# Trabalho sobre a linguagem de programação Ruby

- Aluno: Matheus Costa
- Matrícula: 201720377011
- Matéria: Estrutura de Linguagens

# A origem de Ruby

- Foi originalmente planejada e desenvolvida no Japão em 1995, por Yukihiro Matsumoto.
- Queria desenvolver uma linguagem de script.
- Se tornou uma linguagem de programação interpretada multiparadigma.
- De tipagem dinâmica e forte, com gerenciamento de memória automático.

# Influências em Ruby

- Ruby foi inspirada principalmente por Python, Perl, Smalltak e Lisp.
- Matsumoto queria que Ruby fosse:
  - Mais poderosa do que Perl;
  - Mais orientada a objetos do que Python;
  - E que seguisse a influência da linguagem Smalltalk em atribuir métodos e variáveis de instância a todos os seus tipos.

# Classificação de Ruby

- Ruby tem uma ampla área de uso.
- É uma linguagem orientada a objetos.
- Uso de Ruby com Rails para Web.
- Aplicações desktop.
- Desenvolvimento de aplicações back-end.

# Funcionalidade Mixins

- A linguagem Ruby não trabalha com herança múltipla diretamente.
- Não é possível herdar de várias classes ao mesmo tempo.
- Na prática, Mixins utiliza módulos que podem ser consumidos em uma classe permitindo a herança múltipla.

# Exemplo do uso de Mixins:

```
module Primeiro
  def metodoA
  end
end
module Segundo
  def metodoB
  end
end
# Classe que permite incluir vários módulos
class MesclarModulos
  include Primeiro
  include Segundo
    def metodoC
    end
end
# Consumo da classe MesclarModulos
  resultado = MesclarModulos.new
  resultado.metodoA
  resultado.metodoB
  resultado.metodoC
```

# Exemplo de código:

```
module Funcionario
   def id
      @id
   end

   def id=(str)
      @id = str
   end

   def cargo
      @cargo
   end

   def cargo=(str2)
      @cargo = str2

   end

   def investmento_minimo
      @investmento_minimo
   end

   def investmento_minimo=(str5)
      @investmento_minimo = str5
   end

end
```

```
module Investidor
   def empresa
      @empresa
   end

   def empresa=(str3)
      @empresa = str3
   end

   def quant_invest
      @quant_invest
   end

   def quant_invest=(str4)
      @quant_invest= str4

   end

end
```

# Cont. exemplo de código:

```
class Pessoa
  include Funcionario
  include Investidor
  def funcao
    id
    cargo
    investmento_minimo
  end
  def invst
    empresa
    quant_invest
  end
```

```
def comparavalor
      investmento_minimo
      quant_invest
   end
end
p = Pessoa.new
puts
("------------ INFORMAÇÕES DO FUNCIONÁRIO ------------")
puts
print("Digite seu ID: ")
p.id = gets.to_i
print("Digite seu cargo: ")
p.cargo = gets.to_i
print("Digite o investimento minimo aceito: ")
p.investmento_minimo = gets.to_f
```

# Comparação em C#:

```
public class Funcionario
{
    string id;
    public string FuncionarioId
    {
        get
        {
            return id;
        }
        set
        {
            id = value;
        }
    }
    {
        id = FuncionarioId;
        cargo = FuncionarioCargo;
    string cargo;
        public string FuncionarioCargo
        {
            get
            {
                return cargo;
            }
            set
            {
                cargo = value;
            }
        }
                    public class Shop : Funcionario
     * herdado apenas de Funcionario */
    {
    *mas tem objeto "InvestidorAux" dentro */
        InvestidorAux InvestidorPart;
    /* Shops tem endereço
    alem do Funcionario id.
    E Nome é herdado de
    Funcionario e Investidor*/
        string address;
        public string Address
        {
            get
            {
                return address;
            }
            set
            {
                address = value;
            }
        }
```

# Análise crítica:

- Em Ruby:

1. Quebram encapsulamento.

2. Facilidade com sintaxe

3. Resolve problema de herança múltipla.

4. Módulos podem ser consumidos em uma classe.

5. Uma classe pode herdar recursos de uma classe pai.

6. Maneira controlada de adicionar funcionalidade às classes.

Em C#:

1. Sistemas fracamente acoplados.

2. Mais flexíveis à mudanças no código.

3. Não precisa de conhecimento para implementação.

4. Precisa de conhecimento para invocar as funcionalidades.

5. Pode trocar a implementação de um componente em tempo de execução.