window"` e `"Windows"` .

ESCAPANDO CARACTERES ESPECIAIS

Existem alguns caracteres especiais usados para construir expressões regulares, por exemplo o caractere `/` . Se quisermos criar uma expressão regular que possua `/` , devemos escapá-la:

```
/windows\xp/ # casa com 'windows/xp'
```

O método `find_by_titulo` poderia receber uma expressão regular e não uma `String` como argumento. Desta maneira, a busca se tornaria mais poderosa e não existiria necessidade de saber o título exato de uma `Revista` para recuperá-la. Como a implementação do método `find_by_titulo` é feita dentro do método `method_missing` , precisamos fazê-lo genérico a ponto de receber como argumento uma `String` , uma expressão regular e um `Float` (para o método `find_by_valor`).

Usaremos então o método `kind_of?` para verificar o tipo de argumento passado para saber que tipo de comparação é necessário dentro do `select` , executado com o `Array` de objetos `Revista` :

```
module ActiveFile
  module ClassMethods
    def method_missing(name, *args, &block)
      argument = args.first
      field = name.to_s.split("_").last

      super if @fields.include? field

      load_all.select do |object|
        should_select? object, field, argument
      end
    end
  end
end
```

```

    end
  end

  private

  def should_select?(object, field, argument)
    if argument.kind_of? Regexp
      object.send(field) =~ argument
    else
      object.send(field) == argument
    end
  end

  def load_all
    Dir.glob('db/revistas/*.yaml').map do |file|
      deserialize file
    end
  end

  def deserialize(file)
    YAML.load File.open(file, "r")
  end
end
end

```

O operador `=~` tenta casar a `String` como a `Regexp` passada como argumento, caso o argumento passado realmente seja uma instância de uma expressão regular. Caso a expressão regular não case com a `String`, o valor `nil` é retornado; se casar com a posição inicial (onde a expressão regular começa a casar com a `String`), é retornada.

Como `nil` equivale a `false` e qualquer outro valor equivale a `true`, caso a expressão regular case com o título do `DVD`, o objeto é retornado na chamada do método `find_by_titulo`.

```

# Retorna todas as revistas cujo título comecem com 'Mundo'
revistas = Revista.find_by_titulo /Mundo/

```

UTILIZANDO CARACTERES ESPECIAIS DENTRO DE EXPRESSÕES REGULARES

Quando as expressões regulares são criadas usando `/expressao regular/`, precisamos escapar os caracteres `/`:

```
/windows\\ # casa com 'windows/xp'
```

Se a expressão regular for bastante complexa, fica quase impossível interpretá-la visualmente. Nestes casos, utilizamos uma outra maneira de instanciar expressões regulares em que é possível adicionar caracteres `/` sem precisar escapá-las:

```
%r(windows/) # gera a expressão regular /windows\\
```

Melhorando o `method_missing`

A verificação feita na implementação do `method_missing` para saber por qual `field` desejamos efetuar a busca está muito frágil:

```
Revista.abc_titulo "MundoJ"
```

Podemos efetuar a busca usando qualquer nome de método que possua um `_` (underline) seguido do nome do `field`. Lembrando que esse comportamento é executado na seguinte linha de código:

```
field = name.to_s.split("_").last
```

Podemos tornar este código mais seguro utilizando uma expressão regular que verificará se o nome do método começa com

```

find_by_ :

module ActiveFile
  module ClassMethods
    def method_missing(name, *args, &block)
      super unless name.to_s =~ /^find_by_/

      argument = args.first
      field = name.to_s.split("_").last

      super if @fields.include? field

      load_all.select do |object|
        should_select? object, field, argument
      end
    end
  end
end

```

Agora nossa implementação de `method_missing` invoca o `method_missing` herdado através de `super` caso o nome do método não case com a expressão regular `/^find_by_/`. Agora ao executarmos, invocaremos um método `abc_titulo`:

```

# NoMethodError
Revista.abc_titulo "MundoJ"

```

O `method_missing` original é invocado.

Extraindo a variável `field` a partir da expressão regular

A implementação do `method_missing` está mais segura, porém, existem pontos que podem ser melhorados. Ainda usamos o método `split` para extrair o nome do `field` pelo qual desejamos fazer a busca. Podemos extrair este valor a partir da expressão regular que fizemos anteriormente, utilizando grupos:

```

"Windows XP" =~ /Windows (.*)/
"Windows 98"  =~ /Windows (.*)/
"Windows Vista" =~ /Windows (.*)/

```

Os grupos são delimitados por `()` (parênteses) definidos dentro da expressão regular. No exemplo anterior, o grupo criado vai capturar as String s: "98" , "XP" e "Vista" , respectivamente. Mas como podemos recuperar estes valores? Simples, basta usarmos uma variável chamada `$numero_do_grupo` :

```
"Windows XP" =~ /Windows (.*)/
p $1 # => "XP"

"Windows 98" =~ /Windows (.*)/
p $1 # => "98"

"Windows Vista" =~ /Windows (.*)/
p $1 # => "Vista"
```

Como nessa expressão regular definimos apenas 1 grupo, o acessamos utilizando `$1` . Se tivéssemos definido mais grupos, a regra seria a mesma, `$numero_do_grupo` .

Agora podemos alterar a implementação do `method_missing` para extrair o nome do `field` de um grupo da expressão regular:

```
module ActiveFile
  module ClassMethods
    def method_missing(name, *args, &block)
      super unless name.to_s =~ /^find_by(.*)/

      argument = args.first
      field = $1

      super if @fields.include? field

      load_all.select do |object|
        should_select? object, field, argument
      end
    end
  end
end
```


SUBSTITUINDO VALORES UTILIZANDO EXPRESSÕES REGULARES

Expressões regulares são tão poderosas que nos permite substituir determinada ocorrência em uma `String` por outro valor. Por exemplo:

```
mondo_java = Revista.new titulo: "Mundo Java"
p mondo_java.titulo.sub /Java/, "J" # => "Mundo J"
```

O método `sub` procura a primeira ocorrência que case com a expressão regular na `String` e substitui pelo valor do segundo argumento:

```
mondo_java = Revista.new titulo: "Mundo Java Java"
p mondo_java.titulo.sub /Java/, "J" # => "Mundo J Java"
```

Se quisermos substituir todas as ocorrência, utilizamos o método `gsub` :

```
mondo_java = Revista.new titulo: "Mundo Java Java"
p mondo_java.titulo.gsub /Java/, "J" # "Mundo J J"
```

Os método `sub` e `gsub` retornam novos objetos do tipo `String` . Caso queira alterar a `String` original, basta invocar os métodos usando o caractere `!` (exclamação):

```
mondo_java = Revista.new titulo: "Mundo Java"
mondo_java.titulo.gsub! /Java/, "J"
p mondo_java.titulo # => "Mundo J"
```

8.11 PRÓXIMOS PASSOS

Neste capítulo, vimos conceitos básicos e avançados sobre metaprogramação. Aprendemos a criar método dinamicamente

utilizando `define_method`. Vimos também o famoso e perigoso `method_missing` que serve como *fallback* para métodos que não existem no objeto receptor.

Estes são conceitos amplamente usados em bibliotecas que os desenvolvedores Ruby utilizam no dia a dia, por isso a importância de aprendê-los. São poucos os casos em que usamos recursos tão avançados.

E por falar em bibliotecas, no próximo capítulo aprenderemos como funcionam a distribuição e a utilização delas em projetos. Veremos como criar uma biblioteca, distribuí-la e como gerenciar dependências em projetos Ruby.

AS BIBLIOTECAS NO UNIVERSO RUBY

Em linguagens modernas como Ruby, é comum a utilização de bibliotecas que acelerem o desenvolvimento. Estas bibliotecas são distribuídas no universo Ruby através de **gems**, que são arquivos que terminam com a extensão `.gem`, onde ficam localizados o código-fonte da biblioteca e também informações e metadados, como versão e nome.

Neste capítulo, nós aprenderemos a instalar e gerenciar estas **gems** e também como criar e distribuir uma.

9.1 COMO MANUSEAR SUAS GEMS COM O RUBYGEMS

Rubygems é o framework padrão que utilizamos para empacotar, instalar, atualizar e remover bibliotecas escritas em Ruby dentro de um aplicativo ou outra biblioteca. Atualmente, todas as bibliotecas modernas e atualizadas estão disponíveis para instalação através do Rubygems. Com apenas um comando, todo o processo de baixar a gem na internet e instalá-la é feito automaticamente.

O próprio Rubygems disponibiliza uma ferramenta de linha de comando para gerenciar as gems. Tudo pode ser feito usando o comando `gem` que está disponível automaticamente se você estiver utilizando uma versão do Ruby superior a 1.9. Nas versões mais antigas, era necessário instalar o Rubygems manualmente.

Instalando uma gem

Os valores dos livros, DVDs e CDs da nossa loja virtual estão sendo impressos com o prefixo "R\$". Queremos agora que o valor seja impresso por extenso e, em vez de escrever um código que faz isso, usaremos uma `gem` criada por brasileiros chamada `brnumeros`, cujo objetivo é manipular números de acordo com padrões propostos apenas no Brasil. Para instalá-la, basta executar o seguinte comando no seu terminal:

```
gem install brnumeros
```

A saída do console deve ser parecida com a saída a seguir, se a instalação foi feita com sucesso:

```
Fetching: brnumeros-3.3.0.gem (100%)
Successfully installed brnumeros-3.3.0
1 gem installed
Installing ri documentation for brnumeros-3.3.0...
Installing RDoc documentation for brnumeros-3.3.0...
```

As duas últimas linhas indicam que a documentação (RDoc) da `gem` foi instalada também. Caso você não queira que a documentação seja instalada, basta passar a opção `--no-rdoc --no-ri`.

Após a instalação, você pode verificar as `gems` disponíveis através do comando `gem list`, que retorna todas as `gems` instaladas:

```
*** LOCAL GEMS ***
```

```
brnumeros (3.3.0)
```

Utilizando a gem instalada

Uma vez que a `gem` foi instalada, basta utilizarmos o método `require` para carregá-la dentro do código-fonte da nossa aplicação, assim como fizemos com bibliotecas nativas anteriormente. É necessário também executar um `require` nas `rubygems` :

```
require 'rubygems'  
require 'brnumeros'
```

Vamos então adicionar um novo método no módulo `FormatadorMoeda` chamado `valor_por_extenso` :

```
# carregando rubygems  
require 'rubygems'  
# carregando gem brnumeros  
require 'brnumeros'  
  
module FormatadorMoeda  
  module ClassMethods  
    def formata_moeda(*variaveis_e_metodos)  
      variaveis_e_metodos.each do |name|  
        define_method("#{name}_formatado") do  
          valor = respond_to?(name)  
            ? send(name)  
            : instance_variable_get("@#{name}")  
          "R$ #{valor}"  
        end  
      end  
  
      # metodo que retorna valor por extenso em reais  
      define_method("#{name}_por_extenso") do  
        valor = respond_to?(name)  
          ? send(name)  
          : instance_variable_get("@#{name}")  
        valor.por_extenso_em_reais  
      end  
    end  
  end  
end
```

```

        end
    end
end

def self.included(base)
    base.extend ClassMethods
end
end

windows = DVD.new "Windows 7 for Dummies", 98.9,
:sistemas_operacionais
windows.valor_por_extenso
# => noventa e oito reais e noventa centavos

```

O código não possui muitas novidades, apenas a utilização da gem `brnumeros` através da chamada ao método `valor_por_extenso` que pega o valor de um `Integer` ou `Float` e retorna uma `String` com o valor por extenso.

Instalando versões específicas de uma gem

Em algumas ocasiões, é necessário instalar uma determinada versão de uma gem, por exemplo, em casos de incompatibilidade de outras gems que utilizamos no nosso sistema. O comando `gem` permite que seja passado um parâmetro `--version versao_desejada`. Podemos instalar uma versão mais antiga da gem `brnumeros`:

```

gem install brnumeros --version '3.2.0'

Fetching: brnumeros-3.2.0.gem (100%)
Successfully installed brnumeros-3.2.0
1 gem installed

```

Da maneira que o parâmetro `--version` foi especificado, a versão 3.2.0 será instalada. Mas podemos ser mais flexíveis e, por exemplo, instalar qualquer versão que seja maior que a versão 3.2.0:

```
gem install brnumeros --version '> 3.2.0'
```

```
Fetching: brnumeros-3.3.0.gem (100%)
Successfully installed brnumeros-3.3.0
1 gem installed
```

Ou ainda especificar que a versão instalada deve ser a última versão de um determinado *minor*. Por exemplo, instale a última versão 3.0.x:

```
gem install brnumeros --version '~> 3.0.0'
```

```
Fetching: brnumeros-3.0.8.gem (100%)
Successfully installed brnumeros-3.0.8
1 gem installed
```

9.2 GERENCIANDO VÁRIAS VERSÕES DE UMA GEM

Como acabamos de ver, podemos instalar uma versão mais antiga da gem `brnumeros`:

```
gem install brnumeros --version '3.2.0'
```

Agora ao rodarmos o comando `gem list`, veremos as duas versões:

```
*** LOCAL GEMS ***
```

```
brnumeros (3.3.0, 3.2.0)
```

Mas qual das duas será usadas no módulo `FormatadorMoeda`? A resposta é: a última. Se você precisa utilizar a versão mais antiga da gem em um determinado projeto, basta usar o método `gem` dentro do código, especificando qual versão será carregada:

```
# carregando rubygems
require 'rubygems'
```

```
# carregando gem brnumeros versao 3.2.0
gem 'brnumeros', '3.2.0'
require 'brnumeros'

module FormatadorMoeda
  module ClassMethods
    end
  end
end
```

O método `gem` diz que a `brnumeros` deve ser carregada na versão 3.2.0. Porém, essa metodologia de gerenciar as versões das gems que desejamos utilizar pode se tornar um problema com o passar do tempo, principalmente porque no universo Ruby é muito comum termos várias dependências em um mesmo projeto, mesmo que este seja pequeno. Controlar cada versão é muito trabalhoso, por isso atualmente existem gerenciadores de dependências escritos em Ruby que facilitam nossa vida e cuidam destas preocupações para nós.

9.3 GERENCIE DEPENDÊNCIAS COM O BUNDLER

A forma mais eficaz de gerenciar dependências em projetos Ruby é através de uma gem chamada *Bundler*. Sua função é garantir que o código da sua aplicação usará as versões exatas das gems das quais ela depende. Para utilizar o *Bundler*, devemos primeiro instalá-lo:

```
gem install bundler
```

As dependências da sua aplicação devem ser declaradas dentro de um arquivo chamado `Gemfile`. Esse arquivo deve, primeiramente, conter a fonte de onde as dependências serão

baixadas:

```
source "http://rubygems.org"
```

A declaração *source* indica que as gems serão baixadas do repositório oficial do Rubygems. O próximo passo é declarar quais são as gems que são dependências do projeto:

```
source "http://rubygems.org"
```

```
gem "brnumeros", "3.3.0"
```

Na declaração *gem*, indicamos o nome e a versão que desejamos. Na declaração da versão, podemos seguir o mesmo padrão que seguimos ao instalar um gem manualmente. No *Gemfile* do nosso exemplo, a versão instalada será exatamente a versão 3.3.0 e, se quisermos instalar a última versão do *minor* 3.2.x, a declaração seria feita da seguinte maneira:

```
source "http://rubygems.org"
```

```
gem "brnumeros", "~> 3.3.0"
```

Para que as gems sejam instaladas respeitando o que foi definido no *Gemfile*, basta rodarmos o comando *bundle install* no terminal. Ele vai gerar um novo arquivo chamado *Gemfile.lock*, que especifica todas as gems obtidas para o *Gemfile* e sua respectiva versão baixada.

O *Gemfile.lock* é uma boa alternativa para congelar as versões das gems a serem utilizadas, uma vez que ao rodarmos o comando *bundle install* sobre a presença de um *Gemfile.lock*, as versões presentes nesse arquivo serão usadas para especificar as gems a serem baixadas.

Para que as versões especificadas no *Gemfile* sejam

respeitadas durante o load da aplicação, precisamos carregar uma classe do bundler:

```
require 'bundler/setup'
require 'brnumeros'

module FormatadorMoeda
  module ClassMethods
    end
end
```

Automaticamente a versão especificada no `Gemfile` será carregada para a gem `brnumeros`. Agora, em vez de executar um `irb` no terminal para testar as classes que estamos construindo, execute o comando `bundle console` que abre o `irb` com o `bundle` pré-carregado:

```
bundle console

require File.expand_path('lib/loja_virtual')

windows = DVD.new "Windows 7 for Dummies", 98.9,
  :sistemas_operacionais
windows.valor_por_extenso
```

Resolvendo conflitos com `bundle exec`

Existem `gems` que criam caminhos executáveis para utilizarmos no console, o próprio `Bundler` nos disponibiliza comandos úteis como o `bundle console`. A gem `rake`, que será explorada no próximo capítulo, cria um executável que nos permite rodar *tasks* a partir do terminal:

```
rake -T
```

O comando anterior lista todas as *tasks* definidas pela aplicação. Porém, se tivermos mais de uma versão da gem `rake` instalada no sistema e se no `Gemfile` da aplicação estiver

especificada alguma destas versões, esse comando acusará um erro:

```
rake aborted!  
You have already activated rake 10.0.3,  
but your Gemfile requires rake 10.0.2
```

Basicamente isso significa que no `Gemfile` da aplicação está especificada a versão 10.0.2 da gem `rake`, enquanto no sistema existe uma versão mais nova, a 10.0.3. Se você não puder atualizar a versão do `rake` no seu `Gemfile`, é necessário executar o comando `rake`, usando um outro comando chamado `bundle exec`:

```
bundle exec rake -T
```

O comando `bundle exec` executará todos os comandos (`rake`, `rails` etc.) respeitando a versão descrita no `Gemfile`. Neste exemplo, o comando `rake -T` será executado com a versão 10.0.2, resolvendo o conflito. No capítulo *RVM (Ruby Version Manager)*, veremos que existe uma alternativa ao uso do comando `bundle exec`.

Quem usa Bundler?

O Rails, o mais famoso framework do mundo Ruby, utiliza por padrão o *Bundler* para gerenciar as suas dependências. Qualquer aplicação Rails terá um arquivo `Gemfile` com as dependências, portanto, é essencial entender como usá-lo. Você pode ver mais detalhes da excelente documentação disponível no site: <http://gembundler.com/>.

9.4 CRIANDO E DISTRIBUINDO GEMS

Como foi dito anteriormente, uma gem nada mais é que um arquivo compactado com o código-fonte da biblioteca e um arquivo que contém os metadados, como nome, versão e outras dependências. Vamos distribuir o código da classe `ActiveFile` em uma gem.

É recomendável que uma gem tenha uma estrutura definida de diretórios. O próprio *Bundler* possui um *bootstrap* que gera estes diretórios de maneira que não é necessário criá-los manualmente. Rodando o comando `bundle gem active_file` que vai gerar a seguinte saída:

```
create active_file/Gemfile
create active_file/Rakefile
create active_file/LICENSE
create active_file/README.md
create active_file/.gitignore
create active_file/active_file.gemspec
create active_file/lib/active_file.rb
create active_file/lib/active_file/version.rb
```

Com a seguinte estrutura:

Name	Date Modified	Size	Kind
active_file.gemspec	Jan 8, 2013 12:38 AM	408 bytes	Document
Gemfile	Jan 5, 2013 12:39 AM	96 bytes	Document
Gemfile.lock	Jan 12, 2013 3:23 PM	143 bytes	Document
lib	Today 11:56 AM	--	Folder
active_file	Jan 5, 2013 12:39 AM	--	Folder
version.rb	Jan 11, 2013 12:05 AM	42 bytes	Ruby Source
active_file.rb	Feb 24, 2013 10:50 AM	2 KB	Ruby Source
tasks	Jan 8, 2013 12:29 AM	--	Folder
LICENSE	Jan 5, 2013 12:39 AM	1 KB	Document
Rakefile	Jan 11, 2013 12:04 AM	629 bytes	Document
README.md	Jan 5, 2013 12:39 AM	510 bytes	Markd...ument

Sem dúvida, o mais importante desta estrutura é a pasta `lib`. É nela que ficam localizados os arquivos com código Ruby que compõem a biblioteca. Este diretório é adicionado ao `GEM_PATH` do RubyGems para informar quais bibliotecas estão disponíveis

para uso.

O RubyGems sobrescreve o método `require` do módulo `Kernel` com uma implementação que faz uma busca pela gem que você precisa utilizando os diretórios adicionados ao `GEM_PATH`. Por isso, sempre que usamos o método `require "alguma_biblioteca"`, o RubyGems procurará por um arquivo chamado `lib/alguma_biblioteca.rb`. Por esse motivo a importância de termos um diretório `lib`.

Abrindo o arquivo `active_file/lib/active_file.rb`, temos o seguinte conteúdo:

```
require "active_file/version"

module ActiveFile
  # Your code goes here...
end
```

Somente um arquivo é carregado e ele contém a versão da gem, importante no momento da geração do arquivo `.gem` que veremos mais à frente. Também foi declarado um módulo com o nome da gem usando a notação camel case. Essa definição é muito importante. Lembre-se de que os módulos servem para criarmos *namespaces*, e com isso evitamos conflitos com outras gems utilizadas como dependências.

Dentro deste módulo, é necessário incluir o código do módulo `ActiveFile` que está dentro da aplicação, o mesmo código que fizemos no capítulo anterior. Todo o código da implementação da gem `active_file` está disponível no meu GitHub: http://github.com/lucasas/active_file. Você pode copiar o código diretamente de lá.

9.5 DISTRIBUIÇÃO DA BIBLIOTECA

Para gerar um arquivo `.gem`, precisamos de um arquivo `.gemspec`, que possui os metadados da nossa biblioteca. Como criamos nossa biblioteca utilizando o *Bundler*, um arquivo `.gemspec` foi criado automaticamente com o seguinte conteúdo:

```
require File.expand_path(' ../lib/active_file/version', __FILE__)

Gem::Specification.new do |gem|
  gem.authors      = ["Lucas Souza"]
  gem.email        = ["lucasas@gmail.com"]
  gem.description  = %q{TODO: Write a gem description}
  gem.summary      = %q{TODO: Write a gem summary}
  gem.homepage     = ""

  gem.files        = `git ls-files`.split($\ )
  gem.executables  = gem.files.grep(%r{^bin/})
                        .map{ |f| File.basename(f) }
  gem.test_files   = gem.files.grep(%r{^(test|spec|features)/})
  gem.name         = "active_file"
  gem.require_paths = ["lib"]
  gem.version      = ActiveFile::VERSION
end
```

Vamos apagar o conteúdo e reescrevê-lo apenas com as informações mais relevantes:

```
require File.expand_path(' ../lib/active_file/version', __FILE__)

Gem::Specification.new do |gem|
  gem.name          = "active_file"
  gem.version       = ActiveFile::VERSION
  gem.description   = "Just a file system database"
  gem.summary       = "Just a file system database"
  gem.author        = "Lucas Souza"
  gem.files         =
    Dir["{lib/**/*}.rb", "README.md", "Rakefile", "active_file.gemspec"]
end
```

As informações declaradas são autodescritivas, sendo a mais

importante delas a `s.files` . Esta indica quais arquivos formarão o código-fonte da gem.

Com essas informações no arquivo `.gemspec` , basta rodarmos o comando `gem build active_file.gemspec` e um arquivo `active_file-0.0.1.gem` será gerado. Para instalar a gem recém-criada, basta rodar o comando `gem install active_file-0.0.1.gem --local` . Após isso, precisamos rodar o comando `gem list active_file -d` para verificar que ela foi instalada com sucesso.

```
*** LOCAL GEMS ***
```

```
active_file (0.0.1)
  Author: Lucas Souza
  Installed at: /Users/lucas/.rvm/gems/ruby-1.9.2-p290

  Just a file system database
```

Por fim, se quisermos distribuir esta gem para desenvolvedores Ruby no mundo inteiro, basta criar uma conta no site <http://rubygems.org> e submetê-la. Para isso, basta executar o comando `gem push active_file-0.0.1.gem` e serão solicitados e-mail e senha recém-criados. Após o upload da gem, basta acessar http://rubygems.org/gems/active_file.

Precisamos agora preparar nosso projeto para que ele deixe de utilizar o módulo `ActiveFile` que está dentro do próprio projeto e passe a usar a gem `active_file` que criamos, que está no `rubygems` . Como nosso projeto já utiliza a gem `brnumbers` como dependência e usará a gem `active_file` também como dependência, podemos usar o *Bundler* para gerenciá-las:

```
source 'http://rubygems.org'
```

```
gem 'brnumeros', '3.3.0'
gem 'active_file', '0.0.1'
```

Rodando um `bundle install`, as dependências são instaladas. Precisamos também apagar o arquivo `active_file.rb` do projeto e adicionar a linha `require 'active_file'` no arquivo `revista.rb`:

```
require 'bundler/setup'
require 'active_file'

class Revista
  # implementação
end
```

Agora estamos utilizando o módulo `ActiveFile` a partir da gem.

Adicionando dependências em uma gem

Quando criamos uma gem, é comum usarmos outras para nos auxiliar. Com isso, criamos uma dependência implícita, ou seja, o projeto que depender da sua gem dependerá automaticamente da gem que você utilizou em sua criação.

Neste caso, é recomendável declarar essas dependências no arquivo `.gemspec`. Assim ao instalar a gem em outros projetos, as dependência implícitas também serão instaladas:

```
Gem::Specification.new do |gem|
  gem.name           = "active_file"
  gem.version        = ActiveFile::VERSION
  gem.description    = "Just a file system database"
  gem.summary        = "Just a file system database"
  gem.author         = "Lucas Souza"
  gem.files          =
    Dir["{lib/**/*.rb,README.md,Rakefile,active_file.gemspec}"]
```



```
# dependência do código com a gem brnumeros
gem.add_dependency "brnumeros", "~> 3.3.0"
end
```

Precisamos também carregar as dependências da gem no arquivo `active_file.rb`:

```
# coding utf-8

# carregando dependências
require "brnumeros"
require "FileUtils"
require "active_file/version"

module ActiveFile
  # implementação
end
```

9.6 PRÓXIMOS PASSOS

Neste capítulo, aprendemos a criar, gerenciar e distribuir dependências no formato gem. Isso é essencial para trabalhar com qualquer projeto Ruby, afinal, é muito comum usarmos bibliotecas prontas para fazer parte do trabalho necessário.

Gerenciar estas dependências também é muito mais simples utilizando a gem *Bundler* que, através dos arquivos `Gemfile` e `Gemfile.lock`, controla quais versões e quais gems utilizamos em um projeto.

No próximo capítulo, veremos como criar `tasks` (pequenos scripts) em aplicações Ruby utilizando a gem `rake`.

CRIANDO TASKS USANDO RAKE

Se você já leu tutoriais ou investigou o código de algum projeto escrito em Ruby, provavelmente já ouviu falar de uma gem chamada `rake`. Suas primeiras versões tinham em mente criar um ferramenta de build similar ao famoso *Make*. Porém, a gem `rake` possui muitas ferramentas que a tornam uma poderosa biblioteca de automação.

O primeiro passo para usá-la é instalar executando a gem `install rake` no seu terminal, ou então adicionando-a no `Gemfile` do projeto e executando em seguida o comando `bundle install`. Após a instalação, teremos disponível o comando `rake`, por onde executaremos os scripts de automação que serão criados.

São muitos os passos que podem ser automatizados dentro de um projeto, desde rodar testes automaticamente, até realizar o *build* de um pacote no ambiente de produção.

Nosso aplicativo precisa de um script que "limpe" a base de dados. Resumidamente, precisamos apagar todo o conteúdo da pasta `db/revistas`. Isso pode ser feito facilmente com o seguinte

código Ruby:

```
require 'fileutils'  
FileUtils.rm Dir['db/revistas/*.yaml']
```

O código é bastante simples, mas se toda a vez que precisarmos apagar o banco de dados for necessário repetir este script, estamos bastante sujeitos a erros. Seria bem mais simples executar algo como `rake db:clean` no terminal, certo?

Isso é possível isolando o script anterior em uma `rake task`. Um pequeno script que será executado pelo comando `rake`, que vai procurar por estes scripts dentro de um arquivo chamado `Rakefile`. A gem `active_file` que criamos no capítulo anterior possui um arquivo chamado `Rakefile` com o seguinte conteúdo:

```
#!/usr/bin/env rake  
require "bundler/gem_tasks"
```

Como criamos a gem utilizando o *bundler*, automaticamente todas as suas `rake tasks` estão disponíveis para uso. Podemos remover esta linha e criar a nossa própria `task` chamada `db:clear`:

```
#!/usr/bin/env rake  
  
require 'fileutils'  
  
namespace :db do  
  task :clear do  
    FileUtils.rm Dir['db/revistas/*.yaml']  
  end  
end
```

Podemos agora rodar o comando `rake -T` que retorna uma lista com todas as `tasks` encontradas dentro do arquivo

`Rakefile` do diretório onde rodamos o comando. Como a `rake task` não possui uma descrição, ela não será retornada na lista de `tasks` disponíveis. Resolvemos este problema adicionando uma descrição:

```
#!/usr/bin/env rake

require 'fileutils'

desc "Limpa todas as revistas da pasta db/revistas"
namespace :db do
  task :clear do
    FileUtils.rm Dir['db/revistas/*.yaml']
  end
end
```

Se rodarmos novamente o comando `rake -T`, obteremos o seguinte resultado:

```
rake db:clear # Limpa todas as revistas da pasta db/revistas
```

Para testarmos a `rake task` que acabamos de criar, basta executar no terminal o comando `rake db:clear`. Porém, não faz sentido tentarmos limpar o banco de dados a partir da pasta da gem `active_file`. Precisamos expor as `rake tasks` criadas para que elas possam ser usadas dentro do projeto `loja virtual`, ou de outros projetos onde forem utilizadas.

O primeiro passo consiste em extrair as `tasks` para arquivos separados, que ficam em uma pasta chamada `lib/tasks`. Fazemos isso porque as `tasks` adicionadas dentro do `Rakefile` são disponibilizadas apenas se executarmos o comando `rake` dentro da pasta onde ele se encontra. Vamos então extrair a `task` para um arquivo chamado `db.rake`:

```
# lib/tasks/db.rake
require 'fileutils'
```

```

namespace :db do
  desc "Limpa todas as revistas da pasta db/revistas"
  task :clear do
    FileUtils.rm Dir['db/revistas/*.yaml']
  end
end

```

Agora torna-se necessário que as tasks criadas na pasta `lib/tasks` sejam copiadas para o arquivo `.gem` :

```

require File.expand_path('../lib/active_file/version', __FILE__)

Gem::Specification.new do |gem|
  gem.name          = "active_file"
  gem.version       = ActiveFile::VERSION
  gem.description   = "Just a file system database"
  gem.summary       = "Just a file system database"
  gem.author        = "Lucas Souza"
  gem.files         = Dir["{lib/**/*}.rb,lib/tasks/*.rake,
                        README.md,Rakefile,active_file.gemspec"]
end

```

Porém, ter os arquivos onde as tasks foram definidas dentro da gem não é suficiente para utilizá-las dentro de projetos externos. Existe a necessidade de carregar a classe `active_file` e as `rake` tasks existentes na pasta `lib/tasks`. Estes arquivos podem ser carregados usando o método `import` do `rake`, que carrega arquivos com a extensão `.rake` :

```

# coding utf-8

require "fileutils"
require "active_file/version"

require 'rake'
import File.expand_path("../tasks/db.rake", __FILE__)

module ActiveFile
  # implementação
end

```

A linha `File.expand_path("../tasks/db.rake", __FILE__)` retorna o caminho absoluto para o arquivo `db.rake` que contém as `rake tasks` que precisam ser importadas. Também foi necessário carregar o `rake`, pois o método `import` faz parte de sua API.

Um ponto importante que deve ser ressaltado é que o conteúdo de um arquivo `.rake` é código Ruby, e o comando `task` é um método Ruby que também faz parte da API pública do Rake. Isso permite que você implemente uma `rake task` utilizando todas as APIs Ruby que desejar.

Voltando ao código, para que a implementação possa ser testada, basta gerar uma nova versão da gem `active_file` e atualizá-la dentro do projeto da loja virtual. Ao executar o comando `rake -T` dentro do projeto, o seguinte resultado será exibido:

```
rake db:clear # Limpa todas as revistas da pasta db/revistas
```

Isso comprova que as `tasks` que foram definidas dentro da gem `active_file` podem ser aproveitadas dentro de projetos que as possuem como dependência.

10.1 PARÂMETROS NA RAKE TASK

É extremamente importante receber parâmetros na invocação de métodos, a fim de flexibilizá-los. Com as `rake tasks` não é diferente, é muito comum a necessidade de flexibilizá-las para evitar repetição de código. Por exemplo, se for necessário excluir o banco de dados referente a classe `DVD`, não faríamos algo como o código:

```
# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todas as revistas da pasta db/revistas"
  task :clear do
    FileUtils.rm Dir['db/revistas/*.yaml']
  end

  desc "Limpa todas os dvds da pasta db/dvds"
  task :clear_dvds do
    FileUtils.rm Dir['db/dvds/*.yaml']
  end
end
```

Existe muita repetição no código anterior: qualquer alteração na API `FileUtils` acarretaria em mudanças em todas as *tasks* que excluem dados das pastas. Além disso, a cada nova mídia que tivermos no sistema, será necessário criar uma nova *task* para removê-las. Ao olhar com cuidado o código, é possível perceber que a única diferença entre as *tasks* é apenas o nome da pasta em que se encontram os arquivos que desejamos excluir.

O mais óbvio neste caso é receber o nome da pasta que desejamos excluir os arquivos `*.yaml`, assim flexibiliza-se uma única *task* que será responsável por apagar os arquivos de qualquer mídia existente ou nova dentro do sistema. Vamos então receber o parâmetro `folder` na rake task `db:clear`:

```
# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todos os arquivos de uma mídia"
  task :clear, [:folder] do |task, args|
    FileUtils.rm Dir["db/#{args.folder}/*.yaml"]
  end
end
```

Todos os parâmetros que precisam ser recebidos na chamada da `rake task` devem ser adicionados logo após seu nome em forma de um `Array`. Ao executar o comando `rake -T`, o `rake` informa que a `task db:clear` precisa ser invocada com um parâmetro chamado `folder`:

```
rake db:clean[folder] # Limpa todas os arquivos de uma midia
```

Outra alteração feita no código são os dois argumentos recebidos no bloco. O primeiro parâmetro `task` é um objeto do tipo `Rake<::Task` que guarda as informações dela; o segundo argumento `args` contém os valores passados na chamada da `task` em forma de *open-struct*, ou seja, para utilizar o parâmetro `folder` declarados, deve-se fazê-lo utilizando `args.folder`, como de fato foi feito.

Para invocar a `rake task`, basta passar o argumento entre `[]`:

```
rake db:clear['revistas']
rake db:clear['dvds']
```

Múltiplos parâmetros

Se for necessário receber vários argumentos na `rake task`, basta declará-los no `Array` de parâmetros definidos após o nome da `task`:

```
# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todas os arquivos de uma midia"
  task :clear, [:folder, :file_extension] do |task, args|
    FileUtils.rm Dir["db/#{args.folder}/*.{args.file_extension}"]
  end
end
```


Para invocar a `rake task`, basta informar os dois parâmetros separados por vírgula:

```
rake db:clear['revistas','yaml']
```

Porém, existe um detalhe importante na chamada da `rake task`. Repare que não colocamos o caractere de espaço na chamada da `task`, porque a chamada deve ser feita em um único comando, juntando o seu nome e argumentos. Se for necessário invocá-la passando nome e argumentos utilizando espaços, o argumento para o comando `rake` deve ser colocado entre aspas. Algo como o comando a seguir:

```
rake "db:clear['revistas', 'yaml']"
```

10.2 TASKS COM PRÉ-REQUISITOS

Podemos criar `tasks` que são utilizadas com pré-requisitos de outras. Uma `task` muito comum quando usamos *Active Record* é a `db:seed`, que popula o banco de dados. Vamos criar uma `task` similar ao `db:seed`:

```
# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todas os arquivos de uma midia"
  task :clear, [:folder] do |task, args|
    FileUtils.rm Dir["db/#{args.folder}/*.yaml"]
  end

  desc "Popula com os dados do arquivo db/folder/seeds.rb"
  task :seed, [:folder] do
    seed_file =
      File.expand_path "db/#{args.folder}/seeds.rb"
    load(seed_file) if File.exist?(seed_file)
  end
end
```

Ela procura por um arquivo `db/seeds.rb` e carrega-o caso ele exista. O arquivo `seeds.rb` contém o código Ruby que popula o banco com dados iniciais, por exemplo:

```
require "active_file"
require "revista"

Revista.new(titulo: "Veja", valor: 10.90).save
Revista.new(titulo: "Época", valor: 12.90).save
```

Ao executar a `rake task` pelo comando `db:seed`, dois objetos do tipo `Revista` serão criados.

Outra `rake task` bastante útil que existe no `Active Record` é a `db:reseed`, que possui o mesmo comportamento da `db:seed`, porém, remove os dados existentes no banco antes de populá-lo novamente. Uma `task` como essa pode ser implementada facilmente na gem `active_file`:

```
# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todos os arquivos de uma mídia"
  task :clear, [:folder] do |task, args|
    FileUtils.rm Dir["db/#{args.folder}/*.yaml"]
  end

  desc "Popula com os dados no arquivo db/folder/seeds.rb"
  task :seed, [:folder] do
    seed_file =
      File.expand_path "db/#{args.folder}/seeds.rb"
    load(seed_file) if File.exist?(seed_file)
  end

  desc "Popula com os dados no arquivo db/folder/seeds.rb,
  apagando os existentes"
  task :reseed, [:folder] do
    FileUtils.rm Dir["db/#{args.folder}/*.yaml"]
  end
end
```

```

        seed_file =
            File.expand_path "db/#{args.folder}/seeds.rb"
        load(seed_file) if File.exist?(seed_file)
    end
end

```

Repare que o código da `task reseed` é uma cópia das `tasks clear` e `seed` na ordem. Como o comportamento que desejamos executar foi feito em outras `tasks`, podemos reutilizá-las:

```

# lib/tasks/db.rake
require 'fileutils'

namespace :db do
  desc "Limpa todas os arquivos de uma midia"
  task :clear, [:folder] do |task, args|
    FileUtils.rm Dir["db/#{args.folder}/*.yaml"]
  end

  desc "Popula com os dados no arquivo db/folder/seeds.rb"
  task :seed, [:folder] do
    seed_file =
      File.expand_path "db/#{args.folder}/seeds.rb"
    load(seed_file) if File.exist?(seed_file)
  end

  desc "Popula com os dados no arquivo db/folder/seeds.rb,
        apagando os existentes"
  task :reseed, [:folder] => ["db:clear", "db:seed"] do
  end
end

```

Na definição da `task db:reseed`, declaramos como pré-requisito a execução das `db:clear` e `db:seed`, declarando-as como um `Array` que representa o valor do `Hash` cuja chave são os parâmetros e nome da `task`.

10.3 PRÓXIMOS PASSOS

O `rake` é uma ferramenta essencial quando existe a

necessidade de automatizar tarefas na linguagem Ruby. A grande maioria dos projetos escritos em Ruby e, principalmente o Rails (o mais famoso deles), utilizam `rake`. Portanto, explore ainda mais a sua documentação e tente automatizar as tarefas do seu projeto sando-o. Outros desenvolvedores que chegarão ao projeto e conhecerem Ruby não terão dificuldades para lidar com esses scripts.

No próximo capítulo, aprenderemos a lidar com várias versões do Ruby na mesma máquina e como separar gems em lugares diferentes, organizando melhor seu ambiente e evitando conflitos, utilizando RVM.

RVM (RUBY VERSION MANAGER)

Ruby Version Manager, mais conhecido como RVM, permite que várias versões do Ruby sejam instaladas em uma máquina, dentre outras coisas. Isso é bastante útil principalmente nas máquinas dos desenvolvedores, que muitas vezes precisam dar manutenção em projetos que utilizam versões diferentes do Ruby.

Neste capítulo, você vai aprender também que o RVM possui mais utilidades do que apenas trocar entre versões do Ruby.

11.1 INSTALAÇÃO

O primeiro passo para utilizar o RVM é instalá-lo em sua máquina. Se você utiliza MacOS ou Linux, basta ter o `curl` e o `make` instalados. Em seguida, execute o seguinte comando:

```
bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

Executando o comando anterior, o RVM será instalado no diretório `home` do seu usuário. Como o RVM é utilizado via linha de comando, é necessário que ele seja carregado pelo arquivo `.bash_profile` ou `.bashrc`. Abra o arquivo e adicione a seguinte linha:

```
[[ -s "$HOME/.rvm/scripts/rvm" ]] && source  
"$HOME/.rvm/scripts/rvm"
```

Ou execute a linha a seguir no terminal:

```
echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] &&  
source "$HOME/.rvm/scripts/rvm"' >> .bash_profile
```

Basta agora recarregar o `.bash_profile` para que o RVM esteja disponível na linha de comando.

```
source ~/.bash_profile
```

Lembre-se de que, uma vez que ele estiver configurado dentro do `.bashrc`, sempre que você se logar, ele será carregado automaticamente.

Verifique que o RVM foi instalado corretamente, executando o comando: `rvm notes`. Ele mostrará algumas informações sobre o RVM e seu sistema operacional.

INSTALAÇÃO NO WINDOWS

Se você utiliza Windows, também pode instalar o RVM em sua máquina. Basta seguir esse post: <http://blog.developwithpassion.com/2012/03/30/installing-rvm-with-cygwin-on-windows>. Ele explica em detalhes como instalar e configurar o RVM em sua máquina.

Dependendo do seu sistema operacional, é necessário a instalação de algumas dependências. Para visualizá-las, basta executar o comando `rvm requirements`:

Notes for Mac OS X 10.8.2, Xcode 4.5.2.

For MacRuby: Install LLVM first.

For JRuby: Install the JDK.

See <http://developer.apple.com/java/download/>

For IronRuby: Install Mono >= 2.6

For Ruby 2.4.1: Install libksba # If using Homebrew,
'brew install libksba'

Em geral, ao executar este comando, você visualizará o que precisa para instalar cada uma das versões do Ruby. Executar este comando com frequência garantirá que sua instalação continue sempre funcionando.

11.2 INSTALANDO DIFERENTES RUBIES

A expressão *rubies* é muito comum no mundo RVM, e que na verdade significa ter várias versões do Ruby instaladas. Mas como instala-se uma versão do Ruby utilizando RVM? Vamos começar instalando a versão do Ruby MRI 2.4.1:

```
rvm install 2.4.1
```

Para utilizar a versão que foi instalada usando RVM, basta executar:

```
rvm use 2.4.1
```

Agora, ao executar `ruby -v`, a versão `2.4.1-p111` será exibida:

```
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

É importante ressaltar que, por *default*, a versão do Ruby em

sua máquina é a versão instalada diretamente no sistema operacional. Se você quiser que a versão *default* do sistema seja uma versão fornecida pelo RVM, basta executar:

```
rvm --default 2.4.1
```

Ao executar `ruby -v`, a versão 2.4.1 será exibida como versão *default* da sua máquina.

Para instalar outros interpretadores Ruby, basta seguir os mesmos passos descritos anteriormente. As versões suportadas pelo RVM podem ser vistas em: <https://rvm.io/interpreters/>.

11.3 ORGANIZE SUAS GEMS UTILIZANDO GEMSETS

O RVM cria compartimentos independentes para cada versão do Ruby instalada, ou seja, o interpretador Ruby, as gems e o IRB são todos separados do sistema e uns dos outros. O que pode acontecer é que dois projetos que utilizam a mesma versão do Ruby precisem de versões diferentes de uma determinada gem.

Como as gems são separadas apenas por instalações do Ruby, o espaço de gems teria duas versões da mesma gem. Isso poderia causar conflitos em sua utilização e a obrigação de utilizar o comando `bundle exec` para executar o projeto com as versões corretas de cada gem.

O RVM permite que, dentro do espaço de gems de cada instalação do Ruby, sejam criados espaços separados para as gems de cada projeto, evitando conflitos de versões e mantendo seu sistema mais organizado. Estes espaços são conhecidos como

gemsets (conjunto de *gems*), e também possui integração com o comando `bundle install` que instalará as *gems* diretamente no *gemset* configurado.

Crie *gemsets*

Para criar um novo *gemset*, basta executar:

```
rvm gemset create loja_virtual
# => 'loja_virtual' gemset created
(/home/lucas/.rvm/gems/ruby-2.4.1-p111@loja_virtual).
```

Para utilizar o *gemset* criado, basta executar:

```
rvm use 2.4.1@loja_virtual
# => Using /Users/lucas/.rvm/gems/ruby-2.4.1-p111
with gemset loja_virtual
```

Na raiz do projeto, vamos rodar o comando `bundle install` e, após a sua execução, podemos verificar as *gems* instaladas no *gemset* executando o comando que já vimos anteriormente `gem list`:

```
*** LOCAL GEMS ***

active_file (0.0.5)
brnumeros (3.3.0)
```

Para comprovar que as *gems* estão isoladas no *gemset*, vamos criar um outro novo e instalar o `rails` nela:

```
rvm gemset create novo_gemset
rvm use 2.4.1@novo_gemset
gem install rails
```

Executando o comando `gem list`, serão listadas apenas o `rails` e suas dependências:

*** LOCAL GEMS ***

```
actionmailer (3.2.11)
actionpack (3.2.11)
activemodel (3.2.11)
activerecord (3.2.11)
activeresource (3.2.11)
activesupport (3.2.11)
...
```

As gems existentes no *gemset* `loja_virtual` não são listadas, pois estão isoladas.

REMOVENDO GEMS DO SISTEMA

O RVM possui um *gemset* chamado `global`, em que as gems instaladas antes da utilização do RVM estão:

```
rvm use 2.4.1@global
# => Using /Users/lucas/.rvm/gems/ruby-2.4.1-p111
with gemset global
```

```
gem list
*** LOCAL GEMS ***
```

```
bundler (1.2.3)
rake (10.0.3, 10.0.2)
rubygems-bundler (1.1.0)
rvm (1.11.3.5)
```

Já que estamos organizando as gems em *gemsets*, devemos remover as gems globais. Basta executar o comando `rvm gemset empty` que limpará o *gemset* que você estiver utilizando.

O *gemset* `global` é bastante útil para adicionar automaticamente gems que serão usadas por todas as novas instalações ou novos *gemsets* criados. As gems `rake` e `bundler` são bons exemplos de dependências que estarão presentes na grande maioria *gemsets* criados.

11.4 TROQUE AUTOMATICAMENTE DE GEMSETS COM .RVMRC

Ao utilizar mais de um *gemset*, é necessário lembrar de trocá-lo de acordo com o projeto que estamos trabalhando. Como

programadores, é ruim termos de lembrar de executar o comando `rvm use 2.4.1@meu_gemset` a todo momento. Em geral, preferimos que estas coisas sejam feitas de maneira automática.

O arquivo `.rvmrc` permite que estas trocas sejam feitas de maneira automática. Ao adicionar um arquivo chamado `.rvmrc` dentro do projeto, toda vez que entrarmos na pasta do projeto, o RVM vai procurar por esse arquivo e executar o comando que se encontra dentro dele.

O arquivo `.rvmrc` terá, geralmente, a linha corresponde a troca para o *gemset* do projeto. Por exemplo, o arquivo `.rvmrc` do projeto `loja_virtual` terá o seguinte conteúdo:

```
rvm use 2.4.1@loja_virtual
```

Porém, o código anterior, ao ser executado pela primeira vez, nos alertará que não podemos usar um *gemset* que não foi criado:

```
Gemset 'loja_virtual' does not exist,  
  'rvm gemset create loja_virtual'  
first, or append '--create'.
```

Como foi alertado pelo RVM, podemos apenas adicionar o `--create` dentro do `.rvmrc` :

```
rvm use 2.4.1@loja_virtual --create
```

CRIANDO GEMSET E RVMRC EM UMA LINHA

É possível criar um novo *gemset* e seu respectivo `.rvmrc` executando a seguinte linha:

```
rvm --create --rvmrc 2.4.1@seu_gemset
```

11.5 PRÓXIMOS PASSOS

O objetivo deste capítulo foi descrever como funcionam as principais funcionalidades do RVM. Você pode consultar a documentação do RVM no site: <https://rvm.io>. Lá são listadas todas as suas funcionalidades.

No próximo capítulo, veremos as novas *features* que estarão disponíveis na nova versão da linguagem Ruby. Faremos alguns testes e exploraremos seus novos conceitos, que tornarão a vida dos desenvolvedores Ruby um pouco mais fácil do que já é.

RUBY 2.0

A versão final foi lançada no dia 24 de fevereiro, exatamente no vigésimo aniversário de sua primeira versão. A ideia deste capítulo é falar sobre as principais mudanças que ocorrerão na linguagem que você aprendeu durante os capítulos anteriores.

Uma lista completa com as mudanças introduzidas na versão 2.0 pode ser vista em: https://github.com/ruby/ruby/blob/ruby_2_0_0/NEWS. Aqui nos ateremos às principais:

- Refinements
- Named parameters
- Module prepend
- Enumerable lazy
- Encoding UTF-8

12.1 EVITANDO MONKEY PATCHES COM REFINEMENTS

Você aprendeu anteriormente que é possível *abrir* uma classe Ruby e adicionar novos métodos dinamicamente. Por exemplo, podemos abrir a classe `String` e adicionar um novo método chamado `bang` :

```
class String
  def bang
    "#{self}!"
  end
end
```

O código anterior faz com que todos os objetos do tipo `String` tenham um método chamado `bang` :

```
"ruby".bang # => "ruby!"
"rails".bang # => "rails!"
```

Ruby é uma linguagem na qual todas as classes são mutáveis. Essa habilidade de alterar classes em *runtime* é utilizada em diversas bibliotecas. Um exemplo é o `ActiveSupport` , um pedaço do Rails que abre várias classes do core do Ruby para adicionar métodos utilitários que visam facilitar o desenvolvimento. Essa técnica é conhecida na comunidade Ruby como *monkey patch*.

Porém, abrir uma classe e adicionar métodos arbitrários é uma técnica questionável. Apesar de parecer legal, pode nos trazer grandes transtornos. O problema principal desta abordagem é que, assim como nós abrimos a classe `String` e adicionamos um método chamado `bang` , qualquer outra biblioteca Ruby que usamos pode fazer exatamente a mesma coisa. O que aconteceria neste caso depende muito da ordem de carregamento, o que causaria também comportamentos inesperados na execução da aplicação.

A mudança feita na classe `String` é global e trabalhar com escopos globais sempre é ruim, por ter de controlar cada lugar que pode, eventualmente, alterar aquele escopo.

Pensando nestes problemas recorrentes, oriundos das versões atuais do Ruby, foi pensado uma funcionalidade chamada

refinements, que visa alterar o escopo de uma classe apenas em contexto local. Ou seja, você escolhe o momento exato que deseja que a classe `String` tenha o método `bang`. Isso evita os possíveis problemas descritos anteriormente.

Vamos ver como essa nova *feature* da linguagem Ruby funciona. Existem dois métodos importantes que temos de conhecer quando decidimos utilizar os *refinements*. O primeiro é o método `refine` do módulo `Kernel`, que permite que o *monkey patch* seja feito em um escopo local. Na verdade, ele funciona como se fosse um container de métodos adicionais que podem ser usados em um futuro próximo, dentro de um escopo que não seja global.

O segundo é o método `using`, que permite que os *refinements* sejam utilizados em uma classe ou método, ou seja, permite que os métodos "adicionais" da classe `String` sejam injetados no escopo onde você o invocou. Os métodos serão adicionados apenas no escopo em que você efetuou a chamada; para todos os outros, o método `bang` simplesmente não existirá.

Para implementar *refinements*, o primeiro passo é abrir a classe `String` e adicionar o método `bang` utilizando o método `Kernel#refine`:

```
module StringBang
  refine String do
    def bang
      "#{self}!"
    end
  end
end
```

Agora, se tentarmos invocar o método `bang` em um objeto `String`, será retornada uma exceção dizendo que o método não

existe:

```
"ruby".bang  
# => NoMethodError: undefined method 'bang' for "":String
```

Isso aconteceu porque a classe `String` foi alterada em um escopo local. Para utilizar o *refinement* que foi feito, devemos utilizar o método `using`, no objeto `main`:

```
using StringBang  
"ruby".bang # => "ruby!"  
  
class DVD  
  # outros metodos aqui  
  def titulo  
    @titulo.bang  
  end  
end  
  
dvd = DVD.new "Chitãozinho e Xororo"  
dvd.titulo # => "Chitãozinho e Xororo!"
```

Essa é uma importante *feature* introduzida na versão 2.0 para evitar vários problemas que os *monkey patches* globais podem causar. A dica para você que utiliza *monkey patches* é evitá-los ao máximo. Uma boa alternativa é criar módulos que definem os métodos que você deseja incluir, e utilizar o `extend` para incluí-los apenas nos objetos que você deseja:

```
module Banger  
  def bang  
    "#{self}!"  
  end  
end  
  
"ruby".extend(Banger).bang # => "ruby!"
```

No código anterior, nós também estamos adicionando o método desejado em um escopo não global. Porém, adicionamos

apenas nos objetos do tipo `String` que desejamos. Apesar de ser bastante chato invocar `extend(Bang)` em vários lugares, é bem mais garantido que não teremos problemas futuros.

12.2 NAMED PARAMETERS

Lançado com o nome *Keywords Arguments*, esta nova *feature* da linguagem Ruby já existe em outras linguagens há bastante tempo e foi finalmente introduzida na nova versão.

Para você que não conhece *named parameters*, vamos entender o problema que este conceito resolve. Vamos supor que você tem um método chamado `setup` que precisa receber algumas informações: `timeout` e `name`. Implementamos o método com o código a seguir:

```
def setup(timeout, name)
  p timeout * 1000, name
end
```

Ao invocarmos o método `setup`, devemos passar os argumentos de acordo com a posição que eles foram declarados:

```
setup 60, 'correios'
```

Se invocarmos o método trocando a posição dos parâmetros, podemos ter surpresas:

```
setup 'correios', 60
# => correioscorreioscorreios...
```

A `String` `'correios'` será impressa 1.000 vezes na saída do console. Ao utilizarmos *named parameters*, invocamos o método passando não apenas os valores dos parâmetros, mas também qual argumento receberá aquele valor. Faremos algo como:

```
setup timeout: 60, name: 'correios'  
# => 60000, 'correios'
```

Porém, para este código funcionar, devemos alterar a implementação do método para suportar *named parameters*:

```
def setup(timeout: 0, name: '')  
  p timeout, name  
end
```

É necessário declarar um valor *default* para que os argumentos possam ser preenchidos corretamente na chamada do método usando *named parameters*. Essa nova implementação do método `setup` nos permite invocar o método das seguintes maneiras:

```
# passando apenas o atributo timeout  
setup timeout: 20  
# => 20, ''  
  
# passando apenas o atributo name  
setup name: 'sedex'  
# => 0, 'sedex'  
  
# passando nenhum deles  
setup  
# => 0, ''  
  
# passando os atributos em ordem inversa  
setup name: 'sedex', timeout: 20  
# => 20, 'sedex'
```

O mais interessante é que não precisamos passar os argumentos na ordem correta, basta chamar o método passando o valor de cada argumento através de seu nome.

Vamos alterar um pouco o método `setup`, adicionando um atributo que não terá a característica de um *named parameter*:

```
def setup(value, timeout: 0, name: '')  
  p value, timeout, name
```

end

Com essa mudança, todas as chamadas que foram feitas no exemplo anterior não funcionarão, pois o atributo `value` está declarado como atributo obrigatório e deve ser **sempre** o primeiro atributo a ser passado na chamada do método. Sendo assim, as chamadas anteriores devem ser transformadas:

```
# passando apenas o atributo timeout
setup 79.9, timeout: 20
# => 79.9, 20, ''

# passando apenas o atributo name
setup 79.9, name: 'sedex'
# => 79.9, 0, 'sedex'

# passando nenhum deles
setup 79.9
# => 79.9, 0, ''

# passando os atributos em ordem inversa
setup 79.9, name: 'sedex', timeout: 20
# => 79.9, 20, 'sedex'

# passando o parâmetro valor no final
setup name: 'sedex', timeout: 20, 79.9
# => SyntaxError: (irb):19: syntax error
```

Isso deixa um pouco claro que os *named parameters* na verdade são apenas atributos com valores *default*. Vamos explorar um pouco mais os *named parameters* tentando declarar um novo atributo após o último *named parameter*:

```
def setup(value, timeout: 0, name: '', url)
  p value, timeout, name, url
end

# => SyntaxError: (irb):23: syntax error, unexpected tIDENTIFIER
```

O interpretador Ruby não permite que tal declaração de

método seja feita. É permitido apenas que um Hash seja declarado como atributo não nomeado no final da declaração de um método:

```
def setup(value, timeout: 0, name: '', **options)
  p value, timeout, name, options
end
```

Com estes últimos argumento, podemos invocar o método setup passando um Hash com os valores que quisermos:

```
setup 79.9, name: 'sedex', timeout: 20,
      url: 'http://correios.cep.com.br/consulta'

# => 79.9, 20, 'sedex',
#   { :url => "http://correios.cep.com.br/consulta" }
```

A dica então é sempre declarar os *named parameters* no final do método, e os outros atributos no começo.

12.3 UTILIZE PREPEND EM VEZ DE INCLUDE

Vamos relembrar rapidamente como funcionam os módulos em Ruby. Módulos podem ser usados para a criação de *namespaces* ou como *container* de métodos que serão incluídos em classes, técnica conhecida como *mixings*. Por exemplo:

```
module FormatadorMoeda
  def valor_formatado
    @valor.por_extenso_em_reais
  end
end

class DVD
  include FormatadorMoeda

  def initialize(valor)
    @valor = valor
  end
end
```

```
end
```

```
dvd = DVD.new 56.6  
dvd.valor_formatado  
# => cinquenta e seis reais e sessenta centavos
```

Como vimos anteriormente, ao dar `include` de um módulo dentro de uma classe, esta 'ganha' todos os métodos que foram declarados no módulo. Algum desenvolvedor mais desatento pode recriar um método `valor_formatado` na classe `DVD` com um comportamento completamente diferente do que foi definido no módulo:

```
class DVD  
  include FormatadorMoeda  
  
  def initialize(valor)  
    @valor = valor  
  end  
  
  def valor_formatado  
    "R$ #{@valor}"  
  end  
end  
  
dvd = DVD.new 56.6  
dvd.valor_formatado # => R$ 56.6
```

E o comportamento do método `valor_formatado` definido no módulo `FormatadorMoeda` foi sobrescrito pela própria classe que o incluiu. O que aconteceu na verdade não foi uma sobrescrita de método, o que aconteceu é que o interpretador Ruby sempre vai procurar pelo método respeitando o `object model`. Isto é, primeiro buscará o método na própria classe `DVD` e, como encontrará uma implementação do método `valor_formatado` disponível, vai executá-lo e não será necessário procurá-lo dentro do módulo `FormatadorMoeda`.

Podemos ordenar que o interpretador Ruby procure pelo método `valor_formatado` primeiro nos módulos e depois na própria classe `DVD`. Basta usarmos o método `prepend` e não mais o método `include`:

```
class DVD
  prepend FormatadorMoeda

  def initialize(valor)
    @valor = valor
  end

  def valor_formatado
    "R$ #{@valor}"
  end
end

dvd = DVD.new 56.6
dvd.valor_formatado
# => cinquenta e seis reais e sessenta centavos
```

Repare que o método `valor_formatado` foi procurado primeiramente no `FormatadorMoeda` e, como ele foi encontrado, não foi necessário buscá-lo na classe `DVD`.

12.4 UTILIZANDO LAZY LOAD NO MÓDULO ENUMERABLE

Ruby é uma linguagem imperativa, porém, permite que sejam feitos códigos elegantes que possuem características funcionais. Por exemplo:

```
[1, 2, 3, 4, 2, 5].map { |numero| numero * 10 }
# [10, 20, 30, 40, 20, 50]
```

Criamos um novo `Array` a partir do primeiro `Array` com todos os números multiplicados por 10. Agora, precisamos que

deste novo `Array` sejam extraídos apenas os números maiores que 30. Simples:

```
[1, 2, 3, 4, 2, 5].map { |numero| numero * 10 }  
  .select { |numero| numero > 30 }  
# [40, 50]
```

Porém, o código anterior, apesar de ser bastante elegante e legível, possui um problema de performance. São criados `Array`s intermediários a cada manipulação que é feita, neste caso, dois `Array`s desnecessários. Na verdade, ao manipular um `Array` pequeno como o do exemplo, isso pode não parecer um problema, mas se estivéssemos parseando um arquivo de 1GB, teríamos sérios problemas.

A versão 2.0 do Ruby copiou, no bom sentido, uma característica presente na maioria das linguagens funcionais existentes, as *lazy collections*. Isso significa que podemos evitar a criação destes `Array`s intermediários e executar o processamento das funções utilizadas para manipular a coleção uma única vez.

Na prática, basta invocarmos o método `lazy` antes de encadear as chamadas funcionais nos dados que desejamos manipular:

```
[1, 2, 3, 4, 2, 5].lazy.map { |numero| numero * 10 }  
  .select { |numero| numero > 30 }  
# <Enumerator::Lazy:  
# <Enumerator::Lazy:  
#   #<Enumerator::Lazy: [1, 2, 3, 4, 2, 5]>:map>:select>
```

Repare que o retorno não foram os números 40 e 50 como anteriormente. Em vez disso, foi retornado um objeto `Enumerator<::Lazy`, ou seja, nenhum processamento foi feito no código anterior. Para que as funções `map` e `select` sejam

executadas no Array original, basta invocar o método `to_a` :

```
lazy_array = [1, 2, 3, 4, 2, 5].lazy
               .map { |numero| numero * 10 }
               .select { |numero| numero > 30 }

p lazy_array.to_a
# [40, 50]
```

12.5 ENCODING UTF-8

Na versão 1.9 do Ruby, o *encoding default* é US-ASCII . Na prática isso significa que, se você quiser utilizar caracteres acentuados, será necessário a utilização de um comentário no começo do arquivo: `# coding: utf-8` , ou uma exceção `invalid multibyte char (US-ASCII)` será lançada.

No versão 2.0, o *encoding default* é UTF-8 . Com isso, ao escrever caracteres com acento, não é mais necessário a utilização do comentário que altera o *encoding*. Você pode querer manter a compatibilidade entre as versões 1.9 e 2.0, então, neste caso, manter o comentário é necessário.

12.6 PRÓXIMOS PASSOS

Neste capítulo, vimos as principais novidades que estão disponíveis na versão 2.0 da linguagem Ruby. No próximo, veremos um dos assuntos mais polêmicos sobre a linguagem: concorrência e paralelismo. Veremos se isso existe de verdade em Ruby e quais as vantagens e desvantagens de cada uma das abordagens disponíveis atualmente.

Você pode discutir sobre este livro no Fórum da Casa do

Código: <http://forum.casadocodigo.com.br>.

APÊNDICE: CONCORRÊNCIA E PARALELISMO

Concorrência é a capacidade de compor um cenário no qual as tarefas podem ser executadas independentes umas das outras. Paralelismo é a capacidade de executar várias destas tarefas simultaneamente.

Complicado, não é mesmo? Vou tentar algo 'do mundo real' para exemplificar a diferença entre estes dois conceitos computacionais.

Vamos supor que você precisa fazer exercícios de matemática e história que foram atribuídos como *homework*. Você decide resolver um exercício de matemática, e depois um exercício de história, depois você faz outro exercício de matemática e após um outro exercício de história, até que todos os problemas estejam resolvidos. Ao trazermos a metáfora para o mundo da computação, estamos fazendo os exercícios de forma concorrente.

Se concorrência significa fazer várias tarefas simultaneamente, uma de cada vez, o que é paralelismo? Paralelismo é a habilidade de executar um pouco de cada tarefa, de maneira simultânea. Por

exemplo, dirigir seu veículo e fazer um audio curso de inglês ao mesmo tempo. Você pode progredir nas duas tarefas de maneira simultânea.

Trazendo de volta para o mundo da computação, mais especificamente para o mundo web, suponha que você tem uma aplicação Rails que possui um grande tráfego de usuários, algo em torno de 1 request por segundo. Ao executar um benchmark nesta aplicação, você percebe que ela tem uma média de tempo de resposta em torno de 100ms por request, o que nos leva a crer que a aplicação suporta 10 requests por segundo, aproximadamente.

Se a aplicação alcançar um número maior que 10 requests por segundo, eles serão enfileirados e *startados* apenas quando o servidor estiver livre, causando um aumento no tempo final do request. Esse é um dos motivos pelos quais devemos nos preocupar com concorrência e paralelismo.

Independente da linguagem que estamos falando, existem diversas maneiras de resolver este problema. Cada uma das soluções causa muita discussão, pois, muitas pessoas defendem seus pontos de vista, tanto os bons quanto os ruins. Frases como: 'Ruby não escala' são muito comuns, porém, é um sinal de que o desenvolvedor não entende corretamente o que é necessário para escalar um programa feito em determinada linguagem, seja ela qual for.

Os mais sensatos defenderão suas abordagens para escalar o sistema, alguns defendem a utilização de Threads, mas talvez se esquecem das dificuldades de lidar com memória compartilhada. Outros dizem que você deve trocar a linguagem e utilizar Node.js, por exemplo. Existem os que defendem a programação orientada

(*event based*) dentro da própria linguagem hospedeira.

O importante é entender os benefícios e malefícios que cada uma das abordagens pode trazer para sua equipe e para a sua empresa, e escolher a que melhor se encaixa no problema. Entender que a escolha da linguagem muda a decisão que será tomada para melhorar a performance de uma aplicação também é importante, pois cada uma delas possui abordagens mais adequadas para resolver o mesmo problema, cada uma com suas características.

A ideia deste capítulo não é ensinar você a escalar uma aplicação, pois isso seria demanda de apenas um livro, mas sim mostrar os caminhos, vantagens e desvantagens das soluções existentes para você melhorar a performance da sua aplicação Ruby, na maioria das vezes Rails também. Proponho um 'cala boca' às pessoas que criticam a linguagem Ruby sem realmente saber o que é possível fazer para lidar com seus problemas.

13.1 THREADS

Antes da versão 1.9 do Ruby, as *Threads* não eram nativas do Sistema Operacional, ou seja, elas eram gerenciadas pelas *Virtual Machine*. E isso causava grandes problemas, já que elas não aproveitavam várias vantagens disponibilizadas automaticamente pelo S.O (leia mais sobre as green threads em: http://en.wikipedia.org/wiki/Green_threads).

A partir da versão 1.9, as *Threads* são nativas do Sistema Operacional, o que pode levar você a pensar que os desenvolvedores Ruby poderiam implementar *Threads* assim

como os desenvolvedores Java fizeram por muitos anos. Porém, isso não é bem a verdade.

A linguagem Ruby possui um mecanismo de *lock* chamado *Global Interpreter Lock*, carinhosamente conhecido como GIL. Este mecanismo evita que várias *Threads* possam alterar um dado compartilhado entre elas. Com isso, é praticamente impossível alcançar concorrência em aplicações Ruby.

Outro agravante é que muitas bibliotecas Ruby são escritas na linguagem C, e estas não são *Thread Safe*. Com isso, seria muito perigoso utilizar estas biblioteca em um ambiente *multi thread* sem utilizar o GIL.

Isso que dizer que nunca poderei usar *Threads* enquanto utilizar Ruby? Não, mentira. Você pode usar *Threads* desde que utilize algum interpretador que não possua um *Global Interpreter Lock*, como o JRuby. Mas lembre-se, principalmente se você veio do mundo Java, você pode usar *Threads* utilizando JRuby, mas enfrentará os mesmos problemas enfrentados quando desenvolvia em Java: garantir que seu código é thread safe, checar que todas as bibliotecas que você utiliza são thread safe também, não fazer *deadlocks* e várias outras dificuldades encontradas quando manipulamos *Threads* manualmente.

13.2 MÚLTIPLOS PROCESSOS

Este é de longe o mecanismos mais usado no mundo Ruby para escalar principalmente aplicações feitas em Rails. Pelo simples motivo de ser difícil trabalhar com *Threads*, os desenvolvedores Ruby na maioria das vezes preferem apenas *startar* mais processos.

O problema neste caso é que você precisa se preocupar em sincronizar os dados que transitam entre os processos, e será necessário alguma ferramenta, como por exemplo, o DRb (<http://segment7.net/projects/ruby/drbb>) que permite a comunicação entre diferentes processos Ruby.

Existe ainda um problema em relação à memória física que será usada. Por exemplo, se um processo Ruby gastar 200MB de memória, e você fazer um *fork* de mais 4 processos idênticos a este, o espaço em memória para cada um será de 200MB. Isso gera uma demanda de pelo menos 1GB de memória física. Servidores web, como o Mongrel, funcionam desta maneira, por isso não devem ser utilizados em produção.

Serviços mais robustos como Passenger (módulo para Apache e o Nginx) e o Unicorn implementaram uma solução um pouco mais elegante e eficiente. Eles fazem o *fork* do processo através do Sistema Operacional Unix, que cria uma cópia do processo original e faz com que eles compartilhem a mesma memória física. A sacada é que, quando um processo altera algum dado na memória, os outros processos não são afetados. Desta maneira, é possível termos 5 processos rodando gastando apenas 200MB de memória física e atendendo cinco vezes mais requisições concorrentemente.

A parte interessante é que podemos parar apenas um processo por vez. Isso é bom quando desejamos, por exemplo, fazer deploy de uma aplicação sem que haja *downtime* para os usuários.

13.3 FIBERS

A partir do Ruby 1.9, é possível criar códigos que são executados de forma concorrente, utilizando as *Fibers*, recurso muito parecido com as *threads*, porém, mais simples e leve. A diferença é que uma *Fiber* não é gerenciada pela máquina virtual ou pelo sistema operacional, mas sim pelo próprio desenvolvedor.

Outra diferença importante é que *fibers* são mais rápidas de serem criadas e não ocupam muito espaço em memória, pois rodam sempre dentro de uma única *thread*. Justamente por rodar em apenas uma única *thread* é que não se pode confiar que dados compartilhados sejam alterados entre diversas *fibers*. Por isso a importância do *Global Interpreter Lock* para garantir a integridade dos dados neste cenário.

A pergunta que nos fazemos é: *se fibers rodam dentro de uma única thread, onde está o grande benefício da concorrência?*

A resposta é que *fibers* são partes de uma solução mais robusta. Elas permitem, por exemplo, que os desenvolvedores gerenciem manualmente o fluxo de concorrência que executará as tarefas necessárias. Por exemplo, suponha que seu servidor web receba um *request* e que existe um tempo, mesmo que mínimo, para que a resposta seja gerada ao usuário. Neste meio tempo, outros *requests* podem chegar e não é interessante que estes esperem que o primeiro *request* seja concluído para serem atendidos.

Ao utilizar uma *fiber* por *request*, garantimos que, ao receber uma requisição, o servidor a deixe sob os cuidados de sua própria *fiber*, enquanto isso outras requisições são atendidas e repassadas para suas próprias *fibers*. Conforme cada *fiber* for concluída, o seu resultado será retornando para o usuário.

O único problema desta abordagem é quando existe a necessidade de utilizar I/O bloqueante, por exemplo, acessar um banco de dados, enviar email etc. Como existe uma única *thread*, qualquer *fiber* que precise utilizar I/O vai bloqueá-la e assim perderemos os benefícios da utilização de *fibers*.

13.4 O DESIGN PATTERN REACTOR

O design pattern *reactor* é bem simples, ele consiste em delegar chamadas IO para serviços externos (*reactors*) que suportam requisições concorrentes. Estes *reactors* proveem uma maneira de registramos *callbacks* que serão disparados assincronamente, assim que as requisições forem completadas.

Suponha que sua aplicação web precisa consultar um *web service* externo para retornar informações ao usuário. Essa requisição pode ser bastante lenta, dependendo da performance de um aplicativo de terceiros. Utilizando *reactor*, nós apenas recebemos a requisição e delegamos a chamada ao *web service* externo para um *handler*, e dizemos para este o que deverá ser feito quando a requisição externa terminar, ou seja, registramos *callbacks* que serão executados baseados na resposta do recurso externo em questão.

No caso da sua aplicação rodar em um servidor web *single thread*, existe uma questão importante. Quando recebemos o *request* e delegamos a chamada ao *web service* para um *handler*, continuamos bloqueando a *thread* e evitando o processamento de novas requisições. Para evitar que isso aconteça, o ideal é encapsular o *request* dentro de uma *fiber*, efetuar o *request* ao *web service* e pausar a *fiber* de modo que outras requisições possam ser

processadas enquanto aguarda-se a chamada ao *web service*. Quando esta chamada for efetuada, a *fiber* é novamente ativada e finalmente a resposta é enviada ao cliente.

Esta é a abordagem utilizada pelo EventMachine (Ruby) e pelo Node.js (JavaScript). O *Thin* é um dos servidores web Ruby escritos baseando-se no EventMachine. Existem também várias bibliotecas baseadas no EventMachine que podem ser usadas de maneira *standalone* para fazer requisições assíncronas.

O único problema de utilizar o padrão *reactor* é a quebra de paradigma no momento de escrever o código. Se você não tomar o cuidado necessário, pode acabar com uma grande quantidade de *callbacks* aninhados, problema também enfrentado quando utilizamos Node.js. Recomendo que você leia o artigo do Martyn Loughran, um dos criadores do Pusher: <http://rubylearning.com/blog/2010/10/01/an-introduction-to-eventmachine-and-how-to-avoid-callback-spaghetti>. Neste artigo, ele explica como evitar *callbacks* macarrônicos quando usamos EventMachine.

13.5 CONCLUSÃO

A conclusão que particularmente chego é que concorrência em Ruby é algo possível e pode ser feito de muitas maneiras. Utilizando os *fibers* que surgiram no Ruby 1.9, em conjunto com IO não bloqueante, permite que seja alcançado altos níveis de concorrência. Existe também a opção de fazer um fork nos processos existentes, como é feito no *passenger*, que pode ser utilizado tanto com *Nginx* quanto com *Apache*.

Lembrando de que, quando desenvolvemos aplicações web, questões como cache e minimizar dependências externas sempre devem ser levadas em consideração para melhorias na performance. Se você precisa escalar uma aplicação web que utiliza Ruby, tem a obrigação de estudar não apenas as técnicas descritas anteriormente, mas também estas questões.