Capítulo 4

Mais Ruby: classes, objetos e métodos

"Uma imagem vale mais que mil palavras"
— Napoleão Bonaparte

Orientação a objetos pura

Entre as linguagens de programação orientada a objetos, muito se discute se são puramente orientadas a objeto ou não, já que grande parte possui recursos que não se comportam como objetos.

Os tipos primitivos de Java são um exemplo desta contradição, já que não são objetos de verdade. Ruby é considerada uma linguagem puramente orientada a objetos, já que **tudo** em Ruby é um objeto (inclusive as classes, como veremos).

4.1 – Mundo orientado a objetos

Ruby é uma linguagem puramente orientada a objetos, bastante influenciada pelo Smalltalk. Desta forma, **tudo** em Ruby é um objeto, até mesmo os tipos básicos que vimos até agora.

Uma maneira simples de visualizar isso é através da chamada de um método em qualquer um dos objetos:

```
"strings são objetos".upcase()
:um_simbolo.object_id()
```

Até os números inteiros são objetos, da classe `Fixnum`:

```
10.class()
```

4.2 - Métodos comuns

Uma das funcionalidades comuns a diversas linguagens orientadas a objeto está na capacidade de, dado um objeto, descobrir de que tipo ele é. No ruby, existe um método chamado `class()`, que retorna o tipo do objeto, enquanto `object_id()`, retorna o número da referência, ou identificador único do objeto dentro da memória heap.

Outro método comum a essas linguagens, é aquele que "transforma" um objeto em uma `String`, geralmente usado para log. O Ruby também disponibiliza esse método, através da chamada ao `to_s()`.

Adicionalmente aos tipos básicos, podemos criar nossos próprios objetos, que já vem com esses métodos que todo objeto possui (`class`, `object_id`).

Para criar um objeto em Ruby, basta invocar o método `new` na classe que desejamos instanciar. O exemplo a seguir mostra como instanciar um objeto:

```
# criando um objeto
objeto = Object.new()
```

Agora é a melhor hora de aprender algo novo

Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

Conheça a Alura.

4.3 – Definição de métodos

`def` é uma palavra chave do Ruby para a **definição** (criação) de métodos, que podem, claro, receber parâmetros:

```
def pessoa.vai(lugar)
  puts "indo para " + lugar
end
```

Mas, e o retorno de um método? Como funciona? Para diminuir o excesso de código que as linguagens costumam introduzir (chamado de ruído sintático), o Ruby optou por retornar o resultado da execução da última instrução executada no

método. O exemplo a seguir mostra um método que devolve uma `String`:

```
def pessoa.vai(lugar)
  "indo para " + lugar
end
```

Para visualizar esse retorno funcionando, podemos acessar o método e imprimir o retorno do mesmo:

```
puts pessoa.vai("casa")
```

Podemos ainda refatorar o nosso método para usar interpolação:

```
def pessoa.vai(lugar)
  "indo para #{lugar}"
end
```

Para receber vários argumentos em um método, basta separá-los por vírgula:

```
def pessoa.troca(roupa, lugar)
  "trocando de #{roupa} no #{lugar}"
end
```

A invocação desses métodos é feita da maneira tradicional:

```
pessoa.troca('camiseta', 'banheiro')
```

Alguns podem até ter um valor padrão, fazendo com que sejam opcionais:

```
def pessoa.troca(roupa, lugar='banheiro')
  "trocando de #{roupa} no #{lugar}"
end

# invocação sem o parametro:
pessoa.troca("camiseta")

# invocação com o parametro:
pessoa.troca("camiseta", "sala")
```

4.4 - Exercícios - Métodos

1. Métodos são uma das formas com que os objetos se comunicam. Vamos utilizar os conceitos vistos no capítulo anterior porém indo um pouco mais além na orientação a objetos agora.

- Crie mais um arquivo no seu diretório **ruby** chamado **restaurante_avancado**.

Ao final de cada exercício você pode rodar o arquivo no terminal com o comando

ruby restaurante_avancado.rb.

2. Vamos simular um método que atribui uma nota a um restaurante.

```
# declaração do método
def qualifica(nota)
  puts "A nota do restaurante foi #{nota}"
end

# chamada do método
qualifica(10)
```

3. Podemos ter parâmetros opcionais em ruby utilizando um valor padrão.

```
def qualifica(nota, msg="Obrigado")
  puts "A nota do restaurante foi #{nota}. #{msg}"
end

# chamadas com parâmetros opcionais
qualifica(10)
qualifica(1, "Comida ruim.")
```

4.5 - Discussão: Enviando mensagens aos objetos

1. Na orientação a objetos a chamada de um método é análoga ao envio de uma mensagem ao objeto. Cada objeto pode reagir de uma forma diferente à mesma mensagem, ao mesmo estímulo. Isso é o polimorfismo.

Seguindo a ideia de envio de mensagens, uma maneira alternativa de chamar um método é usar o método `send()`, que todo objeto em Ruby possui.

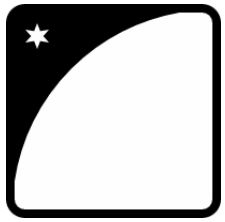
```
pessoa.send(:fala)
```

O método `send` recebe como argumento o nome do método a ser invocado, que pode ser um símbolo ou uma string. De acordo com a orientação a objetos é como se estivéssemos enviando a mensagem **"fala"** ao objeto `pessoa`.

Além da motivação teórica, você consegue enxergar um outro grande benefício dessa forma de invocar métodos, através do `send()`? Qual?

Você pode também fazer o curso RR-71 dessa apostila na Caelum

Querendo aprender ainda mais sobre a linguagem Ruby e o framework Ruby on Rails? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas



com um instrutor?

A Caelum oferece o **curso RR-71** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Desenv. Ágil para Web com Ruby on Rails*.

4.6 - Classes

Para não precisar adicionar sempre todos os métodos em todo objeto que criamos, Ruby possui classes, que atuam como fábricas (molde) de objetos. Classes possibilitam a criação de objetos já incluindo alguns métodos.

```
class Pessoa
  def fala
    puts "Sei Falar"
  end

  def troca(roupa, lugar="banheiro")
    "trocando de #{roupa} no #{lugar}"
  end
end
```

```
p = Pessoa.new
# o objeto apontado por p já nasce com os métodos fala e troca.
```

Todo objeto em Ruby possui o método `class`, que retorna a classe que originou este objeto (note que os parênteses podem ser omitidos na chamada e declaração de métodos):

```
p.class
# => Pessoa
```

O diferencial de classes em Ruby é que são abertas. Ou seja, **qualquer classe** pode ser alterada a qualquer momento na aplicação. Basta "reabrir" a classe e fazer as mudanças:

```
class Pessoa
  def novo_metodo
    # ...
  end
end
```

Caso a classe `Pessoa` já exista estamos apenas reabrindo sua definição para **adicionar** mais código. Não será criada uma nova classe e nem haverá um erro dizendo que a classe já existe.

4.7 - Exercícios - Classes

1. Nosso método qualifica não pertence a nenhuma classe atualmente. Crie a classe **Restaurante** e instancie o objeto.

```
class Restaurante
  def qualifica(nota, msg="Obrigado")
    puts "A nota do restaurante foi #{nota}. #{msg}"
  end
end
```

2. Crie dois objetos, com chamadas diferentes.

```
restaurante_um = Restaurante.new
restaurante_dois = Restaurante.new

restaurante_um.qualifica(10)
restaurante_dois.qualifica(1, "Ruim!")
```

4.8 - Desafio: Classes abertas

1. Qualquer classe em Ruby pode ser reaberta e qualquer método redefinido.

Inclusive classes e métodos da biblioteca padrão, como `Object` e `Fixnum`.

Podemos redefinir a soma de números reabrindo a classe `Fixnum`? Isto seria útil?

```
class Fixnum
  def +(outro)
    self - outro # fazendo a soma subtrair
  end
end
```


Tire suas dúvidas no GUJ Respostas

O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

4.9 – self

Um método pode invocar outro método do próprio objeto. Para isto, basta usar a referência especial `self`, que aponta para o para o próprio objeto. É análogo ao `this` de outras linguagens como Java e C#.

Todo método em Ruby é chamado em algum objeto, ou seja, um método é sempre uma mensagem enviada a um objeto. Quando não especificado, o destino da mensagem é sempre `self`:

```
class Conta
  def transfere_para(destino, quantia)
    debita quantia
    # mesmo que self.debita(quantia)

    destino.deposita quantia
  end
end
```

4.10 – Desafio: self e o método puts

1. Vimos que todo método é sempre chamado em um objeto. Quando não especificamos o objeto em que o método está sendo chamado, Ruby sempre assume que seja em `self`.

Como tudo em Ruby é um objeto, todas as operações devem ser métodos. Em especial, `puts` não é uma operação, muito menos uma palavra reservada da linguagem.

```
puts "ola!"
```

Em qual objeto é chamado o método `puts`? Por que podemos chamar `puts` em qualquer lugar do programa? (dentro de classes, dentro de métodos, fora de tudo, ...)

2. Se podemos chamar `puts` em qualquer `self`, por que o código abaixo não funciona? (Teste!)

```
obj = "uma string"
obj.puts "todos os objetos possuem o método puts?"
```

3. (opcional) Pesquise onde (em que classe ou algo parecido) está definido o método `puts`. Uma boa dica é usar a documentação oficial da biblioteca padrão:

http://ruby-doc.org

4.11 – Atributos e propriedades: acessores e modificadores

Atributos, também conhecidos como variáveis de instância, em Ruby são sempre privados e começam com `@`. Não há como alterá-los de fora da classe; apenas os métodos de um objeto podem alterar os seus atributos (**encapsulamento!**).

```
class Pessoa
  def muda_nome(novo_nome)
    @nome = novo_nome
  end

  def diz_nome
    "meu nome é #{@nome}"
  end
end

p = Pessoa.new
p.muda_nome "João"
p.diz_nome

# => "João"
```

Podemos fazer com que algum código seja executado na criação de um objeto. Para isso, todo objeto pode ter um método especial, chamado de `initialize`:

```
class Pessoa
  def initialize
    puts "Criando nova Pessoa"
  end
end
Pessoa.new
# => "Criando nova Pessoa"
```

Os initializers são métodos privados (não podem ser chamados de fora da classe) e podem receber parâmetros. Veremos mais sobre métodos privados adiante.

```
class Pessoa
  def initialize(nome)
    @nome = nome
  end
end

joao = Pessoa.new("João")
```

Métodos acessores e modificadores são muito comuns e dão a ideia de propriedades. Existe uma convenção para a definição destes métodos, que a

maioria dos desenvolvedores Ruby segue (assim como Java tem a convenção para *getters* e *setters*):

```
class Pessoa
  def nome # acessor
    @nome
  end

  def nome=(novo_nome)
    @nome = novo_nome
  end
end

pessoa = Pessoa.new
pessoa.nome=("José")
puts pessoa.nome
# => "José"
```

Nova editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

4.12 – Syntax Sugar: facilitando a sintaxe

Desde o início, Matz teve como objetivo claro fazer com que Ruby fosse uma linguagem extremamente legível. Portanto, sempre que houver oportunidade de deixar determinado código mais legível, Ruby o fará.

Um exemplo importante é o modificador que acabamos de ver (`nome=`). Os parenteses na chamada de métodos são quase sempre opcionais, desde que não haja ambiguidades:

```
pessoa.nome= "José"
```

Para ficar bastante parecido com uma simples atribuição, bastaria colocar um espaço antes do `'='`. Priorizando a legibilidade, Ruby abre mão da rigidez sintática em alguns casos, como este:

```
pessoa.nome = "José"
```

Apesar de parecer, a linha acima **não é uma simples atribuição**, já que na verdade o método `nome=` está sendo chamado. Este recurso é conhecido como *Syntax Sugar*, já que o Ruby aceita algumas exceções na sintaxe para que o código fique mais legível.

A mesma regra se aplica às operações aritméticas que havíamos visto. Os números em Ruby também são objetos! Experimente:

```
10.class
# => Fixnum
```

Os operadores em Ruby são métodos comuns. Tudo em ruby é um objeto e todas as operações funcionam como envio de mensagens.

```
10.+(3)
```

Ruby tem *Syntax Sugar* para estes métodos operadores matemáticos. Para este conjunto especial de métodos, podemos omitir o ponto e trocar por um espaço. Como com qualquer outro método, os parenteses são opcionais.

4.13 - Exercícios - Atributos e propriedades

1. Crie um **initialize** no Restaurante.

```
class Restaurante
  def initialize
    puts "criando um novo restaurante"
  end
  # não apague o método 'qualifica'
end
```

Rode o exemplo e verifique a chamada ao initialize.

2. Cada restaurante da nossa aplicação precisa de um nome diferente. Vamos criar o atributo e suas propriedades de acesso.

```
class Restaurante
  def initialize(nome)
    puts "criando um novo restaurante: #{nome}"
  end
  # não apague o método 'qualifica'
end

# crie dois nomes diferentes
```

```
restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")
```

3. Embora nossos objetos tenham agora recebido um nome, esse nome não foi guardado em nenhuma variável.

```
class Restaurante
  def initialize(nome)
    puts "criando um novo restaurante: #{nome}"
    @nome = nome
  end

  # altere o método qualifica
  def qualifica(nota, msg="Obrigado")
    puts "A nota do #{@nome} foi #{nota}. #{msg}"
  end
end
```

4. Repare que a variável **nome** é uma variável de instância de Restaurante. Em ruby, o "**@**" definiu esse comportamento. Agora vamos fazer algo parecido com o atributo **nota** e suas propriedades de acesso.

```
class Restaurante
  def initialize(nome)
    puts "criando um novo restaurante: #{nome}"
    @nome = nome
```

```ruby
  end

  # altere o método qualifica
  def qualifica(msg="Obrigado")
    puts "A nota do #{@nome} foi #{@nota}. #{msg}"
  end

  # propriedades
  def nota=(nota)
    @nota = nota
  end

  def nota
    @nota
  end

end

restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")

restaurante_um.nota = 10
restaurante_dois.nota = 1

restaurante_um.qualifica
restaurante_dois.qualifica("Comida ruim.")
```

5. Seria muito trabalhoso definir todas as propriedades de acesso a nossa variáveis. Refatore a classe Restaurante para utilizar o **attr_accessor :nota** Seu arquivo final deve ficar assim:

```
class Restaurante
  attr_accessor :nota

  def initialize(nome)
    puts "criando um novo restaurante: #{nome}"
    @nome = nome
  end

  def qualifica(msg="Obrigado")
    puts "A nota do #{@nome} foi #{@nota}. #{msg}"
  end

end

restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")

restaurante_um.nota = 10
restaurante_dois.nota = 1

restaurante_um.qualifica
restaurante_dois.qualifica("Comida ruim.")
```

4.14 - Coleções

Arrays em Ruby são instâncias da classe `Array`, não sendo simplesmente uma estrutura de dados, mas possuindo diversos métodos auxiliares que nos ajudam no dia-a-dia.

Um exemplo simples e encontrado em objetos de tipo `Array` em várias linguagens é o que permite a inclusão de elementos e `size` que nos retorna a quantidade de elementos lá dentro.

```
lista = Array.new
lista << "RR-71"
lista << "RR-75"
lista << "FJ-91"

puts lista.size
# => 3
```

A tarefa mais comum ao interagir com array é a de resgatar os elementos e para isso usamos `[]` passando um índice como parametro:

```
puts lista[1]
# => "RR-75"
```

```
puts lista[0]
# => "RR-71"
```

Muitas vezes já sabemos de antemão o que desejamos colocar dentro da nossa array e o Ruby fornece uma maneira literal de declará-la:

```
lista = [1, 2, "string", :simbolo, /$regex^/]
puts lista[2]
# => string
```

Um exemplo que demonstra uma aparição de arrays é a chamada ao método `methods`, que retorna uma array com os nomes de todos os métodos que o objeto sabe responder naquele instante. Esse método é definido na classe Object então todos os objetos o possuem:

```
cliente = "Petrobras"

puts cliente.methods
```

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum.

Você pode escolher um curso nas áreas de Java, Front-end, Ruby, Web, Mobile, .NET, PHP e outros, com um plano que dá acesso a todos os cursos.

Conheça os cursos online Alura.

4.15 – Exemplo: múltiplos parâmetros

Em alguns instantes desejamos receber um número arbitrário de argumentos em um método, por exemplo a lista de produtos que estamos comprando em um serviço:

```
def compra(produto1, produto2, produto3, produtoN)
end
```

Para receber um número qualquer de parâmetros usamos a sintaxe * do Ruby:

```
def compra(*produtos)
  # produtos é uma array
  puts produtos.size
end
```

Note que nesse caso, em Ruby receber uma array ou um número indeterminado de argumentos resulta no mesmo código interno ao método pois a variável produtos funciona como uma array.

A mudança ocorre na sintaxe de chamada a esse método. Na versão mostrada acima, invocamos o método compra com diversos parâmetros:

```
compra("Notebook", "Pendrive", "Cafeteira")
```

Já no caso de definir o método recebendo uma array, somos obrigados a definir uma array no momento de invocá-lo:

```
def compra(produtos)
  # produtos é uma array
  puts produtos.size
end
compra( ["Notebook", "Pendrive", "Cafeteira"] )
```

O operador `*` é chamado de *splat*.

4.16 - Hashes

Mas nem sempre desejamos trabalhar com arrays cuja maneira de encontrar o que está lá dentro é através de um número. Por exemplo, ao ler um arquivo de configuração de um servidor desejamos saber qual a porta e se ele deve ativar o suporte ssh:

```
porta = 80
ssh = false
nome = Caelum.com.br
```

Nesse caso não desejamos acessar uma array através de inteiros (`Fixnum`), mas sim o valor correspondente as chaves "porta", "ssh" e "nome".

Ruby também tem uma estrutura indexada por qualquer objeto, onde as chaves podem ser de qualquer tipo, o que permite atingir nosso objetivo. A classe `Hash` é quem dá suporte a essa funcionalidade, sendo análoga aos objetos `HashMap`, `HashTable`, arrays indexados por `String` e dicionários de outras linguagens.

```
config = Hash.new
config["porta"] = 80
config["ssh"] = false
config["nome"] = "Caelum.com.br"

puts config.size
# => 3
```

```
puts config["ssh"]
# => false
```

Por serem únicos e imutáveis, símbolos são ótimos candidatos a serem chaves em Hashes, portanto poderíamos trabalhar com:

```
config = Hash.new
config[:porta] = 80
```

Ao invocar um método com um número maior de parâmetros, o código fica pouco legível, já que Ruby não possui tipagem explícita, apesar de esta não ser a única causa.

Imagine que tenho uma conta bancária em minhas mãos e desejo invocar o método de transferência, que requer a conta destino, a data na qual o valor será transferido e o valor. Então tento invocar:

```
aluno = Conta.new
escola = Conta.new

# Time.now gera um objeto Time que representa "agora"
aluno.transfere(escola, Time.now, 50.00)
```

No momento de executar o método descobrimos que a ordem dos parâmetros era incorreta, o valor deveria vir antes da data, de acordo com a definição do método:

```
class Conta

  def transfere(destino, valor, data)
    # executa a transferência
  end

end
```

Mas só descobrimos esse erro ao ler a definição do método na classe original, e para contornar esse problema, existe um movimento que se tornou comum com a popularização do Rails 2, passando parâmetro através de hash:

```
aluno.transfere( {destino: escola, data: Time.now, valor: 50.00} )
```

Note que o uso do Hash implicou em uma legibilidade maior apesar de uma proliferação de palavras:

```
def transfere(argumentos)
  destino = argumentos[:destino]
  data = argumentos[:data]
  valor = argumentos[:valor]
```

```
  # executa a transferência
end
```

Isso acontece pois a semântica de cada parâmetro fica explícita para quem lê o código, sem precisar olhar a definição do método para lembrar o que eram cada um dos parâmetros.

Combinando Hashes e não Hashes

Variações nos símbolos permitem melhorar ainda mais a legibilidade, por exemplo:

```
class Conta
  def transfere(valor, argumentos)
    destino = argumentos[:para]
    data = argumentos[:em]
    # executa a transferência
  end
end

aluno.transfere(50.00, {para: escola, em: Time.now})
```

Para quem utiliza uma versão anterior ao Ruby 1.9, ao criar um Hash onde a

chave é um símbolo, a sintaxe de Hash é diferente:

```
aluno.transfere(50.00, {:para => escola, :em => Time.now})
```

Além dos parênteses serem sempre opcionais, quando um Hash é o último parâmetro de um método, as chaves podem ser omitidas (*Syntax Sugar*).

```
aluno.transfere destino: escola, valor: 50.0, data: Time.now
```

4.17 – Exercícios – Arrays e Hashes

1. No seu diretório **ruby** crie mais um arquivo chamado **listas.rb**. Teste um array adicionando dois elementos.

```
nomes = []

nomes[0] = "Fasano"
nomes << "Fogo de Chao"

for nome in nomes
  puts nome
end
```

2. Vamos juntar os conhecimentos de classes e métodos para criar uma franquia onde podemos adicionar uma lista de restaurantes. Para isso teremos que criar na inicialização um array de restaurantes.

```
class Franquia

  def initialize
    @restaurantes = []
  end

  def adiciona(restaurante)
    @restaurantes << restaurante
  end

  def mostra
    for restaurante in @restaurantes
      puts restaurante.nome
    end
  end

end

class Restaurante
  attr_accessor :nome
end

restaurante_um = Restaurante.new
```

```
restaurante_um.nome = "Fasano"

restaurante_dois = Restaurante.new
restaurante_dois.nome = "Fogo de Chao"

franquia = Franquia.new
franquia.adiciona restaurante_um
franquia.adiciona restaurante_dois

franquia.mostra
```

3. O método adiciona recebe apenas um restaurante. Podemos usar a syntax com * e refatorar o método para permitir múltiplos restaurantes como parâmetro.

```
# refatore o método adiciona
def adiciona(*restaurantes)
  for restaurante in restaurantes
    @restaurantes << restaurante
  end
end

# adicione ambos de uma só vez
franquia.adiciona restaurante_um, restaurante_dois
```

4. Hashes são muito úteis para passar valores identificados nos parâmetros. Na classe **restaurante** adicione o método **fechar_conta** e faça a chamada.

```
def fechar_conta(dados)
  puts "Conta fechado no valor de #{dados[:valor]}
  e com nota #{dados[:nota]}. Comentário: #{dados[:comentario]}"
end

restaurante_um.fechar_conta valor: 50, nota: 9, comentario: 'Gostei!'
```


Saber inglês é muito importante em TI

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

Pratique seu inglês no Galandra.

4.18 – Blocos e Programação Funcional

Imagine o exemplo a seguir que soma o saldo das contas de um banco:

```
class Banco
```

```
  def initialize(contas)
    @contas = contas
  end

  def status
    saldo = 0
    for conta in @contas
      saldo += conta
    end
    saldo
  end

end

banco = Banco.new([200, 300, 400])
banco.status
```

Esse processo é executado em diversos pontos da nossa aplicação e todos eles precisam exatamente desse comportamento.

Em um dia ensolarado, um ponto de nossa aplicação passa a necessitar da impressão dos saldos parciais, isso é que cada soma seja impressa. Como fazer isso sem alterar os outros tantos pontos de execução e sem duplicação de código?

O primeiro passo é perceber a necessidade que temos de inserir um código

novo, desejamos incluir o seguinte código:

```
for conta in @contas
  saldo += conta
  puts saldo # essa linha é nova
end
```

Então na chamada específica desse método, passemos esse código como "parâmetro":

```
banco.status do |saldo|
  puts saldo
end
```

Isso não é um parâmetro, e sim um bloco de código, o código que desejamos executar. Note que esse bloco recebe um parâmetro chamado **saldo** e que esse parâmetro é o saldo parcial que desejamos imprimir. Para facilitar a leitura podemos renomeá-lo para **saldo_parcial**:

```
banco.status do |saldo_parcial|
  puts saldo_parcial
end
```

Imagine que o bloco funciona exatamente como a definição de uma função em

ruby: ele pode receber parâmetros e ser invocado. Falta invocá-lo no código do banco, para dar a chance de execução a cada chamada do laço:

```
class Banco
  # initialize...
  def status(&block)
    saldo = 0
    for conta in @contas
      saldo += conta
      block.call(saldo)
    end
    saldo
  end
end
```

Note que block é um objeto que ao ter o método `call` invocado, chamará o bloco que foi passado, concluindo nosso primeiro objetivo: dar a chance de quem se interessar no saldo parcial, fazer algo com ele.

Qualquer outro tipo de execução, como outros cálculos, que eu desejar fazer para cada saldo, posso fazê-lo passando blocos distintos.

Ainda faltou manter compatibilidade com aquelas chamadas que não possuem bloco. Para isso, basta verificarmos se foi passado algum bloco como parâmetro

para nossa função, e somente nesse caso invocá-lo.

Isto é, se o bloco foi dado (`block_given?`), então invoque o bloco:

```
for conta in @contas
  saldo += conta

  if block_given?
    block.call(saldo)
  end

end
```

Outra sintaxe de bloco

Existe uma outra sintaxe que podemos utilizar para passar blocos que envolve o uso de chaves:

```
banco.status { |saldo_parcial| puts saldo_parcial }
```

Como vimos até aqui, o método que recebe um bloco pode decidir se deve ou não chamá-lo. Para chamar o bloco associado, existe uma outra abordagem com a

palavra `yield`:

```
if block_given?
  yield(saldo)
end
```

Nessa abordagem, não se faz necessário receber o argumento &block portanto o código do Banco seria:

```
class Banco
  # initialize...
  def status
    saldo = 0
    for conta in @contas
      saldo += conta
      if block_given?
        yield(saldo)
      end
    end
    saldo
  end
end
```

Entender quando usar blocos costuma parecer complicado no início, não se preocupe . Você pode enxergá-los como uma oportunidade do método delegar

parte da responsabilidade a quem o chama, permitindo customizações em cada uma de suas chamadas (estratégias diferentes, ou callbacks).

A biblioteca padrão do Ruby faz alguns usos muito interessantes de blocos. Podemos analisar a forma de iterar em coleções, que é bastante influenciada por técnicas de programação funcional.

Dizer que estamos passando uma função (pedaço de código) como parâmetro a outra função é o mesmo que passar blocos na chamada de métodos.

Para iterar em uma `Array` possuímos o método `each`, que chama o bloco de código associado para cada um dos seus items, passando o item como parâmetro ao bloco:

```
lista = ["rails", "rake", "ruby", "rvm"]
lista.each do |programa|
  puts programa
end
```

A construção acima não parece trazer muitos ganhos se comparada a forma tradicional e imperativa de iterar em um array (`for`).

Imagine agora uma situação onde queremos colocar o nome de todos os

funcionários em maiúsculo, isto é, aplicar uma função para todos os elementos de uma array, construindo uma array nova.

```
funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

for nome in funcionarios
  nomes_maiusculos << nome.upcase
end
```

Poderíamos usar o método each:

```
funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

funcionarios.each do |nome|
  nomes_maiusculos << nome.upcase
end
```

Mas as duas abordagens envolvem o conceito de dizer a linguagem que queremos adicionar um elemento a uma lista existente: o trabalho é imperativo.

Lembrando que o bloco, assim como uma função qualquer, possui retorno, seria muito mais compacto se pudéssemos exprimir o desejo de criar a array diretamente:

```
funcionarios = ["Guilherme", "Sergio", "David"]

nomes_maiusculos = funcionarios.cria_uma_array
```

O código dessa função deve iterar por cada elemento, adicionando eles dentro da nossa array, *exatamente* como havíamos feito antes:

```
class Array
  def cria_uma_array
    array = []
    self.each do |elemento|
      array << elemento.upcase
    end
    array
  end
end
```

Mas podemos reclamar que o método upcase nem sempre vai funcionar: a cada chamada de cria_uma_array queremos executar um comportamento diferente. E essa é a dica para utilização de blocos: passe um bloco que customiza o comportamento do método cria_uma_array:

```
nomes_maiusculos = funcionarios.cria_uma_array do |nome|
  nome.upcase
end
```

E faça o cria_uma_array invocar esse bloco:

```
class Array
  def cria_uma_array
    array = []
    self.each do |elemento|
      array << yield(elemento)
    end
  end
end
```

Esse método que criamos já existe e se chama `map` (ou `collect`), que coleta os retornos de tomas as chamadas do bloco associado:

```
funcionarios = ["Guilherme", "Sergio", "David"]

nomes_maiusculos = funcionarios.map do |nome|
  nome.upcase
end
```

Na programação imperativa tradicional precisamos de no mínimo mais duas linhas, para o array auxiliar e para adicionar os items maiúsculos no novo array.

Diversos outros métodos do módulo `Enumerable` seguem a mesma ideia: `find`, `find_all`, `grep`, `sort`, `inject`. Não deixe de consultar a documentação, que pode

ajudar a criar código mais compacto.

Só tome cuidado pois código mais compacto nem sempre é mais legível e fácil de manter.

4.19 - Exercícios - Blocos

1. Refatore o método mostra de **Franquia** para iterar sobre os elementos usando blocos.

```
def mostra
  @restaurantes.each do |r|
    puts r.nome
  end
end
```

2. Crie um método **relatorio** que envia a lista de restaurantes da franquia. Repare que esse método não sabe o que ocorrerá com cada item da lista. Ele disponibiliza os itens e é o bloco passado que define o que fazer com eles.

```
def relatorio
  @restaurantes.each do |r|
    yield r
```

```
  end
end

# chamada com blocos
franquia.relatorio do |r|
  puts "Restaurante cadastrado: #{r.nome}"
end
```

4.20 – Para saber mais: Mais sobre blocos

Analise e rode o código abaixo, olhe na documentação o método **sort_by** e teste o **next**.

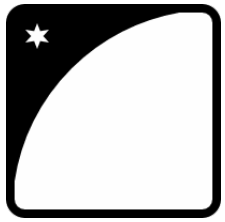
```
caelum = [
   {ruby: 'rr-71', java: 'fj-11'},
   {ruby: 'rr-75', java: 'fj-21'}
 ]

caelum.sort_by { |curso| curso[:ruby] }.each do |curso|
   puts "Curso de RoR na Caelum: #{ curso[:ruby] }"
 end

caelum.sort_by { |curso| curso[:ruby] }.each do |curso|
   next if curso[:ruby] == 'rr-71'
   puts "Curso de RoR na Caelum: #{ curso[:ruby] }"
```

```
end
```


Você não está nessa página a toa



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

Consulte as vantagens do curso *Desenv. Ágil para Web com Ruby on Rails*.

4.21 - Desafio: Usando blocos

1. Queremos imprimir o nome de todos os alunos de uma turma com o código a seguir:

```
fj91 = Turma.new("Guilherme", "Paulo", "Sérgio")

fj91.each do |nome|
  puts nome
end
```

Crie a sua classe **Turma** que durante a invocação do método each, itera por todos os nomes passados em seu construtor.

Tente lembrar dos conceitos de blocos e programação funcional para resolver.

A biblioteca padrão do Ruby faz usos bastante interessante de módulos como mixins. Os principais exemplos são os módulos `Enumerable` e `Comparable`.

4.22 – Manipulando erros e exceptions

Uma *exception* é um tipo especial de objeto que estende ou é uma instância da classe `Exception`. *Lançar* uma exception significa que algo não esperado ou errado ocorreu no fluxo do programa. *Raising* é a palavra usada em ruby para lançamento de exceptions. Para tratar uma exception é necessário criar um código a ser executado caso o programa receba o erro. Para isso existe a palavra `rescue`.

Exceptions comuns

A lista abaixo mostra as exceptions mais comuns em ruby e quando são lançadas, todas são filhas de `Exception`. Nos testes seguintes você pode usar essas

exceptions.

- **RuntimeError** : É a exception padrão lançada pelo método `raise`.
- **NoMethodError** : Quando um objeto recebe como parametro de uma mensagem um nome de método que não pode ser encontrado.
- **NameError** : O interpretador não encontra uma variável ou método com o nome passado.
- **IOError** : Causada ao ler um stream que foi fechado, tentar escrever em algo *read-only* e situações similares.
- **Errno::error** : É a família dos erros de entrada e saída (IO).
- **TypeError** : Um método recebe como argumento algo que não pode tratar.
- **ArgumentError** : Causada por número incorreto de argumentos.

4.23 - Exercício: Manipulando exceptions

1. Em um arquivo ruby crie o código abaixo para testar exceptions. O método `gets`

recupera o valor digitado. Teste também outros tipos de exceptions e digite um número inválido para testar a exception.

```
print "Digite um número:"
numero = gets.to_i

begin
  resultado = 100 / numero
rescue
  puts "Número digitado inválido!"
  exit
end

puts "100/#{numero} é #{resultado} "
```

2. (opcional) Exceptions podem ser lançadas com o comando `raise`. Crie um método que lança uma exception do tipo `ArgumentError` e capture-a com `rescue`.

```
def verifica_idade(idade)
  unless idade > 18
    raise ArgumentError,
    "Você precisa ser maior de idade para jogar jogos violentos."
  end
end

verifica_idade(17)
```

3. (opcional) É possível utilizar sua própria exception criando uma classe e estendendo de `Exception`.

```
class IdadeInsuficienteException < Exception
end

def verifica_idade(idade)
  raise IdadeInsuficienteException,
   "Você precisa ser maior de idade..." unless idade > 18
end
```

Para testar o rescue dessa exception invoque o método com um valor inválido:

```
begin
  verifica_idade(13)
rescue IdadeInsuficienteException => e
  puts "Foi lançada a exception: #{e}"
end
```

Para saber mais: Throw e catch

Ruby possui também throw e catch que podem ser utilizados com símbolos e a sintaxe lembra a de Erlang, onde catch é uma função que, se ocorrer algum throw com aquele label, retorna o valor do throw atrelado:

```
def pesquisa_banco(nome)
  if nome.size < 10
    throw :nome_invalido, "Nome invalido, digite novamente"
  end
  # executa a pesquisa
  "cliente #{nome}"
end

def executa_pesquisa(nome)
  catch :nome_invalido do
    cliente = pesquisa_banco(nome)
    return cliente
  end
end

puts executa_pesquisa("ana")
# => "Nome invalido, digite novamente"

puts executa_pesquisa("guilherme silveira")
# => cliente guilherme silveira
```

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos

atuais.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil. Conheça os títulos e a nova proposta, você vai gostar.

Casa do Código, livros para o programador.

4.24 - Arquivos com código fonte ruby

Todos os arquivos fonte, contendo código Ruby devem ter a extensão *.rb* para que o interpretador seja capaz de carregá-lo.

Existem exceções onde a leitura dos arquivos é feita por outras bibliotecas e, nesses casos, o arquivo possui outras extensões.

Para organizar seu código, é natural dividi-lo em vários arquivos e diretórios, bastando usar o método `require` para incluir o fonte de outro arquivo.

Imagine o arquivo conta.rb:

```
class Conta

  attr_reader :saldo

  def initialize(saldo)
    @saldo = saldo
  end
end
```

Agora podemos acessar uma conta bastando primeiro importar o arquivo que a define:

```
require 'conta'

puts Conta.new(500).saldo
```

Caso a extensão *.rb* seja omitida, a extensão adequada será usada (.rb, .so, .class, .dll, etc). O Ruby procura pelo arquivo em alguns diretórios predefinidos (*Ruby Load Path*). A partir da versão 1.9.2, o diretório atual foi removido do Ruby por questões de segurança.

Uma forma de carregar o arquivo sem especificar o diretório atual é utilizando a forma abaixo:

```
require File.expand_path(File.join(File.dirname(__FILE__),
```

```
'nome_do_arquivo'))
```

Assim como qualquer outra linguagem isso resulta em um possível Load Hell, onde não sabemos exatamente de onde nossos arquivos estão sendo carregados. Tome bastante cuidado para a configuração de seu ambiente.

Caminhos relativos ou absolutos podem ser usados para incluir arquivos em outros diretórios, sendo que o absoluto não é recomendado devido ao atrelamento claro com uma estrutura fixa que pode não ser encontrada ao portar o código para outra máquina:

```
require 'modulo/funcionalidades/coisa_importante'
require '/usr/local/lib/my/libs/ultra_parser'
```

Constantes do sistema

A constante `$:`, ou `$LOAD_PATH` contém diretórios do **Load Path**:

```
$:
# => ["/Library/Ruby/Site/2.0", ..., "."]
```

Existem diversas outras constantes que começam com $, todas elas que são

resquícios de PERL e que, em sua grande maioria dentro do ambiente Rails, possuem alternativas mais compreensíveis como LOAD_PATH.

O comando `require` carrega o arquivo apenas uma vez. Para executar a interpretação do conteúdo do arquivo diversas vezes, utilize o método `load`.

```
load 'conta.rb'
load 'conta.rb'
# executado duas vezes!
```

Portanto no irb (e em Ruby em geral) para recarregar um arquivo que foi alterado é necessário executar um load e não um require - que não faria nada.

4.25 - Para saber mais: um pouco de IO

Para manipular arquivos de texto existe a classe `File`, que permite manipulá-los de maneira bastante simples, utilizando blocos:

```
print "Escreva um texto: "
texto = gets
File.open( "caelum.txt", "w" ) do |f|
```

```
  f << texto
end
```

E para imprimir seu conteúdo:

```
Dir.entries('caelum').each do |file_name|
   idea = File.read( file_name )
   puts idea
end
```

Podemos lidar de maneira similar com requisições HTTP utilizando o código abaixo e imprimir o conteúdo do resultado de uma requisição:

```
require 'net/http'
Net::HTTP.start( 'www.caelum.com.br', 80 ) do |http|
    print( http.get( '/' ).body )
end
```

CAPÍTULO ANTERIOR:

Ruby básico

PRÓXIMO CAPÍTULO:

Metaprogramação

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter