Orientação a Objetos com Ruby

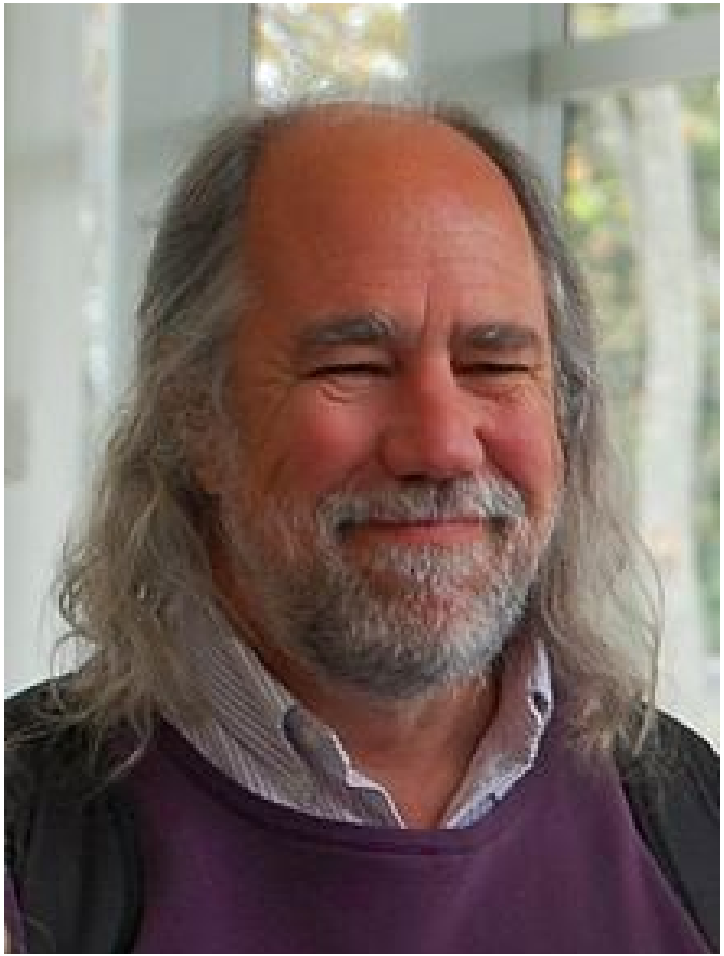
Arthur de Moura Del Esposte - esposte@ime.usp.br

Aula 03 - Duck Typing, Módulos e Exceções

Arthur de Moura Del Esposte - esposte@ime.usp.br

Grady Booch

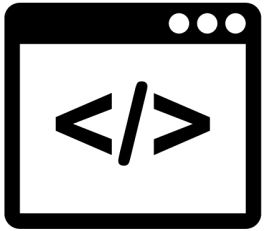
“Na **Análise Orientada a Objetos,** procuramos modelar o mundo identificando as **classes** e **objetos** que formam o **vocabulário do domínio do problema**, e no design orientado a objeto, inventamos as **abstrações e os mecanismos** que fornecem o comportamento que esse modelo requer” - **Object-oriented design: With Applications, (1991)**



Agenda

- Polimorfismo e *Duck Typing*
- Módulos e Mix-ins
- Tratamento de erros

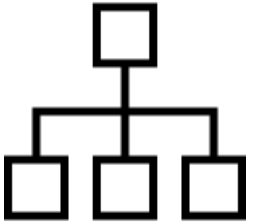
Exercício



- Crie uma classe chamada **Fraction** para representar uma fração matemática. Ela deve ter os seguintes métodos:
  - **Fraction#initalize** - Construtor para receber o numerador e denominador
  - **Fraction#to_f** - Método para conversão para Float
  - **Fraction#to_s** - Método que retorna um string com a fração
  - **Fraction#* -** Método que recebe outro objeto de Fração ou número inteiro e retorna uma novo objeto do tipo Fraction com o resultado da multiplicação
  - Métodos de acesso às variáveis de numerador e denominador

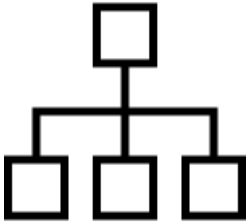
Orientação a Objetos: Polimorfismo

Polimorfismo



- O que é **Polimorfismo**?

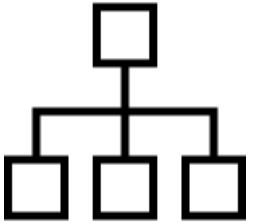
Polimorfismo



- Polimorfismo é um conceito fundamental em POO para obter comportamentos diferentes entre objetos de tipos variados usando-se a mesma interface
- Com **sobrecarga** de método, poderíamos ter várias implementações com a mesma assinatura cujo comportamento variasse de acordo com o parâmetro

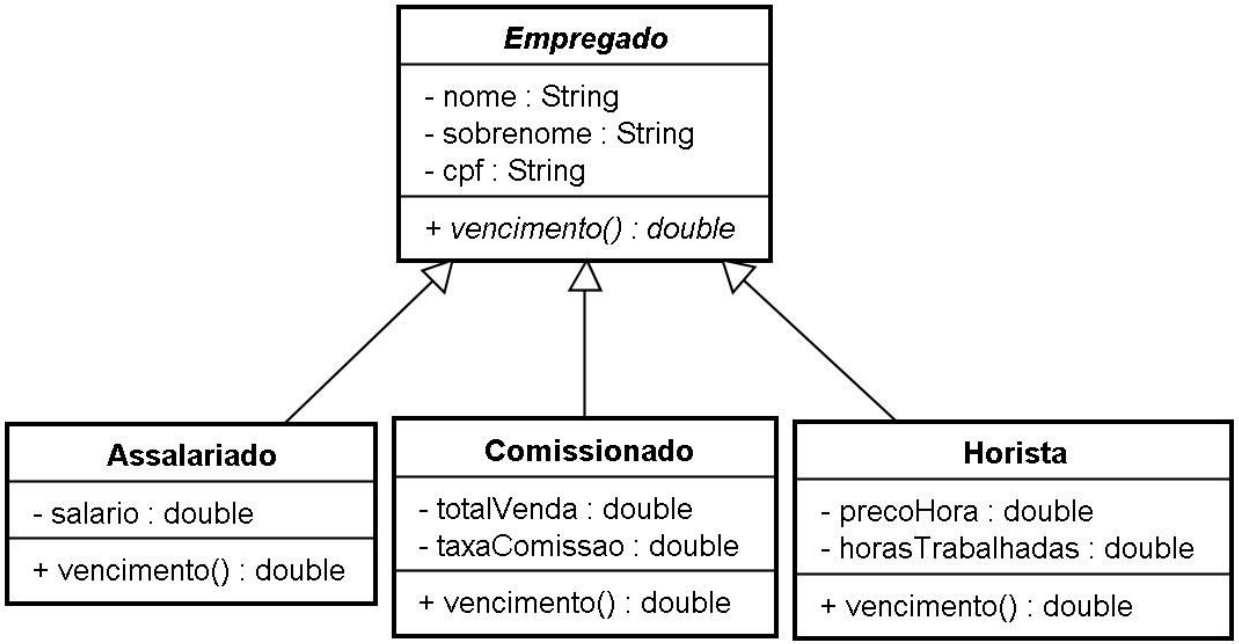

```
pessoa.reagir(new FilmeDeComedia);  // KKKKKKKKKKKK

pessoa.reagir(new FilmeDeTerror);  // Que medo!
```

Polimorfismo



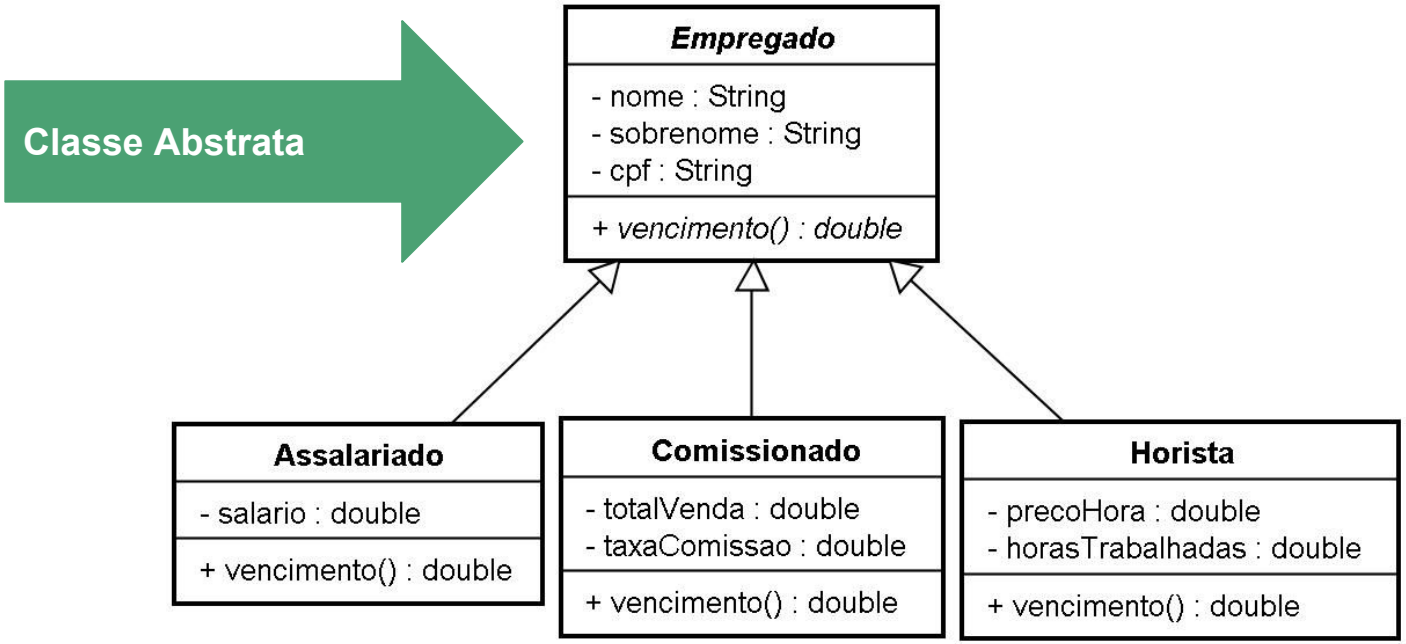
- Podemos tratar vários tipos diferentes de objetos da mesma forma
  - Usando herança
  - Usando interface
- O Polimorfismo substitui a necessidade de verificação de tipo. Por exemplo:
  - Se for um funcionário do tipo horista, então calcule o pagamento baseado nas horas de trabalho
  - Se for um funcionário do tipo assalariado, então calcule o pagamentocom seu salário fixo
  - Se for um funcionário do tipo comissionado, então calcule o pagamento baseado nas suas vendas

Polimorfismo - Exemplo em Java



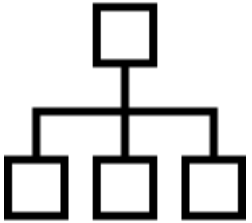

```
Empregado empregado = new Assalariado();

empregado.vencimento();
```

Polimorfismo - Exemplo em Java

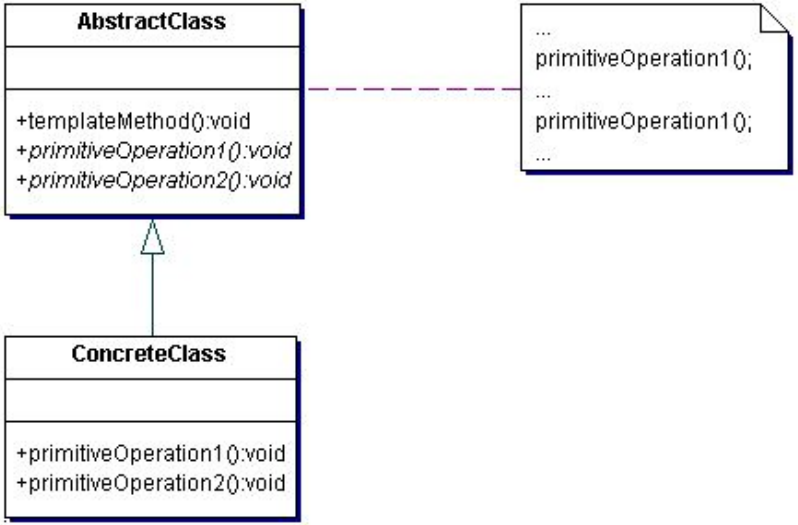



```
Empregado empregado = new Assalariado();

empregado.vencimento();
```

Classes Abstratas - Parte 1



- Não há o conceito de **Classes Abstratas** em Ruby
- Porém, podemos alcançar os mesmos objetivos de design (como **Polimorfismo**) de outras formas
- Veja o exemplo com o padrão de projeto **Template Method**



Classes Abstratas - Parte 2

```ruby
class Worker
  def work
    puts 'Start working'
    design_product
    prepare_material
    build
    puts "The #{product} is ready"
  end

  def design_product
    puts "Designing a #{product}"
  end
end
```

```ruby
class CivilEngineer < Worker
  def prepare_material
    puts "Define materials for the floor, walls and ceiling"
  end

  def build
    puts "Putting the bricks..."
  end

  def product
    "House"
  end
end
```

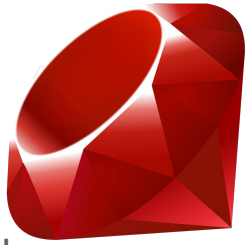
Classes Abstratas - Parte 3

- Outra estratégia é lançar uma **Exceção** no método que deve ser **obrigatoriamente sobrescrito**

```ruby
class Worker
  def work
    raise NotImplementedError.new("#{self.class.name}#work is an abstract method.")
  end
end
```

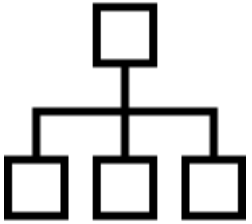
```ruby
class SoftwareEngineer < Worker
end
```

Polimorfismo em Ruby



- Já vimos que não existe sobrecarga de método, portanto esse tipo de polimorfismo está descartado =(
- Herança é bem mais utilizada para reuso de código e definição hierárquica do que para poliformismos baseado em tipos, afinal não temos declaração de tipos em Ruby
- O conceito de Interfaces não existem em Ruby também. Afinal não mantemos referências aos tipos de objetos e a verificação da interface de um método é realizada no momento que o método é chamado!

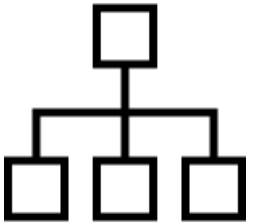
Polimorfismo e Duck Typing - Parte 1



- Em Ruby, duas classes diferentes não precisam compartilhar tipos para obtermos polimorfismo sobre seus métodos
- **Duck Typing** é uma forma de determinar a semântica válida de um objeto baseado no que ele pode fazer (seus métodos e propriedades), em vez de seu tipo (sua herança ou implementação de interface)
- Portanto, Ruby valoriza mais o que um objeto faz (capacidades) do que o ele é (seu tipo/classe). Por isso o uso do método **respond_to?**

“Se anda como um pato e faz barulho como um pato, então deve ser um pato”

Polimorfismo e Duck Typing - Parte 2

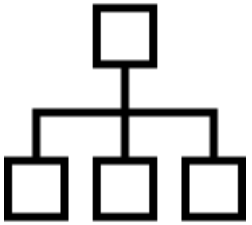


```ruby
class XmlParser
  def parse
    puts 'An instance of the XmlParser class received the parse message'
  end
end
```

```ruby
class JsonParser
  def parse
    puts 'An instance of the JsonParser class received the parse message'
  end
end
```

```ruby
class GenericParser
  def parse(parser)
    parser.parse
  end
end
```

Polimorfismo e Duck Typing - Parte 2



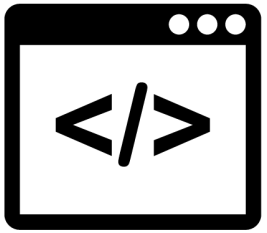
```ruby
class XmlParser
  def parse
    puts 'An instance of the XmlParser class received the parse message'
  end
end
```

```ruby
class JsonParser
  def parse
    puts 'An instance of the JsonParser class received the parse message'
  end
end
```

```ruby
class GenericParser
  def parse(parser)
    parser.parse
  end
end
```

Duck Typing =D

Exercício



- Crie as classes necessárias para o seguinte programa possa funcionar corretamente:

```ruby
def woodcutting(worker)
  unless worker.respond_to? :work
    puts "This is not a worker!"
    return nil
  end
  worker.work
  puts "#{worker.name}'s job is done!"
end

droid = Droid.new('Aida')
woodcutting(droid) # => Aida's job is done
human = Human.new('Steve')
woodcutting(human) # => Steve's job is done
dog = Dog.new('Marley')
woodcutting(dog) # => "This is not a worker!"
```

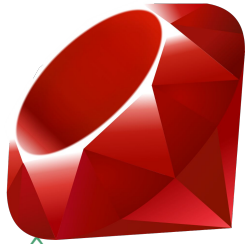
Módulos e Mix-ins

Módulos

- Algumas vezes queremos agrupar algumas estruturas que não formam uma classe naturalmente
- Módulos (**Module**) são agrupadores de métodos, classes e constantes que podem ser utilizados por várias classes
- Classes estão relacionados a objetos e Módulos estão relacionados a funções
- A Ruby tem alguns módulos nativos, como o **Math** (Teste no IRB)

```
module MyModule
  def self.a
    puts "Method 'a' from MyModule"
  end
end
```

Namespaces



- O que aconteceria se dois arquivos ou bibliotecas diferentes que você dependesse tivessem classes com o mesmo nome?

```ruby
class Document
  def generate
    # ...
  end
end
```

```ruby
require "your_xml_lib"
document = Document.new
# do something with document
puts document.generate
```

```ruby
require "their_pdf_lib"
document = Document.new
# do something with document
puts document.generate
```

Adaptado de http://practicingruby.com/articles/uses-for-modules-1

Namespaces

- Módulos são muito úteis para resolver conflitos de nome
- Mais especificamente, não temos mais conflitos de **constantes**

```ruby
module XML
  class Document
    # ...
  end
end
```

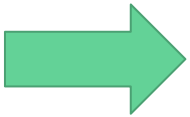
```ruby
require "your_xml_lib"
require "their_pdf_lib"

pdf_document = PDF::Document.new
xml_document = XML::Document.new
```

Adaptado de http://practicingruby.com/articles/uses-for-modules-1

Namespaces

- Namespaces também são importantes para variáveis e constantes:

```ruby
module XML
  OUTPUT = 'file.xml'
end

module PDF
  OUTPUT = 'file.pdf'
end
```



```ruby
require "your_xml_lib"
require "their_pdf_lib"

puts "The filename from PDF document is #{PDF::OUTPUT}"
puts "The filename from XML document is #{XML::OUTPUT}"
```

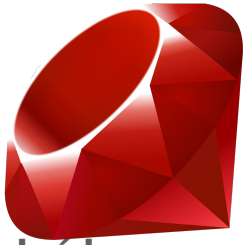
Adaptado de http://practicingruby.com/articles/uses-for-modules-1

Mix-ins

- O que acontece se eu definir métodos de objetos em um módulo?
- Como podemos usar esses métodos?

```ruby
module MyModule
  def MyModule.a
    puts "Method 'a' from MyModule"
  end

  def b
    puts "Instance method 'b' from MyModule"
  end
end
```

Mix-ins



- Embora não possamos criar instâncias de módulos, nós podemos i**ncluí-los na definição de uma classe**!
- Quando fazemos isso, todos os métodos de instância de um módulo se tornam disponíveis como métodos dos objetos da classe extendida também
- Isso são **Mix-ins**
- Módulos incluídos em classes se comportam como “**Superclasses**”
- Módulos eliminam qualquer necessidade de **Herança Múltipla** =D

Mix-ins

```ruby
module EnglishSpeaker
  def talk_in_english
    "Hello, my name is #{self.name} and I'm #{self.age} years old"
  end
end
```

```ruby
class Brazilian
  include EnglishSpeaker
  # ...
end

class French
  include EnglishSpeaker
  # ...
end
```

```ruby
brazilian = Brazilian.new("Maria", 34)
french = French.new("Henry", 45)

brazilian.talk_in_english
french.talk_in_english
```

Mix-ins

```
module Englis
  def talk_in
    "Hello, m
  end
end
```

```
years old"
```

```
class Brazilian
  include EnglishSpea
  # ...
end

class French
  include EnglishSpeaker
  # ...
end
```

```
n.new("Maria", 34)
"Henry", 45)

glish
sh
```

Esse módulo foi Mixado (Mixed in) nessas duas classes!

Mix-ins

```
module Englis
  def talk_in
    "Hello, m
  end
end
```

```
years old"
```

```
class Brazilian
  include EnglishSpea
  # ...
end

class French
  include EnglishSpeaker
  # ...
end
```

```
n.new("Maria", 34)
"Henry", 45)

glish
sh
```

Mix-ins - Interação com a Classe

- O maior poder dos Mix-ins está quando o código do Módulo interage com o código da classe, como no exemplo do módulo **EnglishSpeaker**
- O Módulo nativo **Comparable** pode ser usado para adicionar métodos de comparação a uma classe (<, <=, ==, >= e >). Para que isso funcione, o módulo Comparable assume que qualquer classe que o use define o método de comparação **<=>**
- Vamos fazer isso com a classe **Song**, baseado no tempo de duração das músicas

Mix-ins - Variáveis de Instância

- Lembram como as variáveis de instância são criadas?
- O módulo que você inclui em uma classe pode criar variáveis de instância aos objetos da classe, assim como os métodos de acesso a essas variáveis

```ruby
module BluesTune
  attr_accessor :treble, :bass
  def tuning
    @bass = 300.0
    @treble = 440.0
  end
end
```

```ruby
class Guitar
  include BluesTune
  def initialize
    tuning
    puts "Bass: #{@bass}"
    puts "Treble: #{@treble}"
  end
end

Guitar.new
```

Mix-ins - Include

- O **include** para incluir módulos em uma classe não tem nada a ver com arquivos
- Se o módulo incluído está em um arquivo diferente, esse arquivo deve ser incluído usando **require** para que ele possa ser carregado antes de ser incluído
- O include não copia os métodos para dentro da classe. As classes que incluem um mesmo módulo passam a apontar para as definições desse módulo. Caso o módulo seja alterado, todos as classes terão seus comportamentos modificados

Mix-ins - Include

- Todas as classes respondem ao método **include**
- Portanto podemos adicionar módulos em uma classe após sua definição:
  - `String.include MyModule  # => true`
- Repare que esse método é bem diferente do String#include?
  - `"something".include? "thing"  # => true`

Mix-ins - Extend

- É possível usar Mix-ins em objetos diretamente para estender suas funcionalidade
- Assim, o módulo não é incluído para todos objetos de uma classe, somente para o objeto estendido
- Se um objeto **a** é estendido com o módulo **B,** esse objeto passará a se comportar como **B** definir
- Mix-ins são fundamentais para **Duck Typing**

Mix-ins - Extend

```ruby
module ActLikeADuck
  def quack
    puts "quack"
  end
end
```

```ruby
class Duck
  include ActLikeADuck
end

class Person
end
```

```ruby
normal_person = Person.new
costumed_person = Person.new
costumed_person.extend ActLikeADuck
costumed_person.quack # => "quack"
normal_person.quack # => NoMethodError
```

Mix-ins - Extend

- **Classes também são objetos**. Portanto elas podem ser estendidas com Módulos para adicionar novos **métodos de classe**

```
class Person
  extend ActLikeADuck
end

Person.extend ActLikeADuck

Person.quack
```

Considerando o código abaixo e as diferenças entre **include** e **extend**, quais opções não retornam erro?

```ruby
module A
  def do_something
    puts "something"
  end
end

class B
  include A
end
```

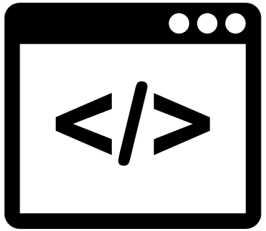
1. Chamada direta em B
   - B.do_something
2. Chamada em um instância de B
   - B.new.do_something
3. Chamada em um objeto de A
   - A.new.do_something
4. Chamada direta em A
   - A.do_something
5. Extensão e chamada em um objeto String
   - word = "something"
   - word.extend A
   - word.do_something

Considerando o código abaixo e as diferenças entre **include** e **extend**, quais opções não retornam erro?

```ruby
module A
  def do_something
    puts "something"
  end
end

class B
  include A
end
```

1. Chamada direta em B
   - `B.do_something`
2. ✓ Chamada em um instância de B
   - `B.new.do_something`
3. Chamada em um objeto de A
   - `A.new.do_something`
4. Chamada direta em A
   - `A.do_something`
5. ✓ Extensão e chamada em um objeto String
   - `word = "something"`
   - `word.extend A`
   - `word.do_something`

Exercício



Escreva um módulo chamada **Reflection** que possua o seguinte métodos de instância:

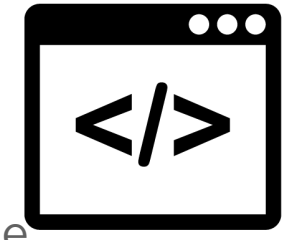
- **class_tree** - método que imprime a classe do objeto e todas as suas classes ancestrais até o **BasicObject**

Após isso, imprima a **class_tree** do número **5**, da String **“Hello World”**, do símbolo **:name**, do **Array** [**1, 2, 3**], de **Hash**

Dica:

- Verifique a superclasse de **BasicObject**

Exercício



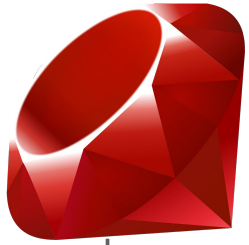
- Uma pessoa quando se torna um **programador** ganha a habilidade de **programar** e pode aprender uma ou mais linguagens de programação. Baseado nisso, crie um programa que satisfaça o seguinte código:

```ruby
person = Person.new('Joao')
person.respond_to? :program # => false
person.become_a :programmer
person.respond_to? :program # => true
person.programming_languages # => []
person.program :ruby # => "I don't know how to program in ruby"
person.learn_to_program(:ruby)
person.program :ruby # => "Programming in ruby"
person.programming_languages # => [:ruby]
```

Tratamento de Erros

Tratamento de Erros

Erros Numéricos e Exceções



- Em muitos lugares usam códigos de erros no retorno para notificar quando um erro acontece em uma operação:
  - Programas em C
  - Comandos no terminal
  - HTTP =D
- As linguagens modernas trouxeram formas mais específicas de tratamento de erros: as **Exceções**!
- Exceções são objetos da classe **Exception** que representam algum tipo de condição excepcional, indicando que algo não ocorreu como esperado!
- Quando isso ocorre, uma exceção é levantada (ou lançada)

Tratamento de Erros

- **Exceptions Handlers** são blocos de código que são executados se uma exceção ocorrer durante a execução de um bloco de código específico

```
puts "Before exception handler!"
begin
  puts 10/0
  puts "It will never be printed!"
rescue
  puts "Now we are safe!"
end
puts "After exception handler!"
```

Tratamento de Erros

- **Exceptions Handlers** são blocos de código que são executados se uma exceção ocorrer durante a execução de um bloco de código específico

Não tratado

Bloco tratado

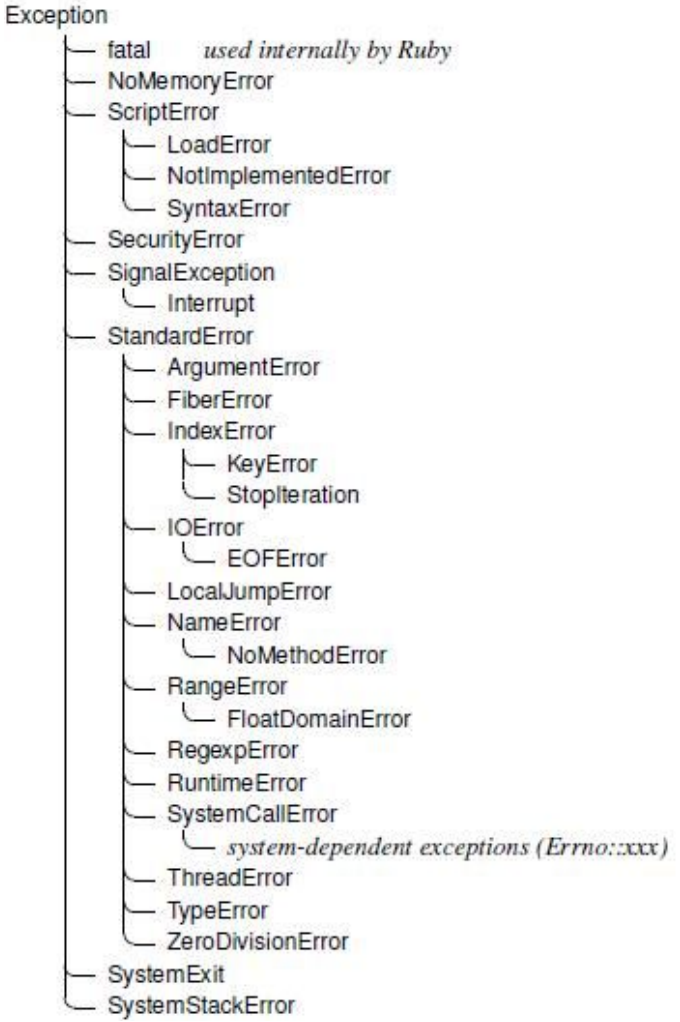
Bloco de Recuperação

Continuação

```
puts "Before exception handler!"
begin
  puts 10/0
  puts "It will never be printed!"
rescue
  puts "Now we are safe!"
end
puts "After exception handler!"
```

Hierarquia de Exceções

- O Ruby tem algumas exceções pré-definidas que podem ser utilizadas para tratar erros em seu código!
- Todas herdam de **Exception** conforme a imagem retirada do livro **Programming Ruby**
- A maior parte das exceções herdam de **StandardError**

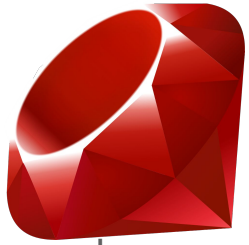


Rescue

- Dentro do bloco de tratamento de exceção, o rescue sempre recebe um parâmetro referente a qual **tipo de exceção** deve ser tratado
- Se nada for especificado, serão capturados **StandardError** por padrão
- Podemos ter vários **rescue** no mesmo bloco para tratar tipos de erros diferentes

```
begin
  puts 10/0
rescue ArgumentError
  # do this
rescue RangeError
  # do that
rescue
  puts "Entrei"
  # StandardError exceptions
end
```

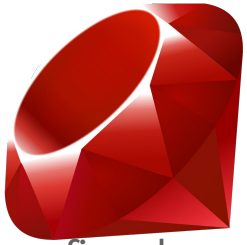
Rescue e detalhes da exceção



- Quando uma exceção é lançada, o Ruby compara essa exceção com cada um dos **rescue** para identificar qual bloco irá realizar o tratamento da exceção
- O bloco será executado se a exceção no parâmetro do **rescue** for do **mesmo tipo** da exceção lançada, ou for uma **superclasse** dessa exceção
- É possível obter mais detalhes do erro ocorrido mapeando o objeto da Exceção para uma variável no parâmetro do **rescue**

```
begin
  puts 10/0
rescue ZeroDivisionError => e
  puts e.message
  puts e.backtrace.inspect
end
```

Ensure



- Se houver alguma parte do código que deve ser executada sempre ao fim de um bloco, independente se foi lançada ou não uma exceção, colocamos esse bloco dentro de uma cláusula **ensure**

```
begin
  file = File.open('array.rb', 'r')
  file.write "bar"
rescue
  puts "Exception handling"
ensure
  puts "Closing file"
  file.close unless file.nil?
end
```

Qual será a saída do seguinte código?

```
begin
  print "1 "
  10/0
  print "2 "
rescue
  print "3 "
rescue ZeroDivisionError
  print "4 "
rescue StandardError
  print "5 "
ensure
  print "6 "
end
```

1. 1 2 6
2. 1 5 6
3. 1 4 6
4. 3 6
5. 1 3 6
6. 2 3 6

Qual será a saída do seguinte código?

```
begin
  print "1 "
  10/0
  print "2 "
rescue
  print "3 "
rescue ZeroDivisionError
  print "4 "
rescue StandardError
  print "5 "
ensure
  print "6 "
end
```

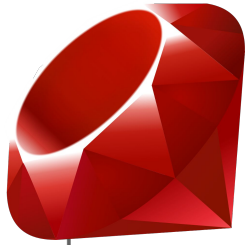
1. 1 2 6
2. 1 5 6
3. 1 4 6
4. 3 6
5. ✓ 1 3 6
6. 2 3 6

Lançando Exceções

- Nós podemos lançar exceções para tratar erros indesejados em nosso código usando a cláusula **raise**, instanciando uma nova Exceção
- Métodos implementados em classes e módulos geralmente lançam exceções, enquanto os clientes dessas classes tratam exceções

```
class Fraction
  def initialize(numerator, denominator)
    raise ArgumentError.new('Denominator cannot be zero') if denominator == 0
    @numerator = numerator
    @denominator = denominator
  end
end
```

Lançando Exceções



- Quando passamos somente um texto com nenhuma classe específica de **Exceção** na chamada do raise, o Ruby cria por padrão uma exceção do tipo **RuntimeError**

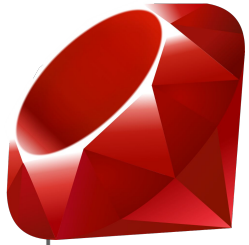
```ruby
class Fraction
  def initialize(numerator, denominator)
    raise 'Denominator cannot be zero' if denominator == 0
    @numerator = numerator
    @denominator = denominator
  end
end
```

Criando Exceções

- Muitas vezes pode ser útil criar seus próprios tipos de Exceções
- Suponha que queremos lançar um exceção do tipo **InvalidDenominatorError**
- Tente lançar a exceção abaixo diretamente no seu **IRB**

```
raise MySoftware::InvalidDenominatorError.new 'Denominator cannot be zero'
```

Criando Exceções



- Precisamos criar nossas classes de Exceção herdando de algum tipo de Exceção já existente!
- Portanto crie a seguinte classe com o namespace do seu software e tente novamente lançar a exceção abaixo

```ruby
module MySoftware
  class InvalidDenominatorError < StandardError
  end
end
```

```ruby
raise MySoftware::InvalidDenominatorError.new 'Denominator cannot be zero'
```

Revisão!

O que já vimos!

- Polimorfismo e *Duck Typing*
- Módulos e Mix-ins
- Tratamento de Erros

Atividades Sugeridas!

Resolver os seguintes desafios

- Melhore o código do o exercício do **Programador** realizado anteriormente:
  - Lance a exceção **Programmer::LanguageNotLearnedError** quando o método **program** for chamado com uma linguagem de programação não aprendida
  - Crie um módulo **ProgrammingLanguage** que contenha uma constante com a lista das linguagens de programações disponíveis para aprender e usar (Java, Ruby, Python e C++). O método **learn_to_program** deve lançar uma exceção caso seu parâmetro não seja uma dessas linguagens.
  - Programadores possuem diferentes níveis de habilidades em cada linguagem de programação. Quando ele aprende um linguagem inicialmente seu nível é 1 e quando já domina completamente a linguagem chega ao nível 10. Crie o método **train** no módulo **Programmer** que recebe uma linguagem de programação que deverá aumentar a habilidade do programador em relação a essa linguagem em 1.

Resolver os seguintes desafios

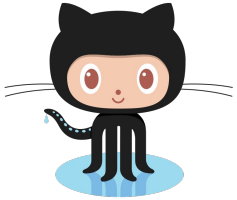
- Modele e crie um programa para gestão de livros em uma biblioteca, identificando as principais classes, módulos, relacionamentos entre elas e erros que devem ser tratados com Exceções específicas do programa:
  - Cadastrar livros com as informações de título, autores, e número de páginas. Cada livro pode ter 0 ou mais exemplares
  - Listar títulos disponíveis
  - Listar todos os livros de um autor
  - Emprestar um livro para um leitor, guardando-se o nome e email do usuário. O empréstimo deve registrar a pendência do leitor com a biblioteca e diminuir o número de exemplares.
  - Devolução de um livro, registrando a quitação do leitor com a biblioteca

Estudar

- Estudar e dominar os principais conceitos de Orientação a Objetos, principalmente relacionado a Herança, classes, composição e módulos em Ruby
- Estudar Polimorfismo em linguagens como Java e comparar com o polimorfismo em Ruby

Contato

https://gitlab.com/arthurmde



https://github.com/arthurmde

http://bit.ly/2jvND12

http://bit.ly/2j0llo9

Centro de Competência em Software Livre - CCSL

esposte@ime.usp.br

Obrigado!