Aula 15

Programação Orientada a Objetos: Atributos, Métodos e Importação

Prof. Jalerson Lima

Apresentação

Nessa aula vamos nos aprofundar um pouco mais nos conceitos e técnicas da Programação Orientada a Objetos, discutindo mais detalhes sobre atributos e métodos. Também iremos aprender como separar e importar código para evitar que *scripts* Ruby fiquem muito extensos e complicados.

Objetivos

1. Compreender o que significa e como funciona a visibilidade de métodos;
2. Aprender como definir a visibilidade de métodos usando modificadores de acesso;
3. Compreender o que são e como definir métodos de classe;
4. Compreender como compor objetos e partir de outros objetos.

1. Métodos

Vamos dar início a nossa aula aprendendo um pouco mais sobre métodos, discutindo sobre visibilidade, bem como o que são e como funcionam os métodos de classe.

1.1 Visibilidade

A visibilidade de um método define quem pode chama-lo (executá-lo). Em grande parte das linguagens orientadas a objeto, inclusive Ruby, existem três tipos de visibilidade: público (*public*), protegido (*protected*) e privado (*private*). Contudo, o funcionamento de cada visibilidade é um pouco diferente em Ruby, conforme vamos ver a seguir.

1.1.1 Visibilidade em Ruby

A visibilidade pública (*public*) é a mais permissiva, pois um método com visibilidade pública pode ser chamado em qualquer lugar. Todos os métodos que criamos até o presente momento tinham visibilidade pública.

Métodos com visibilidade privada podem ser chamados pela própria classe em que foram definidos e pelas suas subclasses. Ainda não aprendemos o que são subclasses porque ainda não estudamos herança, que também é um conceito da Programação Orientada a Objetos, e que será abordado numa futura aula. Em Ruby, a visibilidade protegida é subutilizada porque ela é quase idêntica ao privado.

1.1.2 Visibilidade em outras linguagens

Em outras linguagens de programação, a visibilidade pública funciona exatamente da mesma forma que em Ruby, portanto métodos públicos podem ser chamados em qualquer lugar. Métodos com visibilidade protegida podem ser chamado dentro da própria classe onde ele foi definido e pelas subclasses. A visibilidade privada é a mais limitada, pois métodos com essa visibilidade só podem ser chamados dentro da própria classe onde eles foram definidos.

1.1.3 Modificadores de acesso

Agora que aprendemos conceitualmente como funcionam cada uma das visibilidades, vamos estudar como defini-las em código. Para determinar se um método terá visibilidade pública, protegida ou privada, nós devemos usar os modificadores de acesso, que são: `public`, `protected` e `private`. Observe o Exemplo de código 1.

```
 1  class MinhaClasse
 2
 3    protected # Os métodos abaixo serão protegidos
 4
 5    def metodo_protegido
 6    end
 7
 8    private # Os métodos abaixo são privados
 9
10    def metodo_privado
11    end
```

```
12
13  end
```

Exemplo de código 1 – Definindo a visibilidade dos métodos

Observe, na linha 3, que usamos o modificador de acesso `protected`. Todos os métodos definidos após essa linha terão visibilidade protegida. Observe, na linha 8, que usamos o modificador de acesso `private`, portanto todos os métodos definidos após essa linha serão privados. E os métodos públicos? Quando você define um método numa classe, por padrão ele será público, exceto se você colocar algum modificador de acesso antes dele (`protected` ou `private`). Você também pode, se quiser, usar o modificador de acesso `public` para definir métodos públicos.

Para melhor ilustrar a visibilidade privada, vamos utilizar a nossa classe `ContaBancaria`. Nela existem dois métodos (`transferir` e `sacar`) que precisam verificar se há saldo suficiente na conta antes de realizar a operação. Nós podemos isolar a verificação do saldo num método privado chamado `tem_saldo?`, que recebe um valor como parâmetro e verifica se esse valor é menor ou igual ao `@saldo`. Esse método pode ter visibilidade privada porque não há necessidade de ele ser chamado externamente. Observe como funcionaria isso no Exemplo de código 2, que está omitindo parte da implementação da classe para poupar espaço.

```
 1  class ContaBancaria
 2    # Omitindo parte da implementação para poupar espaço
 3
 4    def sacar(valor)
 5      if tem_saldo?(valor) # Chamada do método tem_saldo?
 6        @saldo = @saldo - valor
 7        return true
 8      else
 9        return false
10      end
11    end
12
13    def transferir(valor, conta_destino)
14      if tem_saldo?(valor) # Chamada do método tem_saldo?
15        @saldo = @saldo - valor
16        conta_destino.saldo = conta_destino.saldo + valor
17        return true
18      else
19        return false
20      end
21    end
22
23    private # Modificador de acesso privado
24
25    def tem_saldo?(valor) # Definição do método privado tem_saldo?
26      return valor <= @saldo
27    end
28
```

```
29  end
```

Exemplo de código 2 – Exemplo de método com visibilidade privada

Observe, nas linhas 5 e 14, que alteramos a implementação dos métodos `sacar` e `transferir` para usar o método `tem_saldo?`. Na linha 23, usamos o modificador de acesso `private` para que todos os métodos definidos a seguir sejam privados. Em seguida, entre as linhas 25 e 27, implementamos o método `tem_saldo?`.

1.2 Métodos de classes

Conforme estudamos anteriormente, os métodos definem ações que podem ser executadas por objetos. Contudo, algumas ações só fazem sentido se vinculadas ao coletivo (conjunto) e não a instâncias particulares (objetos). Portanto, métodos de classe são ações que não são executadas sobre objetos, mas sim sobre as classes. Observe o Exemplo de código 3, que ilustra como definir um método de classe.

```
1  class MinhaClasse
2
3    def self.metodo_de_classe
4      # Implementação do método
5    end
6
7  end
```

Exemplo de código 3 – Exemplo de método de classe

Observe, na linha 3, que usamos `self.` antes do nome do método, e isso determina que esse é um método de classe e não um método de instância. Observe o Exemplo de código 4, que ilustra como chamar o método `metodo_de_classe` da classe `MinhaClasse`.

```
1  MinhaClasse.metodo_de_classe
```

Exemplo de código 4 – Chamando um método de classe

Observe, no Exemplo de código 4, que não estamos executando o método `metodo_de_classe` sobre uma variável que guarda um objeto, mas sim sobre a classe `MinhaClasse`. Para melhor ilustrar a implementação e chamada de métodos de classe, observe o Exemplo de código 5, que ilustra a classe `Calculadora`, que possui dois métodos de classe: `somar` e `subtrair`.

```
 1  class Calculadora
 2
 3    def self.somar(a, b)
 4      return a + b
 5    end
 6
 7    def self.subtrair(a, b)
 8      return a - b
 9    end
10
11  end
12
13  puts Calculadora.somar(5, 2)
14  puts Calculadora.subtrair(6, 3)
```

Exemplo de código 5 – Exemplo de implementação e chamada de métodos de classe

Observe, nas linhas 3 e 7, que os métodos `somar` e `subtrair` foram definidos com o `self.` na frente de seus nomes, portanto eles são métodos de classe e não de objetos. As linhas 13 e 14 ilustram a chamada dos métodos de classe da classe `Calculadora`. Observe que os métodos `somar` e `subtrair` foram chamados sobre a classe `Calculadora`, e não sobre objetos dessa classe. Você pode estar se perguntando se os métodos `somar` e `subtrair` não poderiam ser métodos de instância, e a resposta é sim. A intenção do exemplo da classe `Calculadora` é apenas ilustrar a definição e chamada dos métodos de classe.

Métodos de classe não podem ser chamados sobre objetos. Observe o Exemplo de código 6, que cria um objeto da classe `Calculadora` (linha 1) e tenta chamar o método `somar` (linha 2) sobre o objeto guardado em `calc`.

```
1  calc = Calculadora.new
2  puts calc.somar(5, 3)
```

Exemplo de código 6 – Chamando um método de classe sobre um objeto

Ao tentar executar o código do Exemplo de código 6, nos deparamos com o erro abaixo, que diz que o método `somar` não foi definido na classe `Calculadora`.

```
exemplo166.rb:14:in `<main>': undefined method `somar' for
#<Calculadora:0x00000002e3c7c0> (NoMethodError)
```

Atividade 15.1

Implemente mais dois métodos de classe na classe `Calculadora`: `multiplicar`, que recebe dois parâmetros, multiplica-os e retorna o resultado; e `dividir`, que também recebe dois parâmetros, divide-os e retorna o resultado.

2. Atributos

Vamos dar continuidade a nossa aula discutindo um pouco mais sobre os atributos. Vamos aprender o que são e como funcionam os atributos de classe, bem como trabalhar um pouco com composição de objetos.

2.1 Atributos de classe

Na aula passada nós estudamos que os atributos são definidos nas classes e eles irão compor as características dos objetos dessa classe, portanto, ao criar um objeto de uma classe, esse objeto terá os atributos definidos na classe. É por isso que chamamos esses atributos de atributos de objetos ou atributos de instância, porque eles estão vinculados aos objetos (instâncias). Contudo, existe um outro tipo de atributo chamado atributo de classe, que não fica vinculado aos objetos, mas sim às classes.

Para melhor ilustrar a diferença entre atributos de objetos e atributos de classe, vamos a alguns exemplos.

```
1  class Pessoa
2    def initialize(nome)
3      @nome = nome
4    end
5  end
```

Exemplo de código 7 – Classe Pessoa com um atributo de objeto

O Exemplo de código 7 ilustra a classe `Pessoa` com um atributo de objeto chamado `@nome`. Cada pessoa tem um nome próprio, portanto faz sentido que o nome da pessoa seja um atributo de objeto, para que cada objeto possa ter seu próprio nome.

Contudo, existem algumas características que valem para um conjunto e não para as instâncias. Por exemplo: atualmente uma pessoa é considerada maior de idade quanto atinge os 18 anos. Essa é uma característica que vale para todas as pessoas, e, portanto, não faz sentido que um atributo `@maioridade` fique associado a cada objeto. Para esses casos existem os atributos de classe. Observe o Exemplo de código 8, que ilustra a classe `Pessoa` com o atributo de classe `@@maioridade` (com dois arrobas).

```
1  class Pessoa
2    @@maioridade = 18
3    attr_accessor :nome
4    attr_accessor :idade
5
6    def initialize(nome, idade)
7      @nome = nome
8      @idade = idade
9    end
10
11 end
```

Exemplo de código 8 – Classe Pessoa com um atributo de classe

Para definir que um atributo é de classe e não de objeto, usamos dois arrobas (`@@`) antes do nome do atributo, conforme fizemos na linha 2 do Exemplo de código 8. Agora observe, no Exemplo de código 9, a implementação do método `maioridade?`, que usa o atributo de classe `@@maioridade` para verificar se a pessoa é maior de idade ou não.

```
1  class Pessoa
2    @@maioridade = 18
3    attr_accessor :nome
4    attr_accessor :idade
5
6    def initialize(nome, idade)
7      @nome = nome
8      @idade = idade
9    end
10
11   def maioridade?
12     return @idade >= @@maioridade
13   end
14
15 end
```

Exemplo de código 9 – Acessando o atributo de classe dentro da própria classe

Observe o Exemplo de código 10, que ilustra a chamada do método `maioridade?`. A execução desse código deve resultar em `true`, pois o objeto guardado na variável `pessoa` tem idade superior a 18 anos.

```
1  # Omitindo a implementação da classe Pessoa
2
3  pessoa = Pessoa.new("Pedro", 30)
4  puts pessoa.maioridade?
```

Exemplo de código 10 – Testando o método maioridade?

Apesar de ser possível acessar o atributo `@@maioridade` de dentro da própria classe, ainda não conseguimos acessá-lo externamente. Para ilustrar isso, observe o Exemplo de código 11.

```
1  # Omitindo a implementação da classe Pessoa
2  puts Pessoa.maioridade
```

Exemplo de código 11 – Tentando acessar o atributo de classe externamente

Ao tentar executar o Exemplo de código 11, nos deparamos com a seguinte mensagem de erro.

```
Exemplo1611.rb:19:in `<main>': undefined method `maioridade' for Pessoa:Class
(NoMethodError)
```

O erro indica que não existe um método chamado `maioridade` na classe `Pessoa`. De fato, nós não podemos acessar atributos de classe externamente sem ter um método que nos dê acesso a ele. Portanto precisamos implementar o método `maioridade` que nos dê acesso ao atributo `@@maioridade`. Como `@@maioridade` é um atributo de classe, faz mais sentido que o método `maioridade` também seja um método de classe. Observe, entre as linhas 15 e 17 do Exemplo de código 12, a implementação do método de classe `maioridade`.

```
 1  class Pessoa
 2    @@maioridade = 18
 3    attr_accessor :nome
 4    attr_accessor :idade
 5
 6    def initialize(nome, idade)
 7      @nome = nome
 8      @idade = idade
 9    end
10
11    def maioridade?
12      return @idade >= @@maioridade
13    end
14
15    def self.maioridade
16      return @@maioridade
17    end
18
19  end
```

Exemplo de código 12 – Classe Pessoa com o método de classe maioridade

Com isso, já podemos executar o Exemplo de código 11 sem problemas.

2.2 Composição de objetos

Atributos de classe ou de instâncias podem ser compostos por outros objetos, e quando isso ocorre dizemos que está havendo uma composição de objetos. Para ilustrar isso, vamos adicionar um atributo de instância chamado `titular` na classe `ContaBancaria`, que irá guardar um objeto da classe `Pessoa` para representar o titular da conta bancária. Observe o Exemplo de código 13.

```
 1  class ContaBancaria
 2    attr_accessor :saldo
 3    attr_accessor :numero
 4    attr_accessor :titular
 5
 6    def initialize(numero, saldo_inicial, titular)
 7      @numero = numero
 8      @saldo = saldo_inicial
 9      @titular = titular
10    end
11
12    # Omitindo o resto da implementação da classe
13
14  end
```

Exemplo de código 13 – Adicionando o atributo titular na classe

No Exemplo de código 13, modificamos o método `initialize` para receber um novo parâmetro (`titular`) e atribuí-lo ao `@titular`, que é o novo atributo dessa classe. Além disso, adicionamos o `attr_accessor :titular` (linha 4) para fornecer acesso externo para esse atributo.

Usando a classe `Pessoa` ilustrada no Exemplo de código 12 e a classe `ContaBancaria` com a modificação apresentada no Exemplo de código 13, vamos usar um objeto da classe `Pessoa` para compor o atributo `@titular` da classe `ContaBancaria`. Observe o Exemplo de código 14.

```
1  # Omitindo implementação da classe ContaBancaria
2  # Omitindo implementação da classe Pessoa
3
4  pessoa = Pessoa.new("Pedro", 30)
5  conta = ContaBancaria.new(1, 1000, pessoa)
6  puts conta.titular.nome
```

Exemplo de código 14 – Exemplo de composição de objetos

No Exemplo de código 14, criamos um objeto da classe `Pessoa` (linha 4) e em seguida criamos um objeto da classe `ContaBancaria` (linha 5) passando o objeto da classe `Pessoa` como parâmetro. Esse objeto será atribuído a `@titular` no método `initialize` da classe `ContaBancaria`, de forma que, na linha 6, podemos chamar o método `titular` na variável `conta` para acessar o atributo `@titular` desse objeto. O resultado esperado na execução desse código é o nome do titular da conta, `"Pedro"`.

Vamos aperfeiçoar um pouco mais o nosso sistema bancário? Sabemos que as contas bancárias pertencem às agências bancárias, e cada agência possui um número. Portanto precisamos implementar a classe `AgenciaBancaria`, que precisa ter, no mínimo, três atributos: `@numero`, que guarda o número da agência; `@contas`, que irá armazenar as contas dessa agência, e `@nome`, que irá guardar o nome da agência. Além disso, a classe `AgenciaBancaria` precisa ter métodos que permitam localizar e manipular as contas bancárias da agência. Observe o Exemplo de código 15, que ilustra a implementação completa da classe `AgenciaBancaria`.

```
 1  class AgenciaBancaria
 2    attr_accessor :numero
 3    attr_accessor :nome
 4    attr_accessor :contas
 5
 6    def initialize(numero, nome)
 7      @contas = Hash.new
 8      @nome = nome
 9      @numero = numero
10    end
11
12    def adicionar_conta(conta)
13      return @contas[conta.numero] = conta
14    end
15
16    def remover_conta(numero)
17      if @contas.has_key?(numero)
18        @contas.delete(numero)
19        return true
20      else
21        return false
22      end
23    end
24
25    def localizar_conta(numero)
26      return @contas[numero]
27    end
28
29  end
```

Exemplo de código 15 – Implementação da classe AgenciaBancaria

A classe `AgenciaBancaria` possui três atributos: `@numero`, `@nome` e `@contas`. O atributo `@contas` é um *Hash* que irá armazenar as contas bancárias da agência. Também poderíamos ter usado um *Array*, contudo um *Hash* vai facilitar a localização de contas bancárias pois usaremos o número da conta como chave.

O método `adicionar_conta` (linhas 12 a 14) recebe um objeto da classe `ContaBancaria` como parâmetro que é adicionado ao *Hash* `@contas` usando o número da conta como chave. O método `remover_conta` (linhas 16 a 23) recebe um número de conta como parâmetro, verifica se esse número está contido no *Hash* `@contas` (linha 17). Se existir, a conta é removida do *Hash* na linha 18 e retorna `true`. Caso contrário, simplesmente retorna `false` (linha 21). O método `localizar_conta` recebe um número de conta como parâmetro, que é usado como chave no *Hash* `@contas` para retornar a conta com o número fornecido.

Seria interessante também modificar a classe `ContaBancaria` para incluir o atributo `@agencia`. Dessa forma os objetos dessa classe teriam uma referência para a sua agência. Observe o Exemplo de código 16.

```
 1  class ContaBancaria
 2    attr_accessor :saldo
 3    attr_accessor :numero
 4    attr_accessor :titular
 5    attr_accessor :agencia
 6
 7    def initialize(numero, saldo_inicial, titular, agencia)
 8      @numero = numero
 9      @saldo = saldo_inicial
10      @titular = titular
11      @agencia = agencia
12    end
13
14    # Omitindo o resto da implementação da classe
15  end
```

Exemplo de código 16 – Classe ContaBancaria com o atributo @agencia

Novamente bastou modificar o método `initialize` (linha 7) para receber um novo parâmetro (`agencia`) e associa-lo ao atributo `@agencia` (linha 11). Além disso, usamos o `attr_accessor :agencia` para fornecer os métodos de acesso a esse atributo.

Agora podemos testar as nossas classes `Pessoa`, `AgenciaBancaria` e `ContaBancaria`. Para isso, observe o Exemplo de código 17.

```
 1  # Omitindo a implementação das classes
 2
 3  pedro = Pessoa.new("Pedro", 30)
 4  maria = Pessoa.new("Maria", 28)
 5
 6  agencia = AgenciaBancaria.new(10, "Agência Câmara Cascudo")
 7
 8  conta1 = ContaBancaria.new(1, 1000, pedro, agencia)
 9  conta2 = ContaBancaria.new(2, 1500, maria, agencia)
10
11  agencia.adicionar_conta(conta1)
12  agencia.adicionar_conta(conta2)
13
14  puts conta1.titular.nome
15  puts conta1.agencia.nome
16
17  conta2 = agencia.localizar_conta(2)
18  puts conta2.saldo
```

Exemplo de código 17 – Testando as classes Pessoa, AgenciaBancaria e ContaBancaria

Nas linhas 3 e 4, criamos dois objetos da classe `Pessoa`. Em seguida, na linha 6, criamos um objeto da classe `AgenciaBancaria`. Nas linhas 8 e 9, criamos dois objetos da classe `ContaBancaria`, passando os objetos da classe `Pessoa` e `AgenciaBancaria` como parâmetro. Nas linhas 11 e 12, as contas foram adicionadas na agência. Em seguida, nas linhas 14 e 15,

acessamos os atributos `@titular` e `@agencia` da `conta1`. Na linha 17, usamos o método `localizar_conta` da classe `AgenciaBancaria` para recuperar a conta de número `2`.

Atividade 15.2

Modifique a classe `AgenciaBancaria` para incluir um atributo de instância, chamado `@gerente`, que irá armazenar um objeto da classe `Pessoa`, que irá representar o gerente da agência. Crie métodos que permitam acessar o atributo `@gerente` externamente.

Atividade 15.3

Crie a classe `Banco` com um atributo de instância chamado `@agencias`, que irá armazenar objetos da classe `AgenciaBancaria`. Crie também métodos que permitam adicionar, remover e localizar agências bancárias, conforme foi feito com a classe `AgenciaBancaria` no Exemplo de código 15.

3. Importando código

Você deve estar percebendo que escrever todo o código dos nossos exemplos num único *script* Ruby está deixando o código muito extenso e complicado. Observe que, para executar o Exemplo de código 17, precisamos implementar as classes `Pessoa`, `AgenciaBancaria` e `ContaBancaria`, além do trecho de código que usa essas classes. Não seria mais interessante que pudéssemos definir cada classe num *script* Ruby separado? Pois é exatamente isso que vamos fazer: implementaremos cada classe num *script* Ruby separado, e importaremos essas classes num outro *script* para usar as classes.

Para começar, vamos colocar cada classe (`Pessoa`, `AgenciaBancaria` e `ContaBancaria`) num *script* Ruby separado. Coloque o código da classe `Pessoa` (Exemplo de código 12) num *script* chamado "Pessoa.rb" (sem aspas). Coloque o código da classe `AgenciaBancaria` (Exemplo de código 15) num *script* chamado "AgenciaBancaria.rb" (sem aspas). Coloque o código da classe `ContaBancaria` (vide Apêndice A no final da aula) num *script* chamado "ContaBancaria.rb" (sem aspas).

Feito isso, crie um outro *script*, chamado "teste.rb" (sem aspas) com o código do Exemplo de código 17, mas sem a implementação das classes `Pessoa`, `ContaBancaria` e `AgenciaBancaria`, pois essas classes serão importadas dos *scripts* que acabamos de criar. Existem duas formas de fazer essa importação, que serão detalhadas a seguir.

3.1 Importando código com load

A primeira forma de importar código é usando o método `load`, conforme ilustrado abaixo.

```
load '<nome-do-arquivo>.rb'
```

Para importar código de um *script* com o `load`, coloque-o no mesmo diretório do *script* que você está programando e em seguida use o nome do *script* a ser importado com a extensão `.rb`. Vale salientar que é preciso ter cuidado para não usar o `load` mais de uma vez para importar o mesmo *script*. Caso isso ocorra, o `load` irá importar o *script* mais de uma vez e isso poderá gerar problemas. Digite o Exemplo de código 18 no *script* "teste.rb" para importar as classes `ContaBancaria`, `AgenciaBancaria` e `Pessoa` usando o `load`. Lembre-se que os scripts "teste.rb", "AgenciaBancaria.rb", "ContaBancaria.rb" e "Pessoa.rb" precisam estar no mesmo diretório.

```
 1  load 'ContaBancaria.rb'
 2  load 'AgenciaBancaria.rb'
 3  load 'Pessoa.rb'
 4
 5  pedro = Pessoa.new("Pedro", 30)
 6  maria = Pessoa.new("Maria", 28)
 7
 8  agencia = AgenciaBancaria.new(10, "Agência Câmara Cascudo")
 9
10  conta1 = ContaBancaria.new(1, 1000, pedro, agencia)
11  conta2 = ContaBancaria.new(2, 1500, maria, agencia)
12
13  agencia.adicionar_conta(conta1)
14  agencia.adicionar_conta(conta2)
15
16  puts conta1.titular.nome
17  puts conta1.agencia.nome
18
19  conta2 = agencia.localizar_conta(2)
20  puts conta2.saldo
```

Exemplo de código 18 – Importando código com load

3.2 Importando código com require

A segunda forma de importar código é usando o método `require`, conforme ilustrado abaixo.

```
require '<nome-do-arquivo>'
```

O `require` funciona de forma semelhante ao `load`, contudo com algumas pequenas diferenças: a primeira delas é que a extensão do arquivo não é necessária. A segunda é que, caso você use o `require` mais de uma vez para importar o mesmo *script*, ele importará apenas

uma vez, evitando problemas. A terceira diferença é que, para importar um *script* que está no mesmo diretório do script que estamos programando, devemos colocar `./` antes do nome do arquivo, conforme ilustrado a seguir.

```
require './Pessoa'
```

Observe o Exemplo de código 19, que ilustra como importar as classes `ContaBancaria`, `AgenciaBancaria` e `Pessoa` para o "teste.rb".

```
 1  require './ContaBancaria'
 2  require './AgenciaBancaria'
 3  require './Pessoa'
 4
 5  pedro = Pessoa.new("Pedro", 30)
 6  maria = Pessoa.new("Maria", 28)
 7
 8  agencia = AgenciaBancaria.new(10, "Agência Câmara Cascudo")
 9
10  conta1 = ContaBancaria.new(1, 1000, pedro, agencia)
11  conta2 = ContaBancaria.new(2, 1500, maria, agencia)
12
13  agencia.adicionar_conta(conta1)
14  agencia.adicionar_conta(conta2)
15
16  puts conta1.titular.nome
17  puts conta1.agencia.nome
18
19  conta2 = agencia.localizar_conta(2)
20  puts conta2.saldo
```

Exemplo de código 19 – Importando código com require

Atividade 15.4

Implemente as classes `Pessoa`, `ContaBancaria` e `AgenciaBancaria` em três *scripts* separados: "Pessoa.rb", "ContaBancaria.rb" e "AgenciaBancaria.rb". Implemente um *script* "teste.rb", com o código ilustrado no Exemplo de código 19, importando as referidas classes.

Leitura complementar

Confira o artigo escrito por Carlos Brando, do blog Nome do Jogo, que trata com mais detalhes sobre as diferenças entre os métodos de importação existente: http://nomedojogo.com/2010/01/07/entendendo-os-metodos-load-e-require-por-dentro/

Resumindo

Essa aula apresentou o que são e como funcionam os modificadores de acesso para alterar a visibilidade de métodos. Também aprendemos como construir métodos de classe, que não ficam vinculados a objetos, mais sim às classes onde eles são definidos. Compreendemos o que são e como funcionam os atributos de classes, e por fim estudamos a composição de objetos, que nos permite compor objetos de uma classe com objetos de outras.

Referências

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 ContaBancaria

```
 1  class ContaBancaria
 2    attr_accessor :saldo
 3    attr_accessor :numero
 4    attr_accessor :titular
 5    attr_accessor :agencia
 6
 7    def initialize(numero, saldo_inicial, titular, agencia)
 8      @numero = numero
 9      @saldo = saldo_inicial
10      @titular = titular
11      @agencia = agencia
12    end
13
14    def sacar(valor)
15      if tem_saldo?(valor)
16        @saldo = @saldo - valor
17        return true
18      else
19        return false
20      end
21    end
22
23    def depositar(valor)
24      @saldo = @saldo + valor
25      return true
26    end
27
28    def transferir(valor, conta_destino)
29      if tem_saldo?(valor)
30        @saldo = @saldo - valor
31        conta_destino.saldo = conta_destino.saldo + valor
32        return true
33      else
34        return false
35      end
36    end
37
38    private
39
40    def tem_saldo?(valor)
41      return valor <= @saldo
42    end
43
44  end
```