Aula 16

Programação Orientada a Objetos: Herança, Módulos e Mixins

Prof. Jalerson Lima

Apresentação

Nessa aula iremos nos aprofundar mais um pouco na Programação Orientada a Objetos, estudando sobre herança, que nos permite organizar as classes de forma hierárquica, além de aprender sobre módulos e *mixins*, que são mecanismos que nos permitem simular herança múltipla em Ruby.

Objetivos

1. Compreender o conceito de herança entre classes;
2. Aprender como definir superclasses e subclasses;
3. Compreender e aplicar os conceitos de módulos e *mixins*;
4. Exercitar os conhecimentos através de exercícios práticos.

1. Herança

Conforme explicado em aulas anteriores, a Programação Orientada a Objetos tenta facilitar a programação aproximando-a do mundo real, permitindo criar objetos para representar elementos da realidade. Mas nós sabemos que, no mundo real, vários elementos se organizam em hierarquia, portanto a Programação Orientada a Objetos também deve nos oferecer meios para que os objetos sejam organizados dessa forma, e é exatamente isso o que a herança nos permite fazer: permitir que classes e objetos se relacionem em hierarquia.

Através da herança de classes podemos criar superclasses (ou classes pai), que representam elementos mais genéricos e se relacionam com subclasses (ou classes filha), que representam elementos mais específicos. Por exemplo: podemos criar a classe `SerVivo`, que pode representar qualquer ser vivo. Também podemos criar as classes `Animal` e `Vegetal` que herdam da classe `SerVivo`. Podemos ainda criar mais duas classes, `Mamifero` e `Herbivoro`, que herdam da classe `Animal`. Observe a Figura 1, que ilustra graficamente como essas classes se organizam.

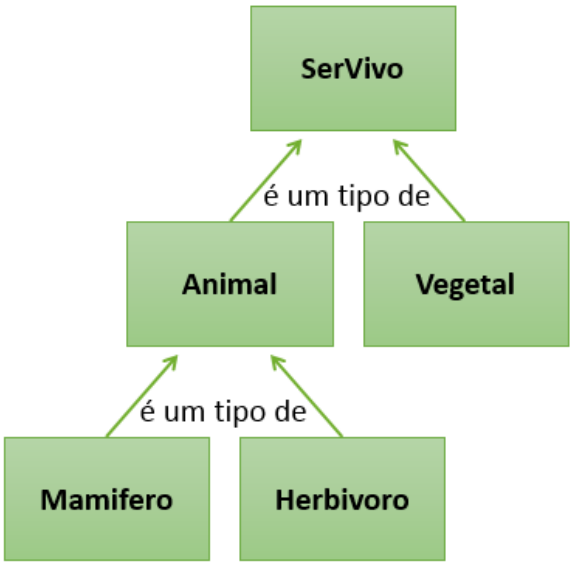

Figura 1 – Herança entre classes

A relação de herança entre uma superclasse e uma subclasse é uma relação “é um tipo de”, portanto, pela estrutura apresentada na Figura 1, um mamífero é um tipo de animal, que por sua vez é um tipo de ser vivo. Mas quais são as vantagens em se organizar os objetos hierarquicamente? A herança permite que classes compartilhem atributos e métodos, permitindo melhor organização e o reuso de código. Por exemplo: em aulas passadas implementamos a classe `Pessoa`, que possui atributos que são comuns a qualquer pessoa: `@nome`, `@idade`, `@altura` e `@peso`.

Contudo, uma pessoa é um elemento muito genérico e, portanto, podemos criar elementos mais específicos como, por exemplo, as classes `Aluno` e `Professor`, que são um tipo de pessoa, e, portanto, herdam da classe `Pessoa`. As subclasses `Aluno` e `Professor` herdam atributos e métodos da classe `Pessoa`, e dessa forma não precisamos definir os atributos `@nome`,

`@idade`, `@peso` e `@altura` nas classes `Aluno` e `Professor`, pois esses atributos já estão lá através da herança. Nas subclasses `Aluno` e `Professor` devemos apenas definir atributos e métodos que são específicos dessas classes. Por exemplo: na classe `Aluno` podemos definir o atributo `@matricula`, e na classe `Professor` podemos definir o atributo `@salario`.

Para melhor ilustrar como funciona a herança em Ruby, vamos utilizar o nosso exemplo do sistema bancário: vamos criar uma subclasse chamada `ContaEspecial`. As contas especiais possuem limite de crédito, que funciona como um "saldo extra" fornecido pelo banco. O cliente poderá utilizar o limite quando desejar, portanto, ao fazer uma transferência ou saque, o valor solicitado pelo cliente poderá ser, no máximo, o seu saldo somado ao limite de crédito. Por exemplo: uma conta especial possui R$ 2.000,00 reais de saldo e tem um limite de crédito de R$ 1.000,00 reais, portanto o cliente dessa conta poderá sacar até R$ 3.000,00 reais, que é a soma do saldo com o limite.

Para implementar a classe `ContaEspecial`, vamos herdar a classe `ContaBancaria`, que já possui métodos e atributos que poderão ser usados em `ContaEspecial`. Além disso, vamos criar um novo atributo na classe `ContaEspecial` chamado `@limite`, que irá guardar o limite de crédito da conta. Vamos redefinir o método `initialize` da classe `ContaEspecial` para receber um novo parâmetro (`limite`), e também vamos redefinir o método `tem_saldo?` Para implementar a nova forma de verificação de saldo da classe `ContaEspecial`, que deve levar em consideração o atributo `@limite`, conforme explicado no parágrafo anterior.

Observe a implementação da classe `ContaEspecial` no Exemplo de código 1.

```
 1  load 'ContaBancaria.rb'
 2
 3  class ContaEspecial < ContaBancaria
 4    attr_accessor :limite
 5
 6    def initialize(numero, saldo_inicial, limite, titular, agencia)
 7      @numero = numero
 8      @saldo = saldo_inicial
 9      @limite = limite
10      @titular = titular
11      @agencia = agencia
12    end
13
14    private
15
16    def tem_saldo?(valor)
17      return valor <= (@saldo + @limite)
18    end
19  end
```

Exemplo de código 1 – Implementação da classe ContaEspecial

Inicialmente observe como a implementação da classe `ContaEspecial` ficou enxuta. Você pode estar se perguntando por que essa classe não tem os métodos `sacar`, `depositar` e `transferir`, e a resposta dessa pergunta é simples: ela tem esses métodos! E isso só é possível graças à herança: quando definimos que a classe `ContaEspecial` herda de `ContaBancaria`

(linha 3), os atributos e os métodos da classe `ContaBancaria` foram herdados pela classe `ContaEspecial`, e portanto os métodos `sacar`, `depositar` e `transferir` também estão disponíveis na classe `ContaEspecial`.

Na linha 1, importamos o *script* `ContaBancaria.rb` para poder utilizar a classe `ContaBancaria`. Isso só é necessário caso essas duas classes tenham sido implementadas em *scripts* diferentes. Em seguida, na linha 3, iniciamos a definição da classe `ContaEspecial` herdando da classe `ContaBancaria`. Para isso, bastou usar `ContaEspecial < ContaBancaria`. Isso indica que a classe `ContaEspecial` deve herdar atributos e métodos da classe `ContaBancaria`. Na linha 4, usamos o `attr_accessor` para fornecer métodos de acesso e manipulação do atributo `@limite`. Entre as linhas 6 e 12, redefinimos o método `initialize` para receber um novo parâmetro (`limite`), e finalmente entre as linhas 16 e 18, redefinimos o método `tem_saldo?` para implementar a nova forma de verificação do saldo, que leva em consideração o atributo `@limite`.

Para testar a implementação da classe `ContaEspecial`, use o Exemplo de código 2.

```
1  conta = ContaEspecial.new(1, 3000, 2000, "Pedro Silva", 2020)
2  conta.sacar(200)
3  puts "R$ #{conta.saldo}"
4  conta.sacar(3800)
5  puts "R$ #{conta.saldo}"
```

Exemplo de código 2 – Testando a classe ContaEspecial

O resultado esperado após a execução do Exemplo de código 2 é apresentado abaixo.

```
R$ 2800
R$ -1000
```

Atividade 16.1

Implemente a classe `ContaUniversitaria` que herda da classe `ContaBancaria`. As contas universitárias possuem um limite para saques e transferências. Por exemplo: considere uma conta universitária com R$ 1.000 reais de saldo e um limite de R$ 300 reais. O cliente dessa conta só poderá sacar e transferir até R$ 300 reais, mesmo que a conta dele possua saldo para valores maiores.

No Exemplo de código 1, os métodos `initialize` e `tem_saldo?`, definidos na superclasse `ContaBancaria`, foram redefinidos na classe `ContaEspecial`. Quando uma subclasse reimplementa (redefine) um método da superclasse dizemos que houve uma sobrescrita de métodos. Sobrescrever métodos é uma técnica importante da Programação Orientada a Objetos que nos permite redefinir o comportamento de métodos das superclasses nas subclasses.

Algo não ficou muito bom na implementação da classe `ContaEspecial`: o método `initialize` da classe `ContaEspecial` ficou quase idêntico ao método `initialize` da classe `ContaBancaria`. Conforme já aprendemos, devemos sempre favorecer o reuso de código, e não ter código repetido em diferentes locais. Para resolver esse problema vamos usar o `super`. O `super` é uma palavra-chave que nos permite chamar o método da superclasse na subclasse. Para melhor ilustrar a utilidade do `super`, vamos reescrever o método `initialize` da classe `ContaEspecial`, conforme ilustrado no Exemplo de código 3.

```
1  def initialize(numero, saldo_inicial, limite, titular, agencia)
2    super(numero, saldo_inicial, titular, agencia)
3    @limite = limite
4  end
```

Exemplo de código 3 – Método initialize da classe ContaEspecial

No Exemplo de código 3, reescrevemos o método `initialize` para usar o `super`. Nesse caso, o `super` irá chamar o método `initialize` da superclasse (`ContaBancaria`) passando os parâmetros `numero`, `saldo_inicial`, `titular` e `agencia`. Em seguida, o parâmetro `limite` é atribuído ao `@limite`. Observe como a implementação do método `initialize` ficou bem mais limpa e sem repetição de código.

Atividade 16.2

Implemente o método `initialize` na classe `ContaUniversitaria` para usar o `super`.

Ruby utiliza herança simples (ou herança única), o que significa que uma subclasse só pode herdar de uma única superclasse, e uma superclasse pode ser herdada por várias subclasses. A outra forma de herança é a múltipla, em que uma subclasse pode ter mais de uma superclasse. Contudo, há uma forma de simular herança múltipla em Ruby utilizando módulos e *mixins*.

2. Módulos

Os módulos são mecanismos que permitem agrupar métodos, classes e constantes, além de fornecer *namespaces*, previnem o conflito de nomes (de classes, métodos, etc.) e permitem o uso de *mixins*. O Exemplo de código 4 ilustra a sintaxe para definição de um módulo.

```
1  module <nome_do_modulo>
2    # Código do módulo
3  end
```

Exemplo de código 4 – Sintaxe do módulo

O Exemplo de código 5 ilustra a definição do módulo `CalculadoraComplexa`, que possui a constante `PI` com valor `3.14`, além de dois métodos (`potencia` e `fatorial`).

```
 1  module CalculadoraComplexa
 2    PI = 3.14
 3
 4    def potencia(base, expoente)
 5      return base ** expoente
 6    end
 7
 8    def fatorial(numero)
 9      fatorial = 1
10      for i in (1..numero)
11        fatorial = fatorial * i
12      end
13      return fatorial
14    end
15  end
```

Exemplo de código 5 – Módulo CalculadoraComplexa

O Exemplo de código 6 ilustra como utilizar o módulo `CalculadoraComplexa` apresentado no Exemplo de código 5.

```
1  load 'CalculadoraComplexa.rb'
2  include CalculadoraComplexa
3
4  puts potencia(2, 3)
5  puts PI
```

Exemplo de código 6 – Usando o módulo CalculadoraComplexa

Na linha 1, usamos o `load 'CalculadoraComplexa.rb'` para carregar o código de outro *script*. Em seguida, na linha 2, usamos o `include` para importar o módulo `CalculadoraComplexa` definido no *script* 'CalculadoraComplexa.rb'. O uso do `include CalculadoraComplexa` é necessário para que possamos utilizar as variáveis/constantes e métodos definidos no módulo.

3. Mixins

Os *mixins* são módulos incluídos em classes. Isso significa que podemos definir um módulo com constantes e métodos, e em seguida incluir esse módulo numa classe, fazendo com que essa classe passe a ter os métodos e constantes definidos no módulo. Para melhor ilustrar os *mixins*, observe o Exemplo de código 7, que ilustra a classe `Calculadora` com o módulo `CalculadoraComplexo` incluído.

```
 1  load 'CalculadoraComplexa.rb'
 2
 3  class Calculadora
 4    include CalculadoraComplexa
 5
 6    def somar(a, b)
 7      return a + b
 8    end
 9
10    def subtrair(a, b)
11      return a - b
12    end
13  end
```

Exemplo de código 7 – Classe Calculadora com o módulo CalculadoraComplexa

No Exemplo de código 7, iniciamos carregando o código do *script* 'CalculadoraComplexa.rb', no qual foi definido o módulo `CalculadoraComplexa`. Em seguida, na linha 3, iniciamos a definição da classe `Calculadora`, e na linha 4 usamos o `include CalculadoraComplexa` para incluir o módulo `CalculadoraComplexa` na classe `Calculadora`, permitindo que a classe utilize os métodos e constantes definidos no módulo.

```
1  load 'Calculadora.rb'
2
3  calc = Calculadora.new
4
5  puts calc.somar(2, 3)
6  puts calc.fatorial(3)
```

Exemplo de código 8 – Testando a classe Calculadora com os métodos definidos no módulo

Observe, no Exemplo de código 8, que criamos um objeto da classe `Calculadora` (linha 3) e usamos o método `somar` (linha 5), que foi definido na classe `Calculadora`, mas também utilizamos o método `fatorial` (linha 6), que foi definido no módulo e incluído na classe através de um *mixin*.

Leitura complementar

Confira o artigo escrito por Carlos Brando, do blog Nome do Jogo, que explica as semelhanças e diferenças entre classes e módulos em Ruby:
http://nomedojogo.com/2009/12/24/classes-sao-modulos-no-ruby/

Resumindo

Essa aula apresentou importantes conceitos da Programação Orientada a Objetos. Iniciamos aprendendo sobre herança, que permite que organizemos nossas classes hierarquicamente. Em seguida, discutimos como organizar classes, métodos e constantes em módulos, além de como utilizá-los para fazer *mixins* com classes, simulando herança múltipla.

Referências

BRANDO, C. Classes são Módulos no Ruby. **Nome do Jogo**, 2009. Disponivel em: <http://nomedojogo.com/2009/12/24/classes-sao-modulos-no-ruby/>. Acesso em: 16 mar. 2016.

CARLOS, G. T. Ruby - Módulos e Mixins. **Show the Code**, 2014. Disponivel em: <http://www.showthecode.com.br/2014/06/modulos-e-mixins.html>. Acesso em: 16 mar. 2016.

GURU-SP. Módulos e Mixins. **Tutorial de Ruby do GURU-SP**. Disponivel em: <http://guru-sp.github.io/tutorial_ruby/modulos-mixins.html>. Acesso em: 16 mar. 2016.

POINT, T. Ruby Tutorial. **Tutorials Point**, 2015. Disponivel em: <http://www.tutorialspoint.com/ruby/>. Acesso em: 12 nov. 2015.

RANGEL, E. **Conhecendo Ruby**. [S.l.]: Leanpub, 2014.

SOUZA, L. **Ruby - Aprenda a programar na linguagem mais divertida**. 1ª. ed. São Paulo: Casa do Código, v. I, 2012.

Apêndice

Apêndice A – Implementação completa da classe ContaEspecial

```
 1  load 'ContaBancaria.rb'
 2
 3  class ContaEspecial < ContaBancaria
 4    attr_accessor :limite
 5
 6    def initialize(numero, saldo_inicial, limite, titular, agencia)
 7      super(numero, saldo_inicial, titular, agencia)
 8      @limite = limite
 9    end
10
11    private
12
13    def tem_saldo?(valor)
14      return valor <= (@saldo + @limite)
15    end
16  end
```