



iugu for devs

Básico de Orientação a Objetos em Ruby

CAMPUS

CODE

Antes de começar

Para estudar o conteúdo desta apostila recomendamos que você tenha Ruby instalado no seu computador. Vamos escrever código de verdade!

O Ruby foi pensado para desenvolvimento em sistemas baseados em Unix, por essa razão, recomendamos instalar sistemas operacionais como: Ubuntu, Fedora, Mint e outras distribuições Linux ou macOS.

Se você usa Windows, recomendamos que você faça *dual boot* na sua máquina para não ter problemas de performance. Outra opção é fazer a instalação através de máquinas virtuais. Como as formas de instalação mudam frequentemente, você pode usar como guia a documentação oficial do sistema. Para iniciantes, recomendamos o Ubuntu e seu [guia de instalação disponibilizado pela comunidade](#). Uma alternativa, não muito performática, é usar o [Ruby Installer](#) em ambientes Windows.

No Windows 10 também há a opção de instalar um Terminal Ubuntu na máquina para simular um sistema Linux dentro do seu Windows. Você pode ler mais neste [tutorial de instalação](#).

Para instalar o Ruby em sistemas Unix como Ubuntu, você pode usar algo simples como abrir o Terminal e digitar o comando `apt-get install ruby`, mas a maioria dos programadores usa gerenciadores de versão para a instalação do Ruby. Esses gerenciadores permitem instalar e alternar entre versões diferentes da linguagem no mesmo computador. Nossa recomendação é o [RVM](#).

De qualquer forma, o mais importante é você ter o ambiente pronto e se sentir confortável com ele. [No site oficial da linguagem](#) você encontra outras opções de instalação além das duas que recomendamos acima.

Com o Ruby instalado, você vai precisar escrever código em um editor de texto. Diferente dos processadores de texto como o Microsoft Word, os editores permitem que você veja realmente todo o conteúdo dos arquivos que está criando. Em programação, uma vírgula ou um espaço em branco na posição errada pode gerar erros nos programas, por isso é importante ter total controle e percepção do código escrito.

Para os editores, temos duas recomendações gratuitas e bastante adotadas pela comunidade: o [Atom](#) e o [Visual Studio Code](#), mas existem muitas alternativas e você pode usar a que achar melhor.

Essa apostila utiliza exemplos de código. Você poderia copiar/colar, no entanto sugerimos que você tente digitar todas as linhas para se habituar ao processo. Isso também vai facilitar a fixação do conteúdo.

Orientação a Objetos com Ruby

Um conceito-chave em programação é o **paradigma da linguagem** ou paradigma da programação. Os paradigmas são utilizados para classificar a linguagem dependendo de suas funcionalidades e determinam a forma como o código é organizado. Cada paradigma, de forma geral, é usado para resolver problemas ou situações diferentes. O recomendado para escrever uma aplicação desktop pode não ser o mais adequado para uma aplicação Web, por exemplo.

Todo código produzido no conteúdo de Lógica de Programação em Ruby utiliza um paradigma simples: o estruturado. Nele, todas as instruções são interpretadas e executadas linha a linha. Mesmo com a possibilidade de agrupar blocos de código em métodos para evitar repetições, trata-se de um paradigma simples e limitado, especialmente com a complexidade sempre crescente em software. Tente imaginar uma aplicação que você usa no dia a dia, como uma aplicação para produzir slides ou uma rede social, escrita em arquivos em que todo código deve ser sequencial.

Projetos de software atuais utilizam, em sua grande maioria, linguagens que seguem os paradigmas Orientado a Objetos ou Funcional. Linguagens como Java, C#, Ruby, Python e PHP, por exemplo, seguem o paradigma Orientado a Objetos.

Nesse conteúdo vamos abordar uma visão prática sobre a **Orientação a Objetos** (OO) utilizando Ruby como referência.



Dica

Uma linguagem de programação pode suportar mais de um paradigma de programação, oferecendo aos programadores mais flexibilidade para resolver diferentes tipos de problema. Ao mesmo tempo, raramente as linguagens seguem todos os princípios de um paradigma.

Classes e objetos

Em OO tudo deve ser um **objeto**. Um objeto pode representar um grupo de informações contextualizadas em nossa aplicação. Digamos que estamos construindo um sistema de gestão de alunos de uma escola. Cada aluno pode ser representado como um objeto, assim como as turmas e os professores.

De acordo com interações dos usuários em uma aplicação orientada a objetos, o **estado** desses objetos pode ser alterado. Por exemplo: um aluno pode se matricular em uma turma e um professor pode ser alocado em uma turma. Essa troca de estados acontece através da **troca de mensagens** entre os objetos. Essas mensagens são, na prática, **métodos** executados dentro do código.

Em nosso exemplo, queremos manter uma padronização entre os objetos similares: todos os alunos devem ter os mesmos dados armazenados (nome, matrícula, data de nascimento, turma em que está matriculado) e também garantir que todos alunos sejam capazes de executar os mesmos métodos. Para isso, utilizamos uma camada de **abstração**, as **classes**.

Essa é toda a teoria base de Orientação a Objetos, mas a proposta aqui é trazer uma visão prática, então vamos ao código. A partir de agora vamos escrever e executar código Ruby.

Criando uma classe

No fim do dia queremos ter um conjunto de objetos interagindo, mas, para chegar lá, o caminho que as linguagens OO oferecem é a criação de classes. Em cada classe, temos:

- Atributos: dados que serão armazenados por cada objeto;
- Métodos: as mensagens que nossa classe pode receber durante a execução do programa.

Numa analogia simples podemos dizer que uma classe é como a planta baixa de uma casa. Ela possui os detalhes para construir uma casa, mas não é, de fato, uma casa.

Vamos criar a classe `Aluno`. Primeiro crie um diretório para nosso exemplo. Dentro deste diretório crie o arquivo `aluno.rb` e, em seguida, adicione o seguinte ao arquivo:

```
class Aluno
end
```



Dica

Em Ruby sempre usamos `snake_case` para definir variáveis, métodos e nomes de arquivos, mas nomes de classes sempre iniciam com letra maiúscula.

Um **objeto é uma instância de uma classe**. Em programação, uma instância é uma ocorrência concreta de algo, nesse caso um objeto de uma classe. Usamos a classe como base para gerar cada objeto, assim como uma planta baixa pode ser utilizada como referência para construir várias casas. Para criar uma instância, nós usamos o método `new`. Geralmente armazenamos uma instância/objeto em uma variável para acessar seus **atributos** e enviar chamadas de nossos **métodos**.

Para podermos demonstrar um exemplo, vamos utilizar o IRB (*Interactive Ruby Shell*). Com ele você pode testar código Ruby diretamente no seu Terminal.

Sua classe `Aluno`, criada no arquivo `aluno.rb`, é uma classe criada por você e não está disponível por padrão, por isso deve ser carregada. Para isso, primeiro salve seu

arquivo, abra seu Terminal e navegue até a pasta criada para nosso projeto. Dentro da pasta, execute o comando `irb` e utilize o seguinte código:

```
ruby-2.6.1 :001 > require_relative 'aluno'  
=> true
```

Seu arquivo `aluno.rb` foi lido e carregado pelo Ruby! Lembre-se que depois de realizar uma alteração neste arquivo, você precisa sair do IRB com o comando `exit`, entrar novamente e carregar o arquivo para atualizar as modificações!

Agora, com o IRB rodando e o arquivo da sua classe carregado, escreva:

```
jose = Aluno.new  
=> #<Aluno:0x00007fbc2f087710>
```



Dica

Não é bom usar caracteres especiais como `é`, `ã` ou `ç`, já que a maioria das linguagens de programação trabalha com a língua inglesa.

Ok! Temos uma classe `Aluno` e um objeto `jose`, mas qual é a utilidade disso? Como diferenciar os diversos objetos criados a partir de uma mesma classe, como no código abaixo?

```
jose = Aluno.new  
=> #<Aluno:0x00007fbc2f087710>  
lucas = Aluno.new  
=> #<Aluno:0x00007fbc2f084498>  
patricia = Aluno.new  
=> #<Aluno:0x00007fbc2f8d5098>
```

Esse questionamento nos leva de volta aos conceitos e uso prático de Orientação a Objetos. De certa forma, podemos dizer que as linguagens de programação nos oferecem tipos de dados básicos como *strings*, inteiros (*integers*), números decimais (*floats*) e *booleans*. Mas para representar informações mais complexas precisamos reunir esses atributos em um novo formato.

Imagine que você está criando um *e-commerce*. Para representar um produto de forma completa, precisamos reunir um nome, uma marca, uma cor, um conjunto de dimensões, o peso, a data de validade... Ao criar uma classe, queremos abstrair esse conjunto de informações e permitir que infinitos objetos sejam criados contendo sempre o mesmo conjunto de informações. Essas informações que ficarão armazenados em cada objeto são os **atributos**.

Atributos

Na classe, podemos definir um conjunto de **atributos** que serão utilizados em todos os seus objetos. Os atributos são como variáveis amarradas a cada objeto e na classe devemos definir seu nome e, em linguagens fortemente tipadas, o seu tipo de dado. Um aluno pode ter uma *string* `nome`, outra *string* `telefone` e um inteiro `número de matrícula`, por exemplo.

Em Ruby, os atributos são definidos durante a criação dos objetos, ou seja, na execução do método `new`. Esse método é especial no Ruby – para alterar seu comportamento padrão devemos criar um método `initialize` na classe. Chamamos esse método de **construtor**, pois ele é usado na construção de cada objeto.



Dica

Em Ruby esse é o único caso em que você vai criar um método com um nome (`initialize`) mas executá-lo com outro nome (`new`). Cada linguagem costuma ter uma sintaxe especial para executar os métodos construtores das classes.

Utilizamos a notação `@variavel` para declarar os atributos dos objetos `Aluno` que serão gerados. Podemos dizer então que **atributos** são **variáveis** relacionadas a uma **instância da classe**.

```
class Aluno
  def initialize(nome, telefone, matricula)
    @nome = nome
    @telefone = telefone
    @matricula = matricula
  end
end
```

Se você tentar criar objetos de `Aluno` novamente vai ter problemas, afinal agora cada objeto precisa de 3 parâmetros para ser criado.

```
jose = Aluno.new
ArgumentError (wrong number of arguments (given 0, expected 3))
```

Para criar os objetos, veja o exemplo a seguir:

```
ana = Aluno.new('Ana', '99 9999-9999', 1234)
=> #<Aluno:0x00007fd3f28b4d18 @nome="Ana", @telefone="99 9999-9999",
@matricula=1234>
```

Um pouco sobre abstração

Agora que você entende o que é um objeto, podemos falar mais sobre um importante conceito de Orientação a Objetos, a abstração. Tudo no mundo cotidiano pode ser abstraído em um objeto composto de regras específicas (e talvez complexas): um gato, uma bicicleta e até você! No início do capítulo já utilizamos o conceito de abstração quando descrevemos o exemplo do aluno e seus componentes. Abstração destina-se a receber objetos do mundo real e transformá-los em objetos na programação. Para isso, pegamos um objeto e removemos características até reduzi-las apenas às mais essenciais, no nosso caso, às características funcionais mais importantes dentro do contexto de nosso programa.

Repare que programas ou sistemas com diferentes propósitos podem ter abstrações diferentes de um mesmo elemento da vida real. Um *e-commerce* e uma rede social podem ter a classe `Usuario`, que representa uma pessoa dentro daqueles contextos completamente diferentes. A primeira focada é em itens como endereço de entrega e formas de pagamento enquanto a segunda trata de status de relacionamento, lista de amigos, gostos musicais etc.

Modificando nossos objetos

Ao criar objetos, damos valores aos atributos, no entanto, o valor do atributo de um objeto pode e deve mudar ao longo do ciclo de execução de um programa. No contexto que estamos usando, deve ser possível editar o telefone de um aluno, por exemplo.

Por outro lado, uma boa prática de OO é que os atributos devem ser sempre protegidos e a leitura ou modificação dos seus valores originais seja controlada via métodos. Em algumas linguagens, esses métodos são conhecidos como *getters* e *setters*, pois eles servem simplesmente para obter (*get*) ou atribuir (*set*) um valor a um atributo.

O código abaixo mostra os erros que recebemos do Ruby ao tentar obter ou atualizar os valores dos atributos de um aluno.

```
ana = Aluno.new('Ana', '99 9999-9999', 1234)
=> #<Aluno:0x00007fd3f28b4d18 @nome="Ana", @telefone="99 9999-9999",
@matricula=1234>

ana.nome
NoMethodError (undefined method 'nome' for
#<Aluno:0x00007fd3f28b4d18>)
```

```
ana.telefone = '00 8888-9999'
NoMethodError (undefined method 'telefone=' for
#<Aluno:0x00007fd3f28b4d18>)
```

No exemplo acima, os erros apontados durante a execução são dicas dos métodos que devemos criar em nossa classe. Vamos criar dois métodos relacionados ao atributo telefone, um para obter o telefone e outro para atualizar o telefone de um objeto do tipo `Aluno`.

```
class Aluno
  def initialize(nome, telefone, matricula)
    @nome = nome
    @telefone = telefone
    @matricula = matricula
  end

  # Getter
  def telefone
    return @telefone
  end

  # Setter
  # Sim! Em Ruby podemos declarar métodos com =
  def telefone=(valor)
    @telefone = valor
  end
end
```

Agora podemos utilizar nossos novos métodos. Abra o IRB novamente e recarregue nossa classe `Aluno` para então fazer o seguinte:

```
rafael = Aluno.new('Rafael Silva', '11 1234-5678', 333444555)
=> #<Aluno:0x000000025cd080 @nome="Rafael Silva", @telefone="11
1234-5678", @matricula=333444555

rafael.telefone
# => "11 1234-5678"

rafael.telefone = '11 91234-5678'
# => "11 91234-5678"
```


Com algumas linhas, criamos um *getter* e um *setter* para o **atributo** telefone do aluno. Em Ruby esses métodos também são conhecidos por outro nome: *readers* e *writers*. *Readers* são o equivalente dos *getters* e *writers* são o equivalente dos *setters*. Repare que a ideia é a mesma, somente com outra nomenclatura.

Ruby não é conhecida como a linguagem amiga do programador à toa. Como *readers* e *writers* são muito comuns dentro das classes, a linguagem nos dá uma forma de resumir esse código em uma linha.

Podemos declarar todos os métodos de leitura de valor com o código `attr_reader`, assim como podemos definir todos os métodos de escrita com `attr_writer`. Se um mesmo atributo deve permitir a leitura e escrita podemos usar o `attr_accessor`.

```
class Aluno
  attr_accessor :nome, :telefone, :matricula

  def initialize(nome, telefone, matricula)
    @nome = nome
    @telefone = telefone
    @matricula = matricula
  end
end
```

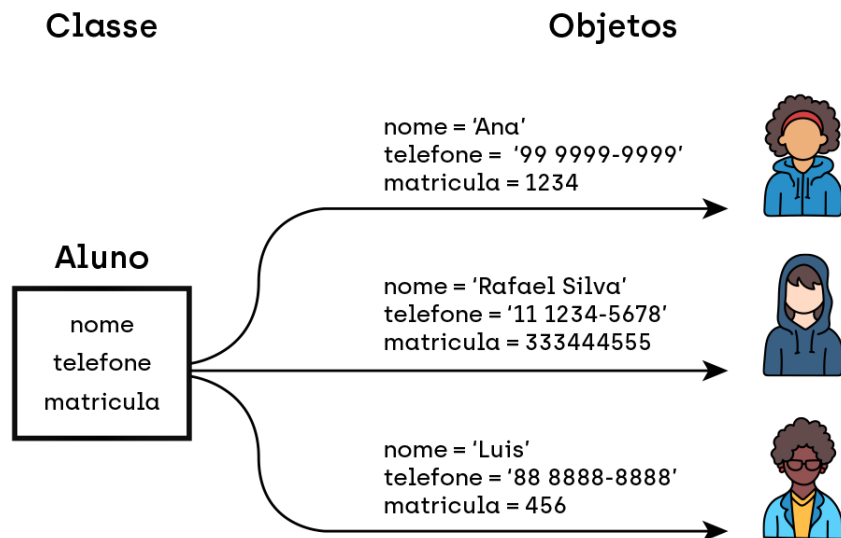
Novamente, abra o IRB em seu Terminal e carregue novamente a classe `Aluno` com `require_relative 'aluno'`.

```
rafael = Aluno.new('Rafael Silva', '11 1234-5678', 333444555)
# => #<Aluno:0x000000025cd080 @nome="Rafael Silva", @telefone="11
1234-5678", @matricula=333444555

rafael.nome
# => "Rafael Silva"

rafael.nome = 'Rafael da Silva'
# => "Rafael da Silva"
```

Voltando à teoria que apresentamos no começo, objetos devem possuir um estado. Esse estado nada mais é do que o conjunto de informações armazenadas em cada objeto e os atributos são responsáveis por isso. Nos trechos de código acima podemos visualizar que em um grande sistema orientado a objetos temos vários objetos de diferentes classes e cada um com seus atributos.



Dica

A partir do momento que criamos os atributos usando os métodos `attr_reader`, `attr_writer` e `attr_accessor`, o uso direto das variáveis iniciadas com `@` passa a ser desencorajado. A boa prática é sempre usar os métodos para obter e atribuir valores.

Em nosso sistema de alunos, temos também as turmas e os professores. Podemos imaginar que a classe `Turma` tem atributos como nome, número da sala e professor responsável. Já os professores podem ter um nome, seu código de funcionário, sua data de nascimento e um *boolean* que indica se estão de férias ou não.

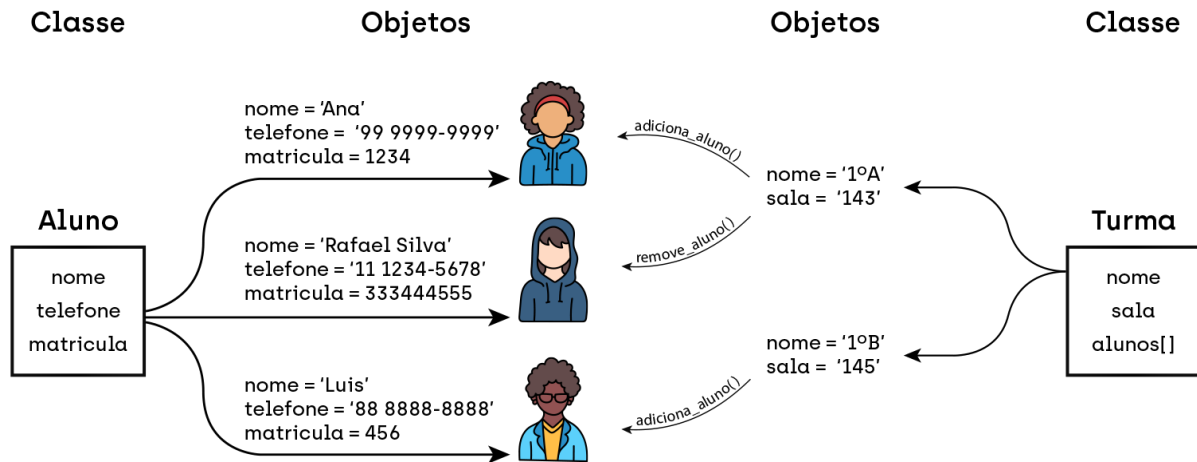


Exercício

Crie as outras duas classes do nosso exemplo. Professor deve ter um nome e um código de funcionário. Já a turma deve ter nome, o número da sala e o professor responsável. Cada classe deve ser criada em um arquivo diferente: `professor.rb` e `turma.rb`.

Métodos

Em nossa introdução a OO falamos que objetos trocam mensagens entre si. No nosso código, essas mensagens são nada mais do que métodos, que são os responsáveis pela mudança de estados do objeto.



Assim como os atributos, os métodos devem ser declarados em nossas classes e estarem disponíveis para todos os objetos criados para esta classe. Veja abaixo a classe **Professor**.

```
class Professor
  attr_reader :ferias
  attr_accessor :nome, :codigo_funcionario, :disciplina

  def initialize(nome, codigo_funcionario, disciplina)
    @nome = nome
    @codigo_funcionario = codigo_funcionario
    @disciplina = disciplina
    @ferias = false
  end
end
```

Repare que o atributo **ferias** só possui permissão de leitura, diferente de **nome** e **codigo_funcionario**. No código do construtor (método **initialize**) também existe uma pré-definição de que o atributo **ferias** sempre é inicializado com o valor **false**. Repare que não é possível enviar o valor do atributo **ferias** como parâmetro ao criar um novo professor.

Vamos adicionar um novo método que indica que um professor entrou em férias.

```
class Professor
  attr_reader :ferias
  attr_accessor :nome, :codigo_funcionario, :disciplina

  def initialize(nome, codigo_funcionario, disciplina)
    @nome = nome
    @codigo_funcionario = codigo_funcionario
    @disciplina = disciplina
    @ferias = false
  end

  def inicia_ferias()
    @ferias = true
  end
end
```

Agora podemos, para qualquer objeto da classe `Professor`, enviar a mensagem `'inicia_ferias'` que teremos uma mudança no estado do objeto.

```
alan = Professor.new("Alan", 12345, 'História')
# => #<Professor:0x000055574e7d4060 @nome="Alan",
@codigo_funcionario=12345, @disciplina="História", @ferias=false>

alan.inicia_ferias()

alan.inspect #vamos inspecionar o objeto para ver seu estado
atual
# => #<Professor:0x000055574e7d4060 @nome="Alan",
@codigo_funcionario=12345, @disciplina="História", @ferias=true>
```

A linha `alan.inicia_ferias()` pode ser lida como: 'Estamos enviando a mensagem `inicia_ferias` para o objeto `alan`, que é da classe `Professor`'. Em Ruby você pode inclusive escrever essa linha da seguinte forma: `alan.send(:inicia_ferias)`. Apesar de ser uma representação mais fiel à teoria, o modo mais usado ainda é o que fizemos no trecho de código acima.



Exercício

Criamos o método `inicia_ferias` e agora você pode criar o método `encerra_ferias`. Mas além de alterar o valor do atributo `@ferias`, vamos criar dois novos atributos: `@data_inicio_ferias` e `@data_fim_ferias`. Esses dois atributos devem ser atualizados nos métodos `inicia_ferias` e `encerra_ferias`.

Uma das formas de obter a data atual em Ruby é usar a classe `Time`. O método `Time.now()` retorna o dia e hora atual do seu computador.

Vamos criar mais um método para praticar? Digamos que sua classe `Turma` ficou da forma abaixo:

```
class Turma
end
```

Vamos adicionar um atributo `@alunos`. Esse atributo deve ser um *array* com todos os alunos que fazem parte dessa turma, por isso o nome no plural. Vamos assumir que uma turma é criada sempre com a lista de alunos vazia. Então acrescentamos ao construtor a linha: `@alunos = []`.

```
class Turma
  attr_accessor :alunos, :nome
  def initialize(nome)
    @nome = nome
    @alunos = []
  end
end
```

O método `adiciona_aluno` vai ser o responsável por adicionar novos alunos a cada objeto de turma. Para isso precisamos receber um objeto do tipo `aluno` que deve ser adicionado. Então, teremos um parâmetro que chamaremos de `aluno`.

```
def adiciona_aluno(aluno)
  alunos << aluno
end
```

Dentro do nosso método usamos o método `<<` da classe `Array`, afinal o atributo `alunos` é um *array* e todos os seus métodos podem ser utilizados. Conhecer os métodos principais dos tipos de dados básicos é muito importante para não reinventar a roda dentro do nosso código.



Exercício

Crie um método `total_alunos` que imprima a quantidade total de alunos de uma turma. Leia mais sobre os métodos de *arrays* na [documentação oficial do Ruby](#).

Agora que você entende os principais conceitos de **linguagens orientadas a objeto** e a sintaxe para utilizar esses conceitos em Ruby, vamos continuar nos aventurando?! Para facilitar a construção de código com OO, a maioria das linguagens oferece mais recursos além de classes, atributos e métodos.

Protegendo dados

Muitas vezes, queremos evitar o uso de certos trechos do nosso código por diversos fatores como: legibilidade, flexibilidade e facilitar modificações. Em OO esse conceito é conhecido como **encapsulamento**. Como o nome diz, ele é usado para isolar comportamentos e controlar o acesso a dados dos objetos das nossas classes. Em Ruby (e quase qualquer outra linguagem) podemos declarar atributos ou métodos como públicos ou privados e assim controlar o seu acesso.

Vamos voltar à nossa classe `Turma` e seu *array* de alunos. Ao criar uma nova turma, o *array* de alunos é criado vazio e temos um método para adicionar um aluno a uma turma. Mas você pode notar que com o `attr_accessor :alunos` podemos, a qualquer momento, sobrescrever todo *array* de alunos com um novo *array* vazio por exemplo.

```
ana = Aluno.new('Ana', '99 9999-9999', 123)
=> #<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99 9999-9999",
@matricula=123>

luis = Aluno.new('Luis', '88 8888-8888', 456)
=> #<Aluno:0x00007f83660d0520 @nome="Luis", @telefone="88
8888-8888", @matricula=456>

turma1 = Turma.new('Turma 101')
=> #<Turma:0x00007f83660d8018 @nome="Turma 101", @alunos=[]>

turma1.adiciona_aluno(ana)
=> [#<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99
9999-9999", @matricula=123>]
```

```
turma1.adiciona_aluno(luis)
=> [#<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99
9999-9999", @matricula=123>, #<Aluno:0x00007f83660d0520 @nome="Luis",
@telefone="88 8888-8888", @matricula=456>]

turma1.alunos = [] #aqui perdemos a referência de todos alunos que
adicionamos nas linhas acima
```

Em nossa classe `Turma`, se queremos gerenciar os alunos de uma turma somente por métodos, podemos utilizar a notação *private*.

```
class Turma

  attr_reader :alunos
  attr_accessor :nome

  def initialize(nome)
    @nome = nome
    @alunos = [ ]
  end

  def adicionar_aluno(aluno)
    alunos << aluno
  end

  private

  attr_writer :alunos
end
```

No código acima fechamos o método *setter* somente para dentro da nossa classe, ou seja, não podemos manipulá-lo de outro lugar. Podemos trocar seu valor somente através dos métodos declarados na classe `Turma`.

Voltando ao exemplo de execução de código anterior:

```
ana = Aluno.new('Ana', '99 9999-9999', 123)
=> #<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99 9999-9999",
@matricula=123>
```

```
luis = Aluno.new('Luis', '88 8888-8888', 456)
=> #<Aluno:0x00007f83660d0520 @nome="Luis", @telefone="88
8888-8888", @matricula=456>

turma1 = Turma.new('Turma 101')
=> #<Turma:0x00007f83660d8018 @nome="Turma 101", @alunos=[]>

turma1.adiciona_aluno(ana)
=> [#<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99
9999-9999", @matricula=123>]

turma1.adiciona_aluno(luis)
=> [#<Aluno:0x00007f83668ea260 @nome="Ana", @telefone="99
9999-9999", @matricula=123>, #<Aluno:0x00007f83660d0520 @nome="Luis",
@telefone="88 8888-8888", @matricula=456>]

turma1.alunos = []
NoMethodError (private method `alunos=' called for
#<Turma:0x00007f83660d8018>)
```

Viu como isso tem um grande ganho? Desta forma, protegemos nossos objetos e, de quebra, podemos evitar que outros programadores desavisados usem métodos que não deveriam ser chamados.

Herança

Com o tempo, nossos programas vão evoluindo e novas funcionalidades podem ser adicionadas. Nossa implementação foi bem sucedida até agora, mas uma escola não funciona apenas com professores. Por isso podemos adicionar ao programa outros tipos funcionários. Assim como professores, os demais funcionários tem um nome, um status de férias e um código de funcionário. Somente a disciplina é exclusiva dos professores. Um professor, então, é um funcionário mais especializado, vamos ver um exemplo de código para a classe `Funcionario`:


```
class Funcionario
  attr_accessor :nome, :ferias, :codigo_funcionario
  def initialize(nome, codigo_funcionario)
    @nome = nome
    @codigo_funcionario = codigo_funcionario
    @ferias = false
  end
end
```

A herança permite abordar o cenário acima evitando duplicar código em nosso programa. Com ela, uma classe pode herdar todos os atributos e métodos de outra classe.

Dado que os professores são funcionários, podemos usar a notação `Professor < Funcionario` para indicar que `Professor` herda de `Funcionario`. Com isso podemos também tirar todos os atributos duplicados da classe professor e manter somente o atributo `:disciplina`.

```
class Professor < Funcionario
  attr_accessor :disciplina
end
```

Agora precisamos tratar do método construtor (`initialize`). Em `Funcionario` temos um construtor, mas `Professor` precisa receber seu atributo adicional `:disciplina`. O construtor também é herdado, então, no caso do código acima, se tentarmos criar um novo objeto de professor não vai ser possível passar a disciplina como terceiro parâmetro para `.new()`.

```
carlos = Funcionario.new('Carlos', 123)
=> #<Funcionario:0x00007fbf9110a6a0 @nome="Carlos",
@codigo_funcionario=123, @ferias=false>

tereza = Professor.new('Tereza', 123, 'Ciências')
ArgumentError (wrong number of arguments (given 3, expected 2))
```

Por ter um atributo a mais que influencia na criação dos objetos, a classe `Professor` precisa manter seu próprio método `initialize` permitindo receber o valor do atributo `disciplina`.

```
class Professor < Funcionario
  attr_accessor :disciplina

  def initialize(nome, codigo_funcionario, disciplina)
    @nome = nome
    @codigo_funcionario = codigo_funcionario
    @ferias = false
    @disciplina = disciplina
  end
end
```

O problema do código acima é duplicar praticamente todo método `initialize` que já havia sido herdado de `Funcionario`. Uma solução é enviarmos os parâmetros `nome` e `codigo_funcionario` para o construtor que herdamos, para isso usamos o método `super`. Esse método permite chamar o método com o mesmo nome na classe pai, ou seja, o `initialize` da classe pai.

```
class Professor < Funcionario
  attr_accessor :disciplina

  def initialize(nome, codigo_funcionario, disciplina)
    super(nome, codigo_funcionario)
    @disciplina = disciplina
  end
end
```

Ao definir o método `initialize` em `Professor` estamos declarando que essa classe tem um comportamento mais específico que aquele herdado de `Funcionario`. Essa possibilidade é válida para qualquer método, na verdade. No caso de métodos que existem nas duas classes, o método da classe mais específica vai ter sempre prioridade. Isso se deve ao que chamamos de **method lookup** – algo como 'busca de métodos' – nas linguagens orientadas a objetos. Ao acionar um método de um objeto de uma classe, a linguagem procura o método na classe e, caso não o encontre, parte para a classe definida na herança (caso ela exista).



Exercício

Adicione na classe `Funcionario` um método chamado `imprime` que deve imprimir na tela (com uso do `puts`) as informações do funcionário. Em seguida, abra o IRB e crie objetos de `Funcionario` e de `Professor` e faça chamadas do seu novo método.

Quando estiver tudo funcionando, acrescente o método `imprime` também em `Professor` para imprimir, além das informações herdadas de funcionário, a disciplina do professor.

O resultado final deve ser parecido com isso no seu Terminal:

```
require_relative 'funcionario'
=> true

require_relative 'professor'
=> true

carlos = Funcionario.new('Carlos', 123)
=> #<Funcionario:0x00007fe905159f78 @nome="Carlos",
@codigo_funcionario=123, @ferias=false>

tereza = Professor.new('Tereza', 123, 'Ciências')
=> #<Professor:0x00007fe9051601c0 @nome="Tereza",
@codigo_funcionario=123, @ferias=false,
@disciplina="Ciências">

carlos.imprime()
Funcionário Carlos - Código 123 - Férias: false
=> nil

tereza.imprime()
Professor Tereza - Código 123 - Férias: false -
Disciplina Ciências
=> nil
```



Dica

Todas as classes criadas por você têm uma 'árvore' de herança criada pelo próprio Ruby. No IRB, depois de carregar a classe `Professor` você pode executar `Professor.ancestors`. O resultado deve ser:

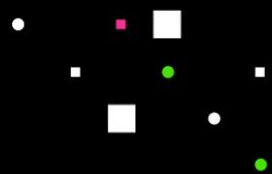
```
Professor.ancestors  
=> [Professor, Funcionario, Object, Kernel,  
    BasicObject]
```

É através dessa herança 'nativa' que todas as classes que criamos possuem métodos básicos para o Ruby como `to_s` e `==`.

Acabamos por aqui mas você pode continuar

Terminamos aqui nossa introdução a Orientação a Objetos. Esse assunto é bastante extenso, mas nosso objetivo de apresentar os principais pontos com uma abordagem prática está cumprido. Por ser um paradigma muito utilizado, você pode e deve continuar estudando e testando código inclusive em outras linguagens além do Ruby. Pesquise mais sobre interfaces em Java ou C#, polimorfismo ou tipos primitivos para ir mais fundo em OO.

Esse material está em constante evolução e sua opinião é muito importante. Se tiver sugestões ou dúvidas que gostaria de nos enviar, entre em contato pelo nosso e-mail: contato@campuscode.com.br.



| Versão | Data de atualização |
|--------|---------------------|
| 1.0 | 18/12/2020 |



BY

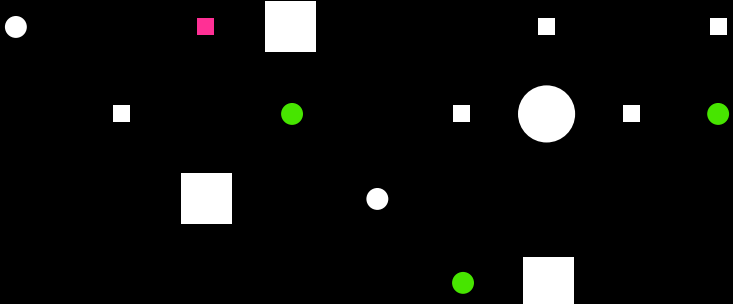


NC



SA

Attribution-NonCommercial-ShareAlike 4.0
International (CC BY-NC-SA 4.0)



CAMPUS CODE

al. santos, 1293, conj. 73
cerqueira César, são paulo, sp
e-mail: contato@campuscode.com.br
telefone: (11) 2369-3476
whatsapp: [\[11\] 2369-3476](https://wa.me/551123693476)

 [/campuscodebr](https://www.facebook.com/campuscodebr)

 [/campuscodebr](https://twitter.com/campuscodebr)

 [/campuscodebr](https://www.youtube.com/campuscodebr)

