

# Orientação a Objetos com Ruby

Arthur de Moura Del Esposte - [esposte@ime.usp.br](mailto:esposte@ime.usp.br)



By Arthur Del Esposte licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0)

# Aula 04 - Mix-ins, Tratamento de Erros e Bibliotecas

---

Arthur de Moura Del Esposte - [esposte@ime.usp.br](mailto:esposte@ime.usp.br)



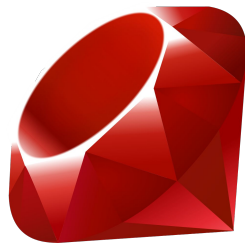
By Arthur Del Esposte licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0)

# Agenda

- Módulos - Continuação
- Considerações sobre Design de Software
- Tratamento de erros
- GEMs e bibliotecas úteis

# Relembrando Módulos e Mix-ins

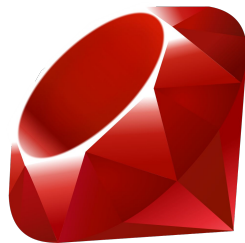
---



# 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

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

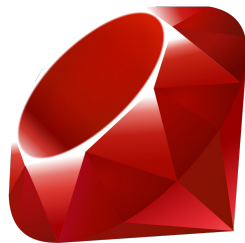
```
module XML
  class Document
    # ...
  end
end
```



```
require "your_xml_lib"
require "their_pdf_lib"

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

# Namespaces



- Namespaces também são importantes para variáveis e constantes:

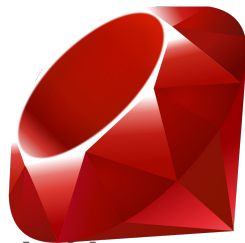
```
module XML
  OUTPUT = 'file.xml'
end
```

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



```
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}"
```

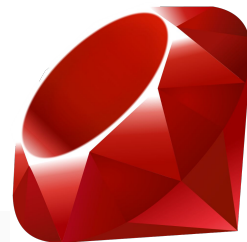


# Mix-ins

- Embora não possamos criar instâncias de módulos, nós podemos **incluí-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 estendida também
- Isso são **Mix-ins**
- Módulos incluídos em classes se comportam como “**Superclasses**”
- Módulos eliminam qualquer necessidade de **Herança Múltipla** =D



# Mix-ins



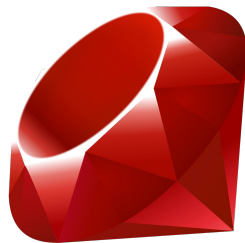
```
module EnglishSpeaker
  def talk_in_english
    "Hello, my name is #{self.name} and I'm #{self.age} years old"
  end
end
```

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

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

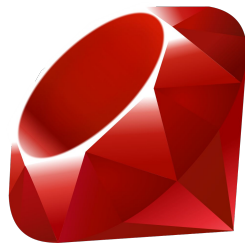
```
brazilian = Brazilian.new("Maria", 34)
french = French.new("Henry", 45)
```

```
brazilian.talk_in_english
french.talk_in_english
```



# 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



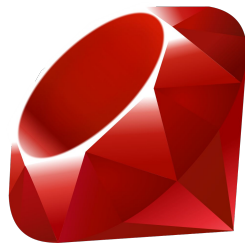
# Mix-ins - Variáveis de Instância

- 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

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

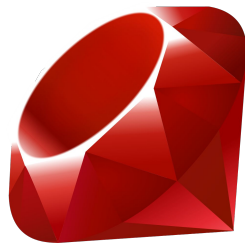
```
class Guitar
  include BluesTune
  def initialize
    tuning
    puts "Bass: #{@bass}"
    puts "Treble: #{@treble}"
  end
end

Guitar.new
```



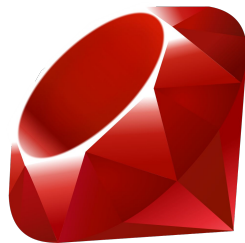
# Mix-ins - Include

- 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



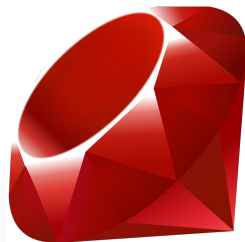
# Mix-ins - Include

- 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**



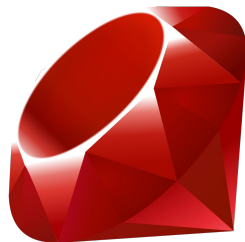
## Mix-ins - Extend

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

```
class Duck
  include ActLikeADuck
end
```

```
class Person
end
```

```
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?



```
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?

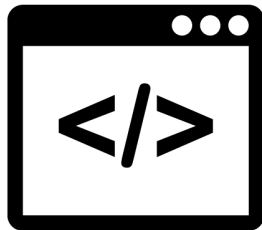


```
module A
  def do_something
    puts "something"
  end
end

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



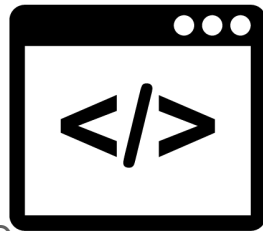
Escreva um módulo chamada **Reflection** que possua o seguinte métodos de instância:

- **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

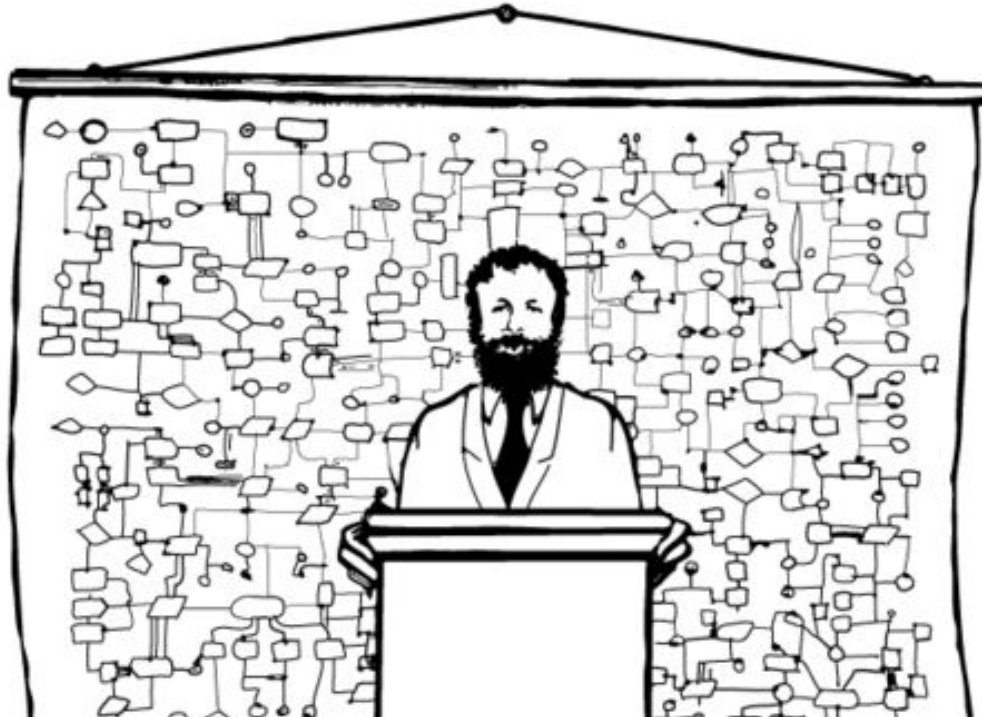
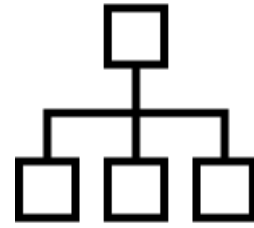
- Uma pessoa quando se torna um **programador** ganha a habilidade de **programar** e pode aprender uma ou mais linguagens de programação. Baseado nisso, crie um programa que satisfaça o seguinte código:

```
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]
```

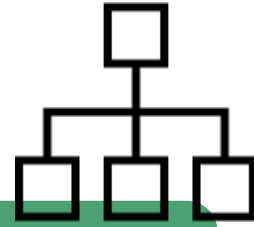
# Conceitos importantes em Design de Software

---

# O que é Design de Software?



# O que é Design de Software?



**Conjunto de decisões técnicas sobre as estruturas e organização de um sistema de software para atingir os objetivos e requisitos desse sistema**



# O que é Design de Software?



**Conjunto de decisões técnicas sobre as estruturas e organização de um sistema de software para atingir os objetivos e requisitos desse sistema**

**Definição de classes e módulos**

**Decisões em nível arquitetural**

**Distribuição de responsabilidades**

**Definição de relacionamentos entre módulos**

**Decisões relacionadas a Desempenho e Escalabilidade**

**Decisões relacionadas a Segurança**

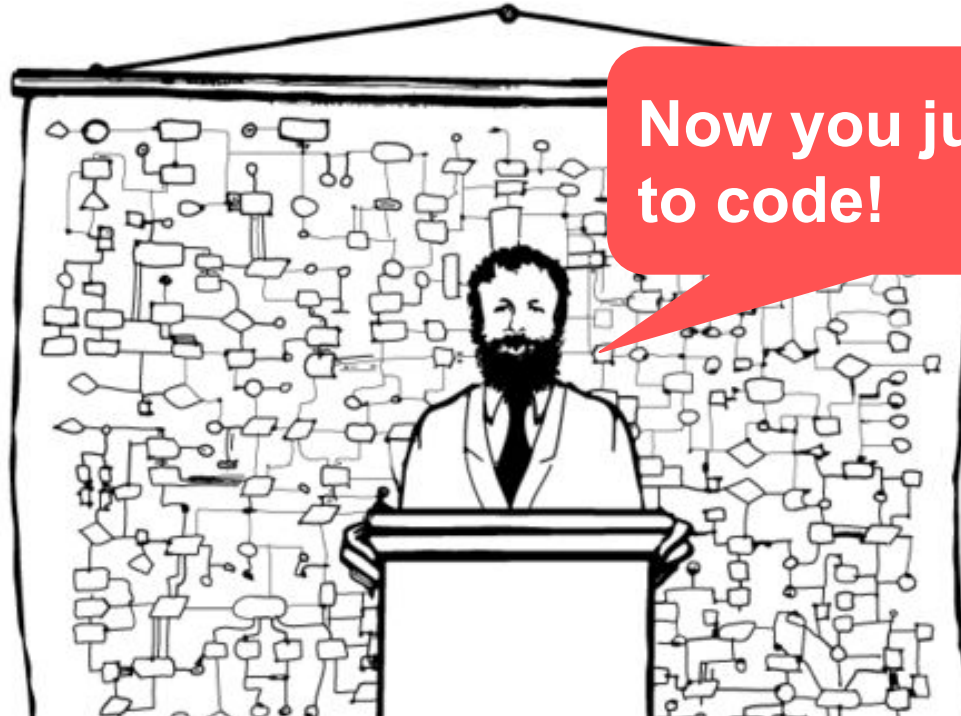
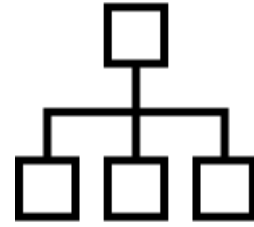
**Escolha de algoritmos**

**Tratamento e Recuperação de Erros**

**Aplicação de Padrões de Projeto**

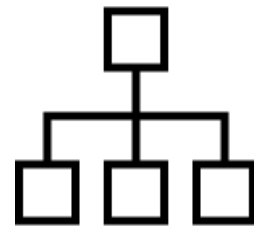


# Design de Software




**Now you just need  
to code!**

# Design de Software



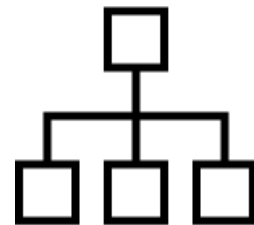
**O Código é a principal representação e objetivo do Design!**



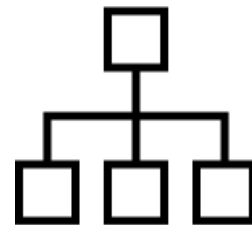
**Atividades de design e desenvolvimento acontecem iterativamente**

# Por que **Design**?

- Fatores de **qualidade interna** são fundamentais!
- Gerenciamento da complexidade do software
- Software deve crescer e evoluir
- Software será mantido por alguém
- Diminuição de custos
- **Reuso**
- **Testabilidade**
- Influenciar diretamente a **qualidade externa**!

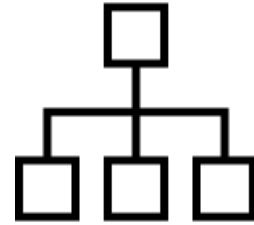


# Coesão



- **Coesão** está relacionado com as responsabilidades de um módulo e ao quanto os componentes de um módulo estão relacionados
- Idealmente, cada módulo deveria ter um única razão para existir e ser modificado
  - Todos os métodos dessa classe deveriam estar nela?
  - Os atributos que compõem esse objeto estão relacionados?
  - Os métodos manipulam esses atributos?
  - Os métodos contribuem para a abstração da classe?
  - É possível entender o que uma classe faz somente pelos nomes de seus métodos ou é preciso olhar sua implementação?
- Um módulo com muitas responsabilidades tem **Coesão Baixa**
- Um módulo com uma única responsabilidade tem **Coesão Alta**

# Coesão

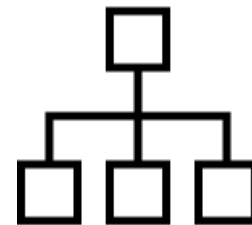
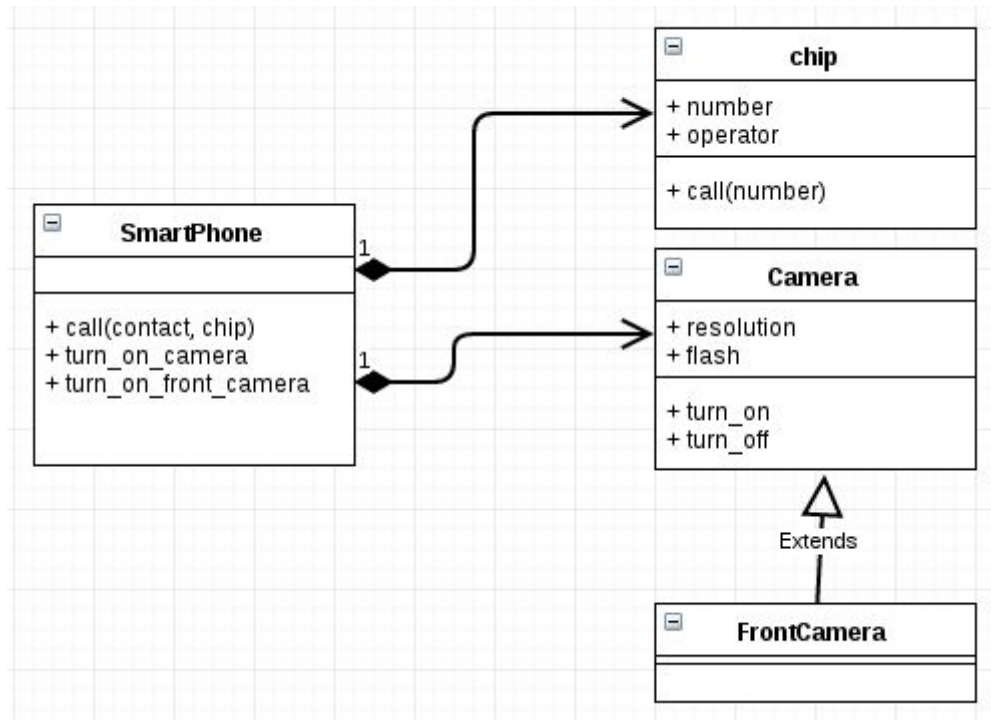


- A classe **SmartPhone** tem coesão alta ou baixa?

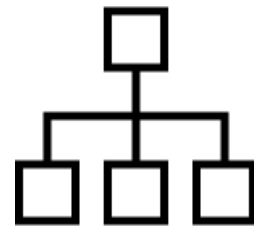


# Coesão

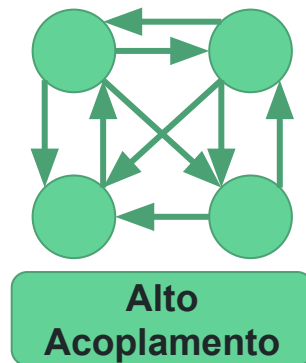
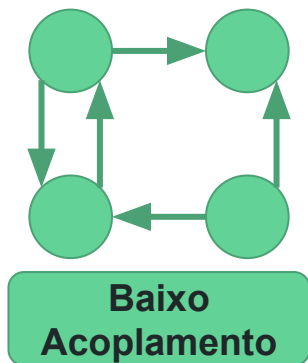
- Proposta com classes mais coesas



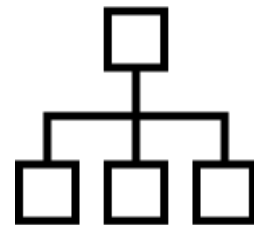
# Acoplamento



- **Acoplamento** é o grau de dependência entre módulos. Se uma classe **A** depende de uma classe **B**, **A** está acoplado a **B**.
- Também pode ser visto como uma medida de o quão conectados dois módulos estão
- Acoplamento é fundamental para o desenvolvimento de software modulares



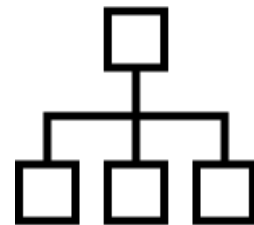
# Acoplamento



- Idealmente, cada módulo deveria ter um acoplamento baixo com dependências fracas
  - É possível reduzir um grande número de parâmetros por um objeto que os encapsule?
  - Não seria melhor fazer uma composição, em vez de usar uma herança?
  - É possível manter referências para a Superclasse, em vez de referenciar uma Subclasse específica?
  - Quando você modifica uma classe A quais outras classes também tem que ser modificadas?
  - Você consegue entender o significado de uma classe sozinha ou geralmente tem que olhar outras classes para entender o seu funcionamento?



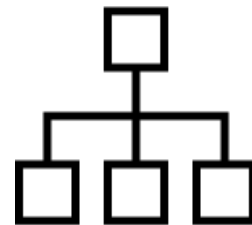
# Acoplamento



- A classe **Parser** está acoplada a quais estruturas?

```
class Parser
  def parse(content, option)
    if option == :xml
      XMLParser.new(content).parse
    elsif option == :csv
      CSVParser.new(content, {col_sep: ";"}).parse
    else
      nil
    end
  end
end
```

# Acoplamento

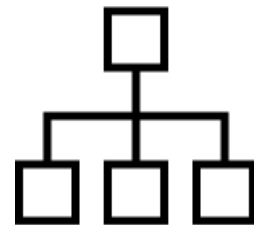


- Proposta com menos acoplamento

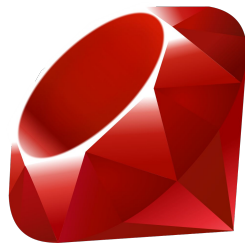
```
class Parser
  def parse(parser)
    parser.parse
  end
end
```

```
Parser.parse(CSVParser.new(content, {col_sep: ";"}))
Parser.parse(XMLParser.new(content))
```

# Coesão e Acoplamento



- Sempre buscamos **Alta Coesão** e **Baixo Acoplamento** nos módulos
- Um baixo acoplamento suporta a evolução do código sem que outros módulos tenham que ser modificados também
- Se o seu software depende muito de uma classe, ela provavelmente não é coesa. Divida suas responsabilidades em classes menores
- Se uma classe é muito acoplada, ela provavelmente tem baixa coesão, uma vez que está mais interessada nas funcionalidades e propriedades de outros módulos
- Uma classe com muitas responsabilidades é difícil de entender
- Os módulos devem ser entendidos separadamente
- Tudo está relacionado a como as **responsabilidades são distribuídas**



# Coesão e Acoplamento - em Ruby

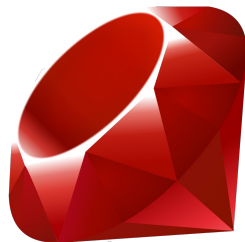
- O uso de *Duck Typing* ajuda a diminuir o acoplamento
- Não abuse de heranças! Composições podem fazer mais sentido em vários casos
- Módulos são ótimos para guardar métodos utilitários e reutilizáveis
- O uso de módulos e Mix-ins são fundamentais para se ter coesão nas Classes
- Evite usar estruturas de controle para variar o comportamento baseado no tipo, use sempre Polimorfismo!
- Use sempre os padrões sugeridos pela comunidade Ruby. Consistência, padrões e bons nomes de classes, métodos e variáveis são fundamentais para o entendimento de um código

# Tratamento de Erros

---

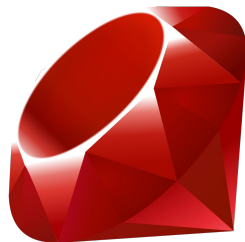
# Tratamento de Erros





# 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)

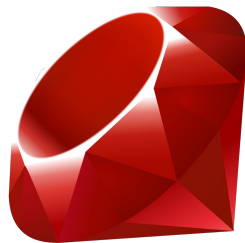


# Tratamento de Erros

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

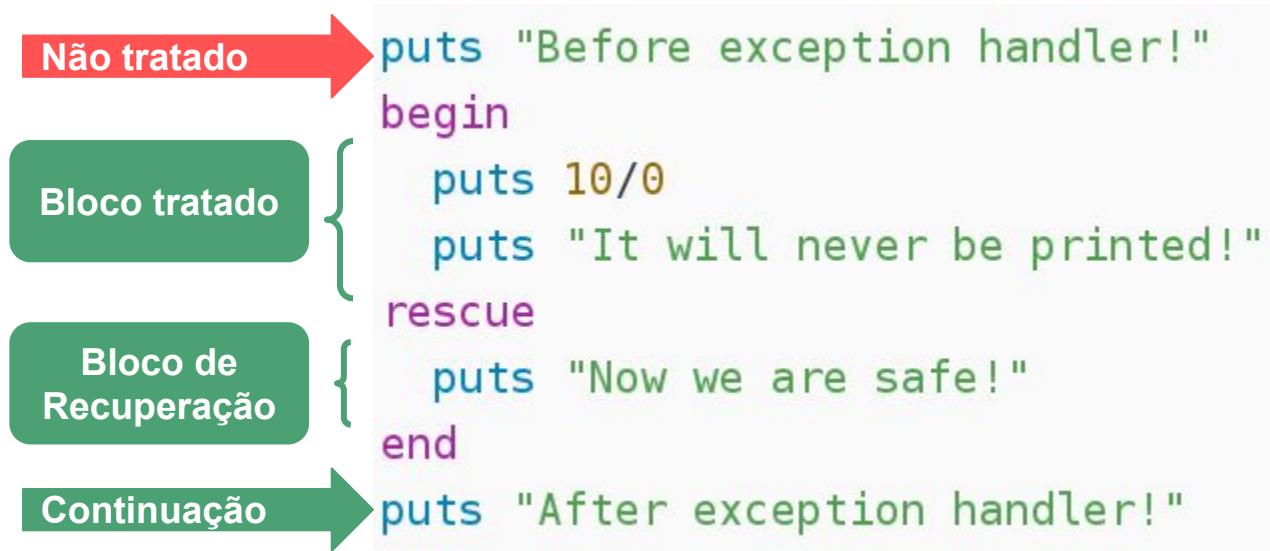
```
puts "Before exception handler!"  
begin  
  puts 10/0  
  puts "It will never be printed!"  
rescue  
  puts "Now we are safe!"  
end  
puts "After exception handler!"
```





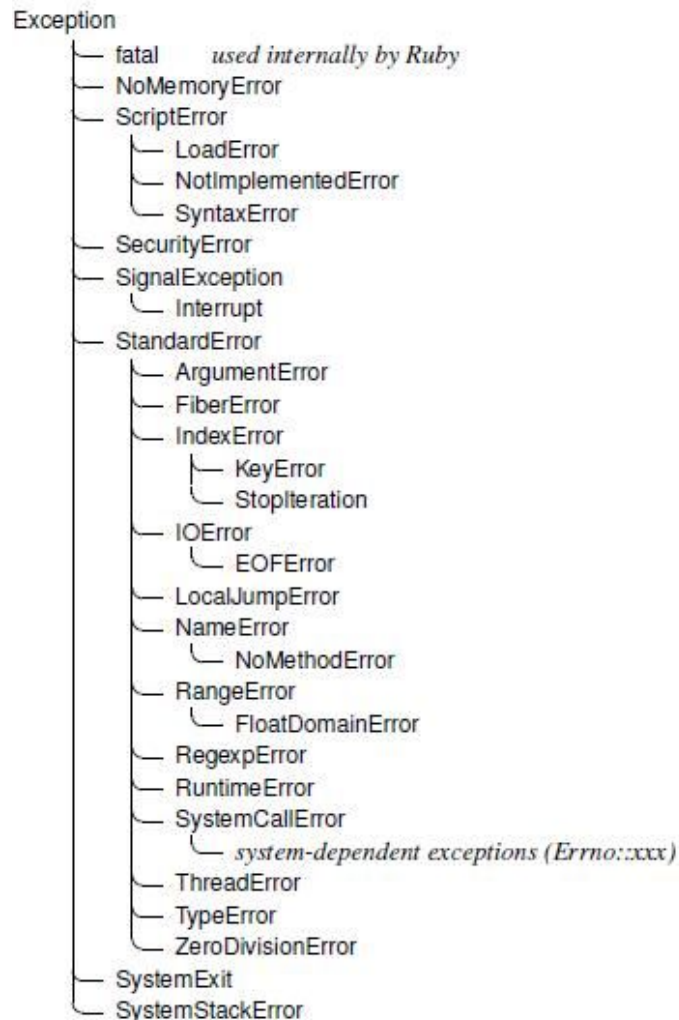
# Tratamento de Erros

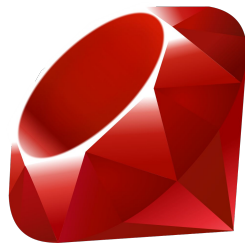
- **Exceptions Handlers** são blocos de código que são executados se uma exceção ocorrer durante a execução de um bloco de código específico



# 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**

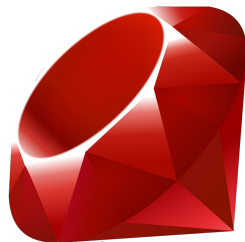




# Rescue

- 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

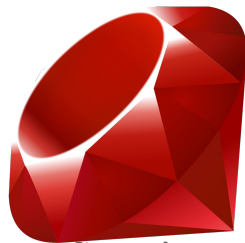
```
begin
  puts 10/0
rescue ArgumentError
  # do this
rescue RangeError
  # do that
rescue
  puts "Entrei"
  # StandardError exceptions
end
```



# 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
```

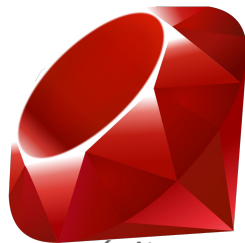
```
1.  1 2 6
2.  1 5 6
3.  1 4 6
4.  3 6
5.  1 3 6
6.  2 3 6
```

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

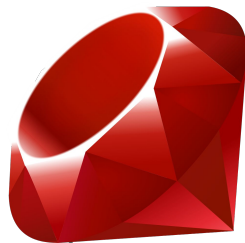


# Lançando Exceções

- 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
```

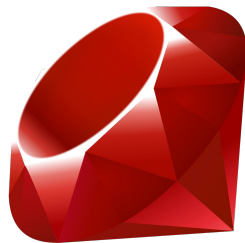




# Lançando Exceções

- 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**

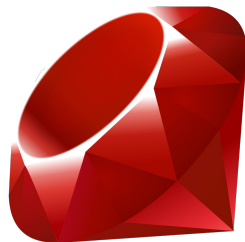
```
class Fraction
  def initialize(numerator, denominator)
    raise 'Denominator cannot be zero' if denominator == 0
    @numerator = numerator
    @denominator = denominator
  end
end
```



# 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'
```



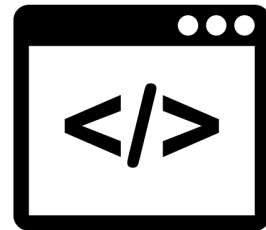
# Criando Exceções

- 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

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

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

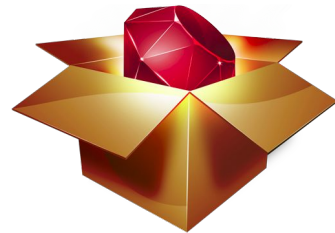
## Exercício



- Altere a classe **Fraction** que você criou anteriormente e lance uma exceção do tipo **ArgumentError** quando o parâmetro *denominator* for **zero**

# Ruby Gems

---

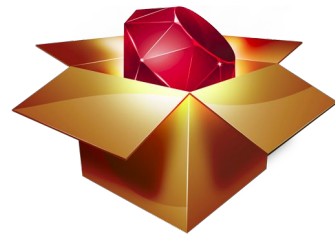


# Gems

- **Gems são pacotes de software ruby**
- Uma Gem é uma biblioteca ou um conjunto de arquivos reutilizáveis, etiquetadas em um nome e uma versão
- **RubyGems** é um sistema de gerenciamento de pacotes Ruby que facilita a criação, compartilhamento e instalação de bibliotecas
- A instalação do Ruby já vem com o gerenciador de pacotes Ruby que pode ser acessado via linha de comando:

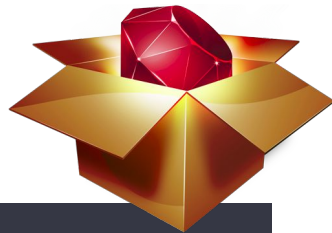
```
$ gem -v
```

```
$ gem -h
```



# Uso de Gems

- **Existem MUITAS bibliotecas disponíveis em Ruby para os mais variados propósitos**
- Os passos básicos para usar uma **Gem** são:
  - a. Encontrar bibliotecas
  - b. Instalar bibliotecas localmente
  - c. Importar as bibliotecas para o código-fonte
  - d. Interagir com a biblioteca através de sua API
- As bibliotecas geralmente possuem código-fonte no [Github](#)



# Encontrando bibliotecas

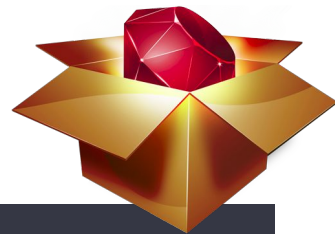
- Buscar por bibliotecas
- Busca com mais informações
- Buscar por bibliotecas instaladas

```
$ gem search rails
```

```
$ gem search remote-user -d
```

```
$ gem search -l rails
```





# Instalando bibliotecas

- Instalação comum de uma biblioteca
- Instalação sem documentação
- Instalar uma versão específica
- Listar todas as gems instaladas
- Remoção de uma gem instalada

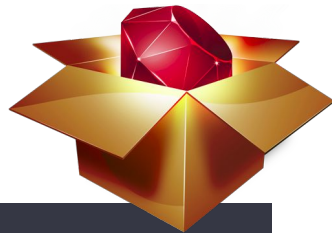
```
$ gem install colorize
```

```
$ gem install colorize --no-doc
```

```
$ gem install rails -v 4.0
```

```
$ gem list
```

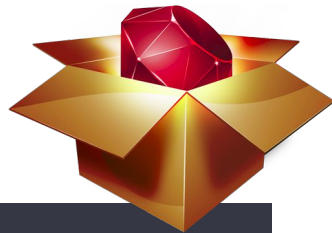
```
$ gem uninstall colorize
```



# Usando bibliotecas

- Ler a documentação localmente
- Para carregamos a infraestrutura de RubyGems temos que usar:
  - `require 'rubygems'`
- Assim, podemos incluir Gems instaladas
  - `require 'colorize'`

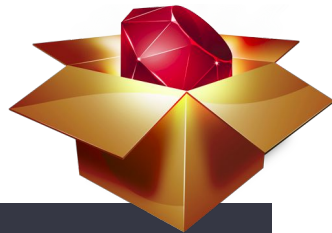
```
$ gem server
```



# GEM - Bundler

```
$ gem install bundler
```

- Bundler proporciona um ambiente para gerenciamento de dependências de RubyGems para projetos em Ruby
- Mapeia e instala as dependências necessárias de um projeto
- Para usá-lo, temos que criar um arquivo na raiz do projeto chamado **Gemfile**, onde especificamos quais Gems e versões são necessárias para esse projeto



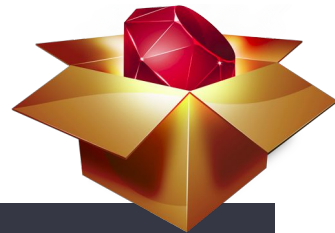
# GEM - Bundler

- Instalando dependências
- Atualizando dependências

```
$ bundle install
```

```
$ bundle update
```

- Após a instalação, o **Bundler** vai gerar um arquivo chamado **Gemfile.lock** que contém exatamente quais versões foram instaladas de cada dependência
- Veja o exemplo em um projeto real: <https://github.com/Kuniri/kuniri>

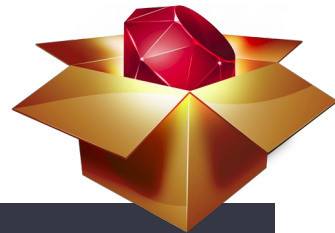


# GEM - Colorize

```
$ gem install colorize
```

- A Gem Colorize adiciona vários métodos que permitem a formatação de **cores** e **modos** em Strings
- Veja o exemplo **colorize.rb**

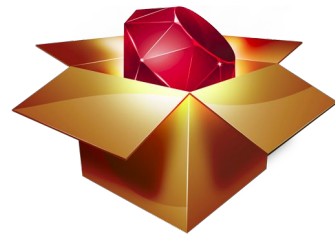
```
=====
SOME EXAMPLES:
Blue text
Red underline text
Green bold text
```



# GEM - PowerPack

```
$ gem install powerpack
```

- A Gem PowerPack oferece extensões úteis para as classes nativas de Ruby
- Em sua página tem a lista de todos os métodos adicionados em cada classe
- Use o IRB para testar alguns dos métodos adicionados
- Alguma ideia de como esse biblioteca adiciona esses métodos?
- **Desafio:** estenda a classe **Array** e adicione o método **element\_types** que retorne uma Hash com a relação dos tipos de elementos existentes e a quantidade de cada tipo
  - **Entrada:** `[1, 2, "oi", :boy].element_types`
  - **Saída:** `{Fixnum=>2, String=>1, Symbol=>1}`



# GEM - Outras Gems

- Existem MUITAS Gems para serem exploradas, tais como:
  - [Graticule](#)
  - [GLI](#)
  - [Mechanize](#)
  - [Sinatra](#)
  - [Rails](#)
- Você também pode criar sua própria Gem utilizando o [Bundler](#)
- Usar bibliotecas é uma das melhores formas de **reutilização de código**. Muitas vezes algo que você deseja, já está pronto e empacotado como um Gem que você pode reaproveitar.

Revisão!





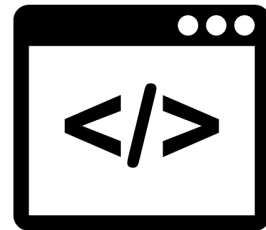
# O que já vimos!

- Módulos e Mix-ins
- Considerações sobre Design de Software
- Tratamento de erros
- Ruby Gems

# Atividades Sugeridas!

---

# Exercício

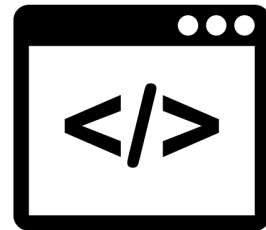


Modele e escreva um programa que crie personagens para um jogo de RPG. Todos os personagens tem nome, idade, vida, ataque, defesa, e uma **Raça** (Humano, Elfo, Orc, Anão, Hobbit). Além disso, os personagens começam com uma **classe** e podem treinar para ter uma outra classe adicional e ganhar mais habilidades.

Raça	Atributos
Humano	Vida: 20, Ata: 8, Def: 8
Elfo	Vida: 25, Ata: 5, Def: 6
Anão	Vida: 18, Ata: 9, Def: 11
Orc	Vida: 15, Ata: 12, Def: 5

Classe	Habilidades
Construtor	Construir casas
Ferreiro	Construir espadas e armaduras
Curandeiro	Curar outras unidades
Guerreiro	Atacar, defender

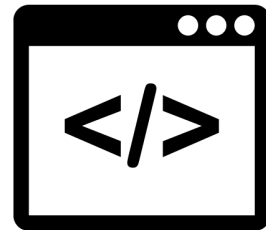
## Exercício



Veja a lista de RubyGems mais populares e escolha uma para instalar e criar um programa que a utilize:

<https://www.ruby-toolbox.com/>

## Exercício



**Web Scraping** é uma abordagem para coletar dados de páginas **Web** a partir da navegação e limpeza de dados automatizadas através de software (scripts, robôs, aplicações). Aprenda e utilize a Gem Mechanize para obter informações de páginas Web através [desse tutorial](#) e posteriormente faça:

- Escolha uma página na Wikipedia e escreva um script que conte quantos links essa página possui
- Escreva um script que receba uma URL de alguma página da Wikipedia e imprima o seu título e resumo.

# Estudar

- Estudar princípios de Design de Software
- Estudar o que são Padrões de Projetos de Software
- Estudar os seguintes padrões de projeto:
  - Template Method
  - Strategy
  - Observer
  - Composite

# 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](mailto:esposte@ime.usp.br)

Obrigado!