IFSP

RUBY..!

MANUAL DO ALUNO PARA O MUNDO DA PROGRAMAÇÃO

puts("

ALISSON VICENTE DE LIMA
ALLEF FERREIRA MOREIRA
CAMILA DE ALMEIDA ROCHA NEVES
DAVI DO NASCIMENTO CAVALGANTE
GABRIEL SIMÕES DE CARVALHO
LUANA MONTEIRO ROSA
MARCELO DE CARVALHO GOMES
YURI ABREU TEIXEIRA PINTO")

CUBATÃO, 2021

ALISSON VICENTE DE LIMA
ALLEF FERREIRA MOREIRA
CAMILA DE ALMEIDA ROCHA NEVES
DAVI DO NASCIMENTO CAVALCANTE
GABRIEL SIMÕES DE CARVALHO
LUANA MONTEIRO ROSA
MARCELO DE CARVALHO GOMES
YURI ABREU TEIXEIRA PINTO
CTII 418

TRABALHO DE CONCLUSÃO DE CURSO: LINGUAGEM RUBY

Pesquisa apresentada à disciplina de Projeto de Sistemas, junto ao Curso Técnico em Informática Integrado ao Ensino Médio do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, Câmpus Cubatão.

BANCA EXAMINADORA

_____

Prof. Maurício Asenjo

CUBATÃO

2021

“Programador: “É simples, pode ser composto. Se firma no concreto e torna-se, abstrato.”

(Carlos Cesar)

ÍNDICE

1. CAPÍTULO 1: INTRODUZINDO A HISTÓRIA DA LINGUAGEM RUBY

A linguagem Ruby foi criada na década de 90 por Yukihiro “Matz” Matsumoto. Foi inspirada em outras linguagens para o seu desenvolvimento, entre elas Perl, Smalltalk, Eiffel, Ada e Lisp.

Desde o momento em que foi tornado público, isso em 1995, o Ruby conseguiu consigo programadores devotos em todo o mundo. Em 2006, o Ruby atingiu aceitação massiva, com a formação de vários de usuários em todas as principais cidades do mundo e com as conferências sobre Ruby com lotação esgotada.

A linguagem Ruby está posicionada entre no top 10 da maioria dos índices que medem o crescimento da popularidade de linguagens de programação pelo mundo todo (tais como o índice TIOBE). Muito deste crescimento é atribuído à popularidade de softwares escritos em Ruby, em particular o framework de desenvolvimento web Ruby on Rails.

O Ruby foi criado para que as pessoas que desenvolvem softwares entendam com facilidade como ele funciona. Não é à toa que ele ocupa a primeira posição entre as linguagens que mais crescem.

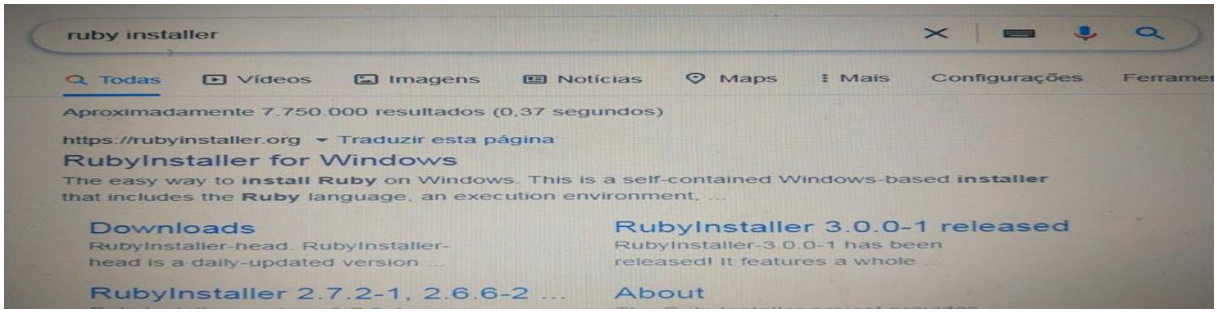
Ruby pode ser utilizada para criar aplicações desktop, e várias outras utilidades.

Com o advento do desenvolvimento web na década de 2000, foi criado, em 2003, o framework web Ruby on Rails, o que popularizou muito a linguagem para desenvolvimento de aplicações web.

2. MODO DE INSTALAÇÃO E CONFIGURAÇÕES

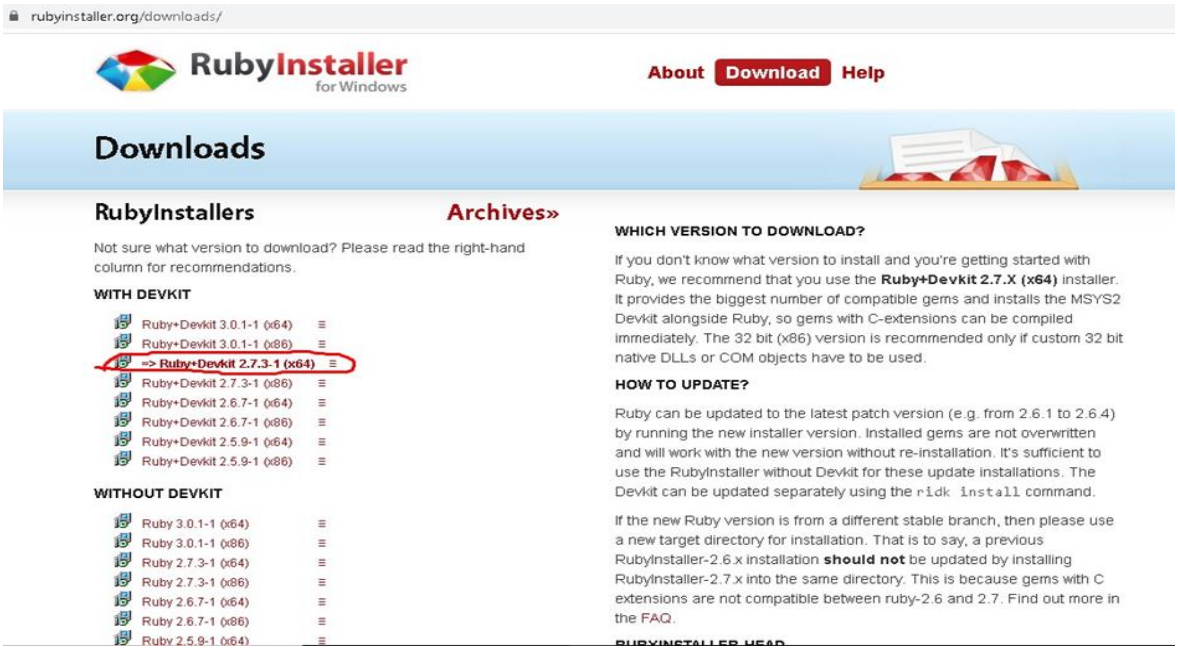
2.1 INSTALAÇÃO DE JDK

- **Passo 1:** Para início de conversa precisamos baixar a linguagem Ruby na sua máquina, pesquise no seu navegador ‘ruby installer’:

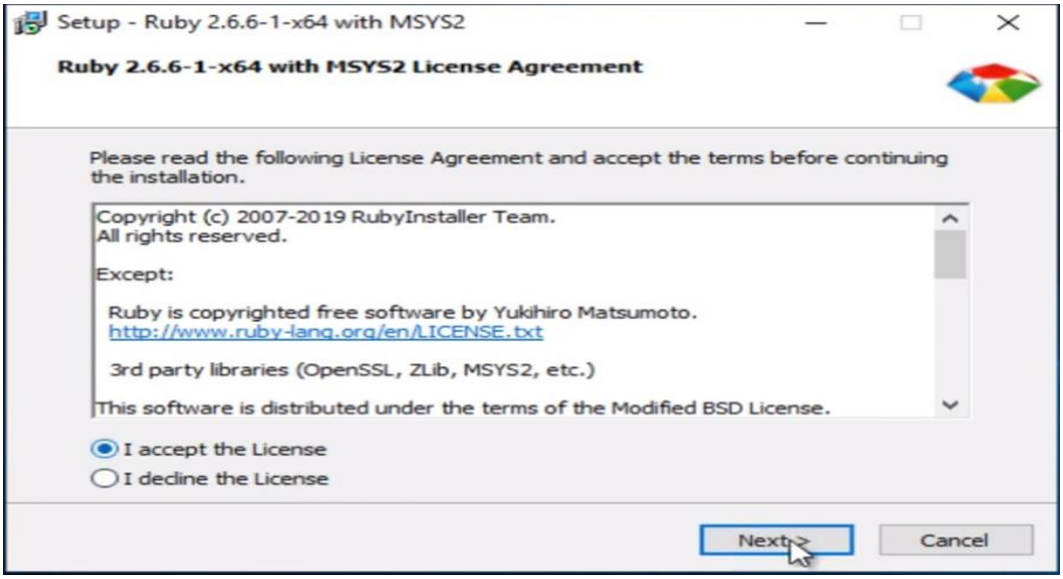


- **Passo 2:** Dentro do site, clique no botão vermelho que aparece no início da página:

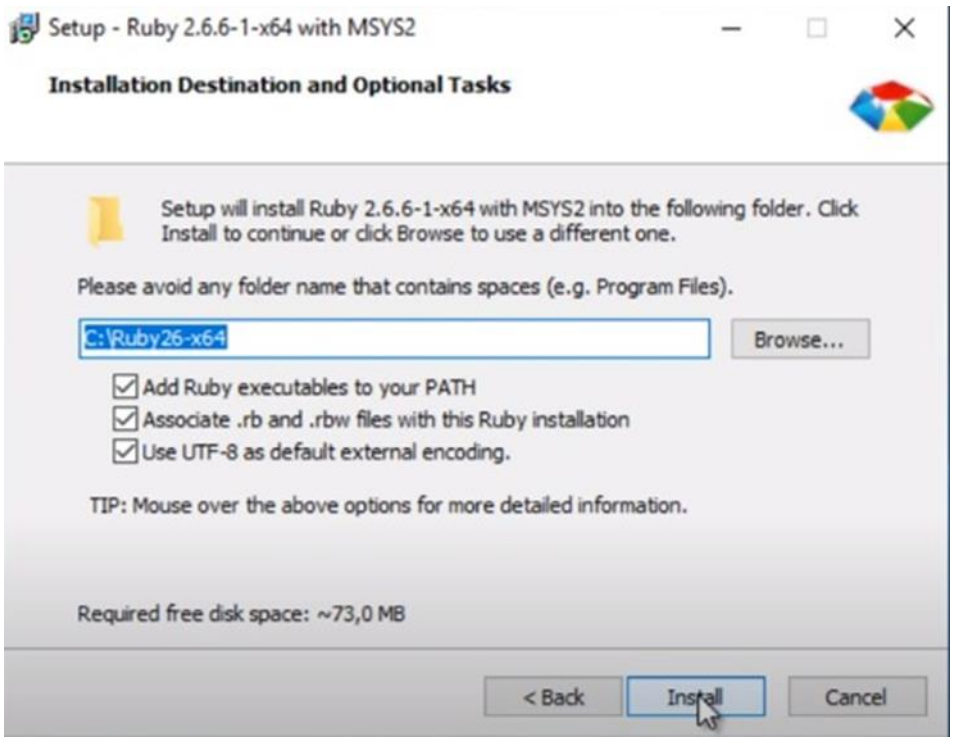
- **Passo 3:** Selecione a versão que mais agradar a sua persona, por padrão o site já recomenda a versão 2.7.3-1 e utilizaremos ela no processo:



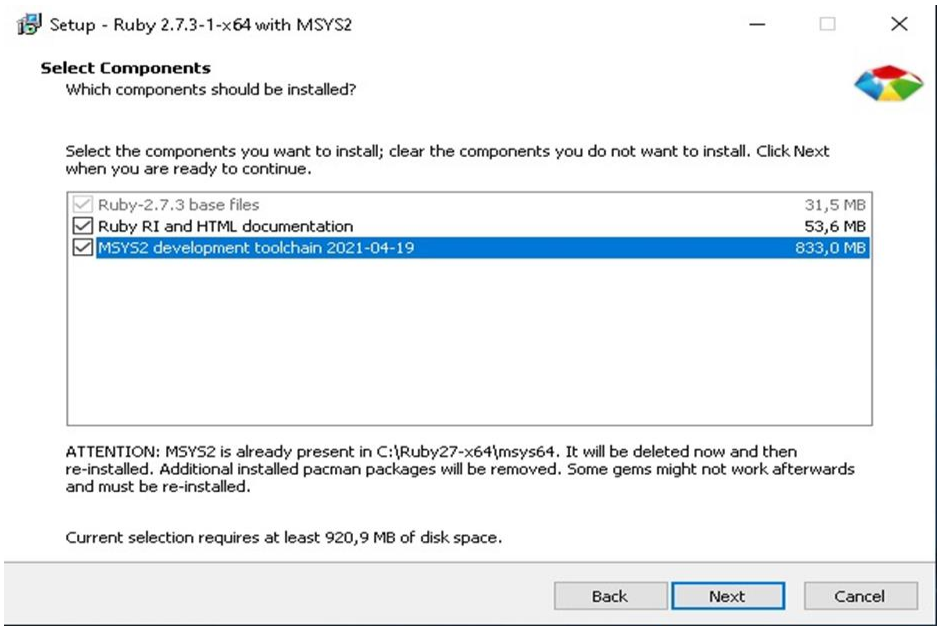
- **Passo 4:** Com o instalador baixado inicie ele, você verá uma tela igual a disponibilizada logo abaixo, você precisa marcar a opção para aceitar a licença e em seguida clicar em *next:*



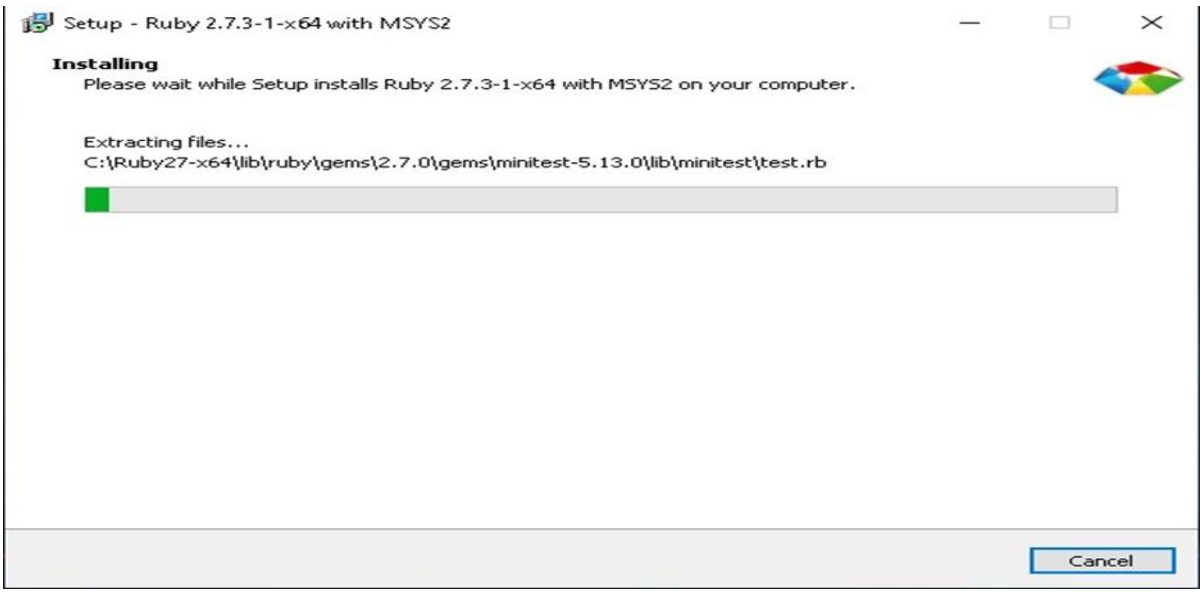
- **Passo 5:** Escolha o diretório onde os arquivos ficarão armazenados, e principalmente, marque todas as '*caixinhas*' pois utilizaremos todas as ferramentas até o final do curso. Com tudo isso feito só falta iniciar a instalação



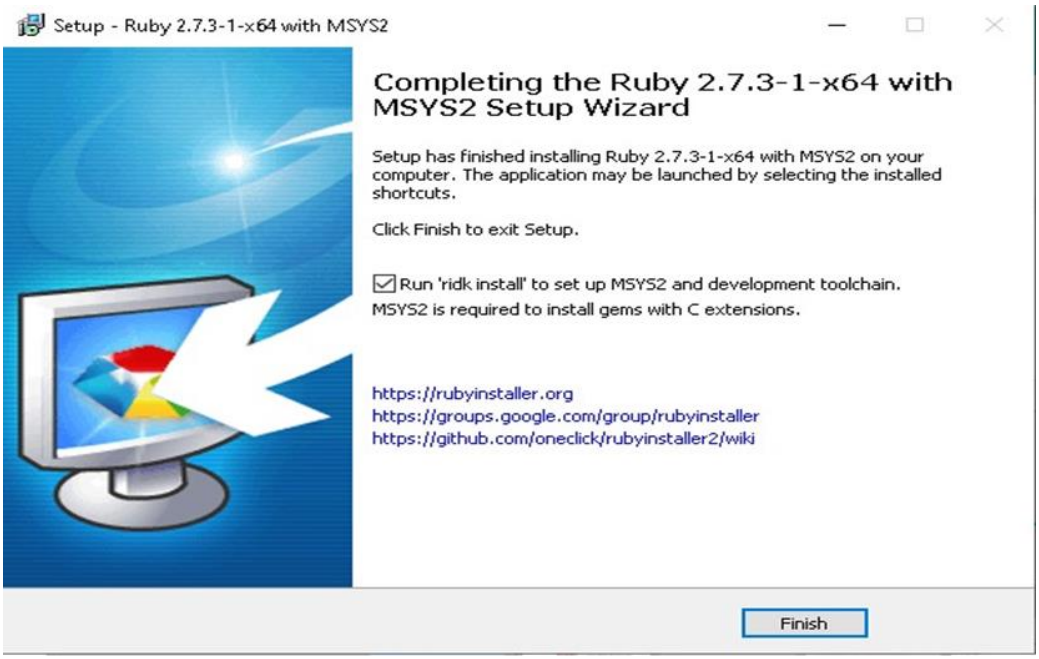
- **Passo 6**: Marque mais essas “caixinhas” que são as nossas ferramentas, clique em *next* depois disso:



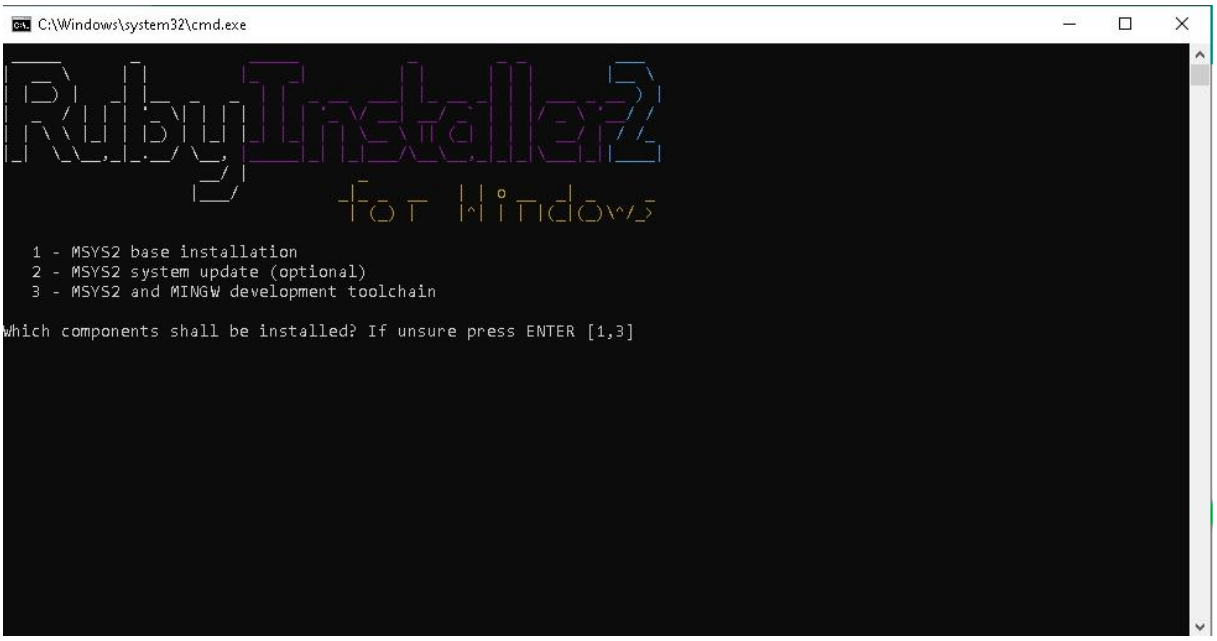
- **Passo 7:** Depois disso o download já está iniciado:



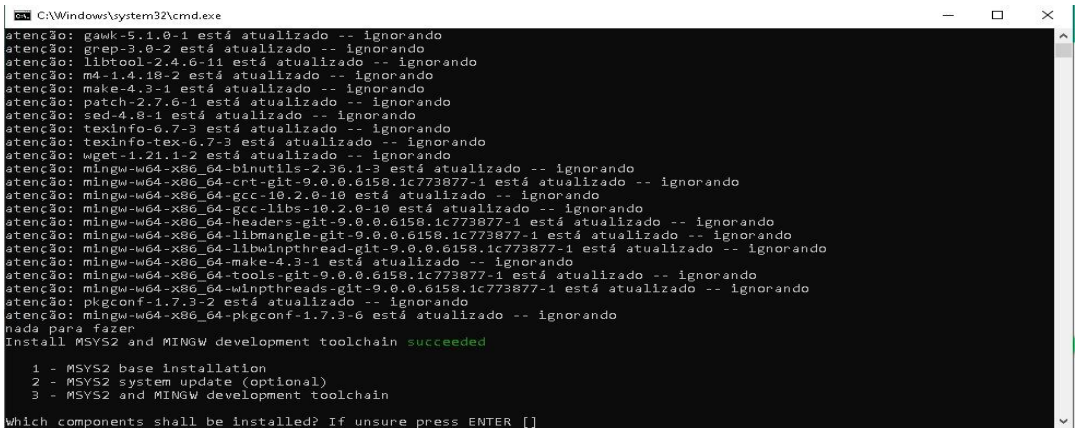
- **Passo 8:** Na sequência essa tela se abrirá, aqui você pode observar que a sua instalação já está concluída:



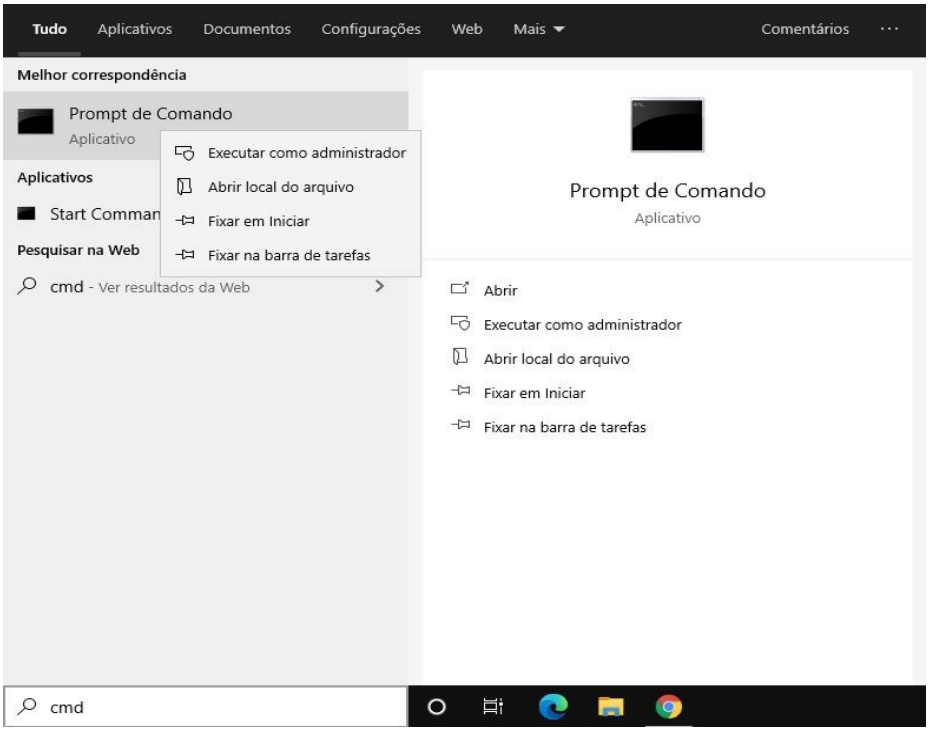
- **Passo 9:** Após apertar o *Finish* o Prompt de Comando será aberto automaticamente, nele é só apertar a tecla *Enter* para confirmar a instalação das ferramentas:



- **Passo 10:** Quando a instalação tiver concluída é só apertar a tecla *Enter* de novo e o *Prompt* se fechará na sequência:

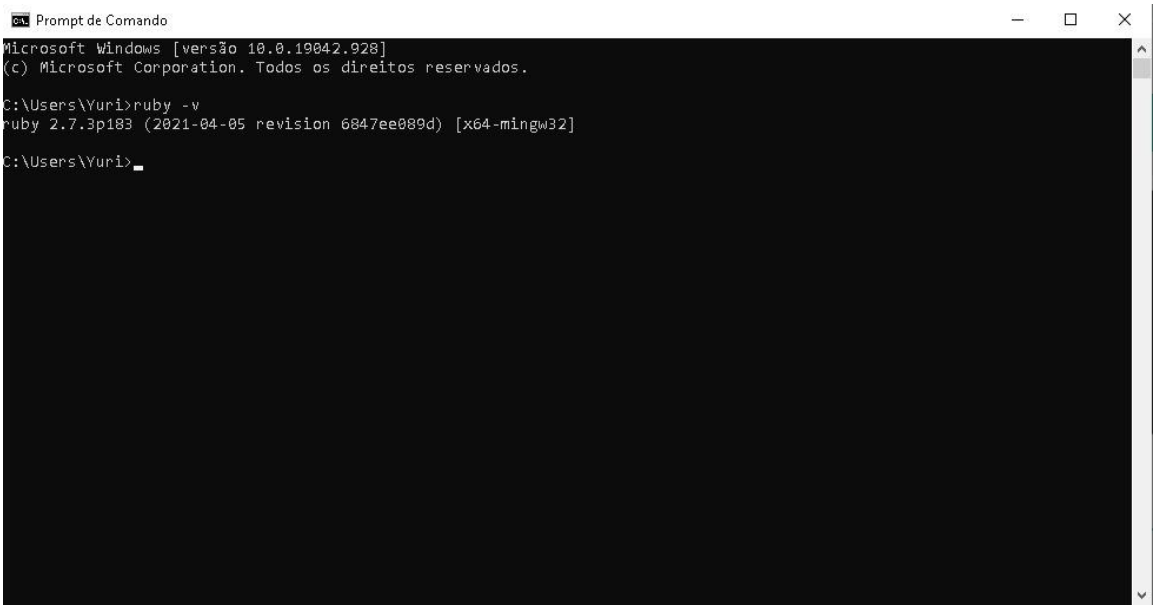


- **Passo 11:** Para confirmar que o Ruby está instalado na sua máquina você pode abrir o Prompt de Comando:



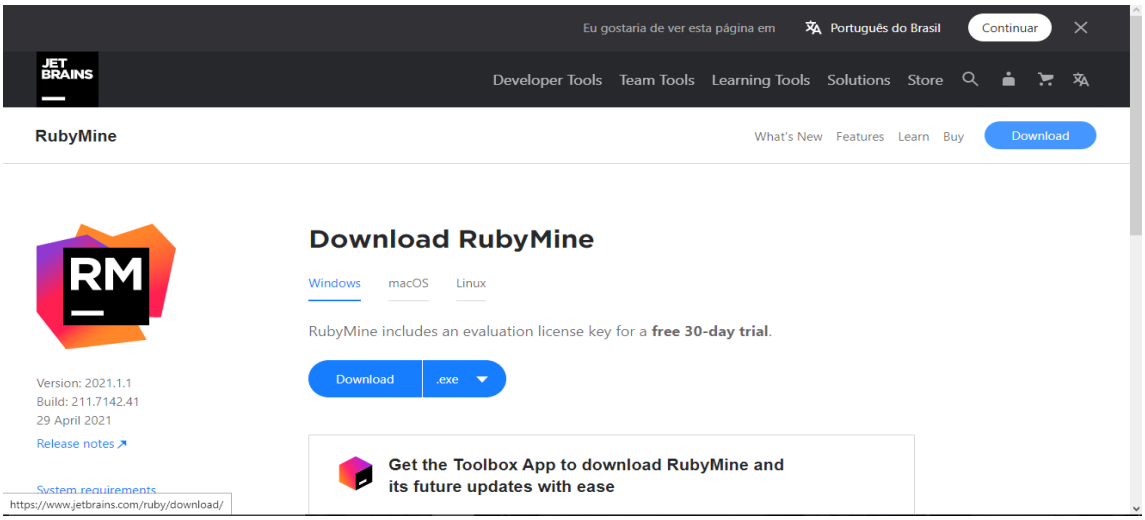
- **Passo 12:** Inserira o trecho “ruby –v”, sem aspas:

- **Passo 13:** Se a instalação estiver concluída com êxito, ao inserir o trecho anterior aparecerá a versão que está instalada na sua máquina:



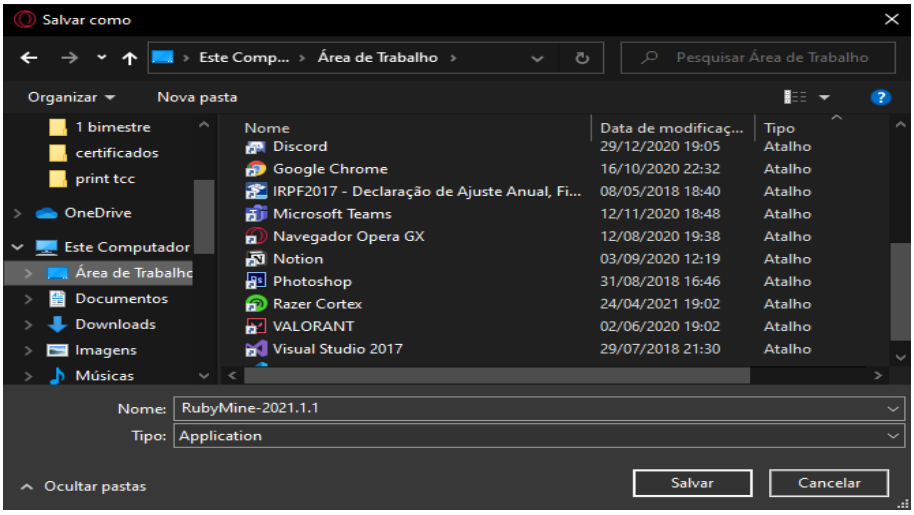
2.2 INSTALAÇÃO DE IDE

- **Passo 1**: Para começar a instalação você deve acessar o link https://www.jetbrains.com/ruby/download/, nele você encontrará a seguinte tela:

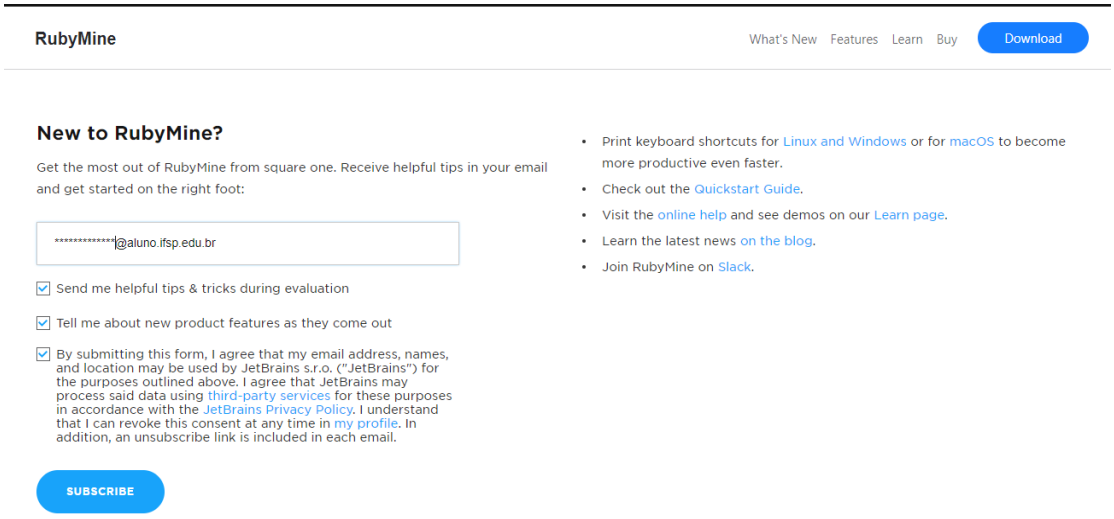


- **Passo 2:** Para iniciar a instalação do Setup clique em download, mais especificamente o que está ao lado do botão *.exe* (que pode ser visto na imagem acima).

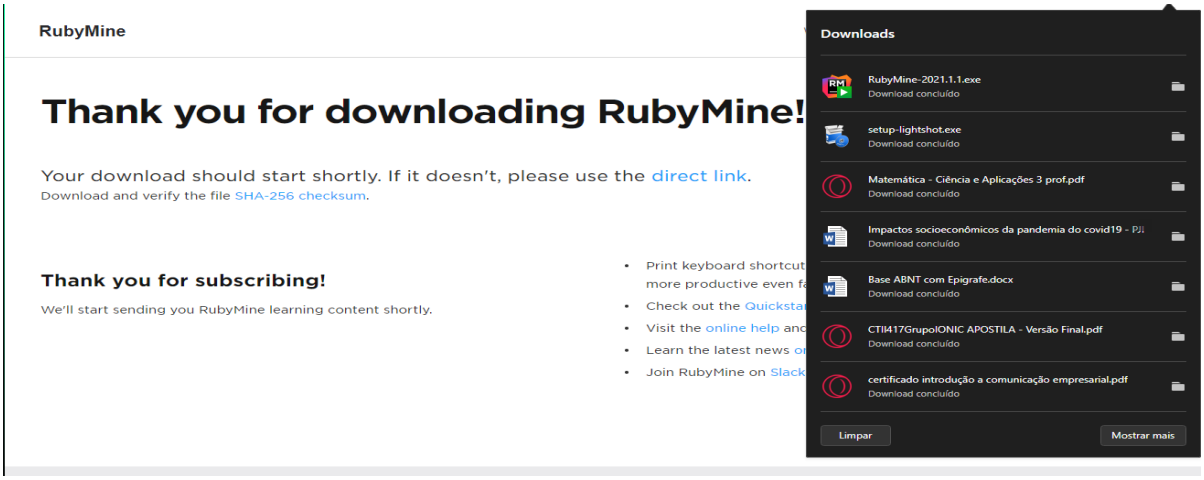
- **Passo 3**: Depois que a instalação do arquivo for concluida voce só tera que escolher onde deseja colocar o seu arquivo, o qual será o Setup para a instalação da IDE:



- **Passo 4:** Depois que o download estiver pronto você precisará criar um cadastro, de preferência deve ser utilizado um e-mail institucional. Para poder seguir adiante você precisa marcar as três “caixinhas” que aparecem no site:

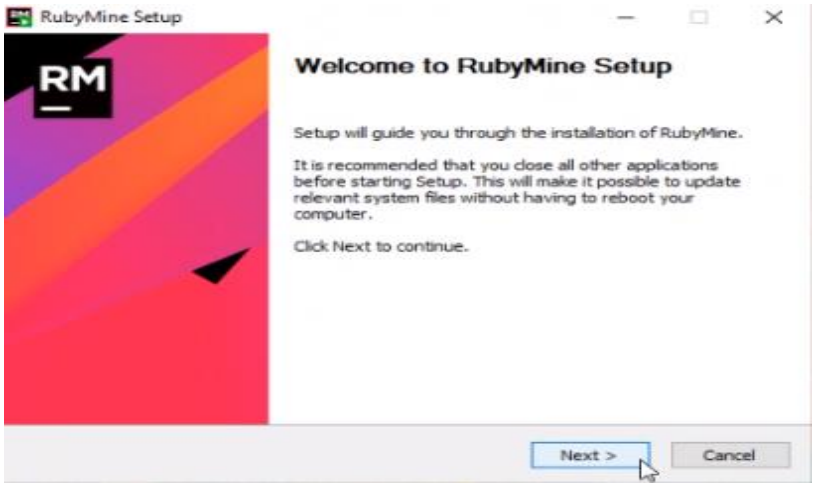


- **Passo 5:** Assim que tudo estiver nos conformes a pagina ficara da seguinte forma:

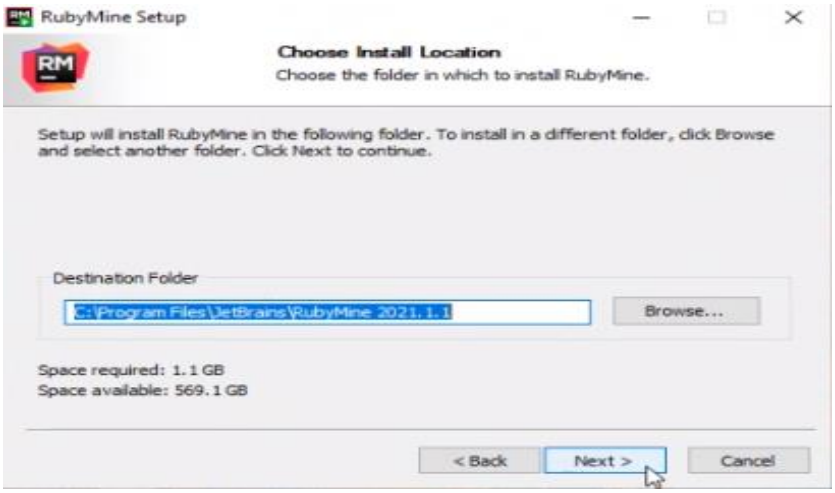


- **Passo 6:** Vá até o diretório onde você deixou o arquivo de setup e execute-o, ele aparecerá da seguinte forma:

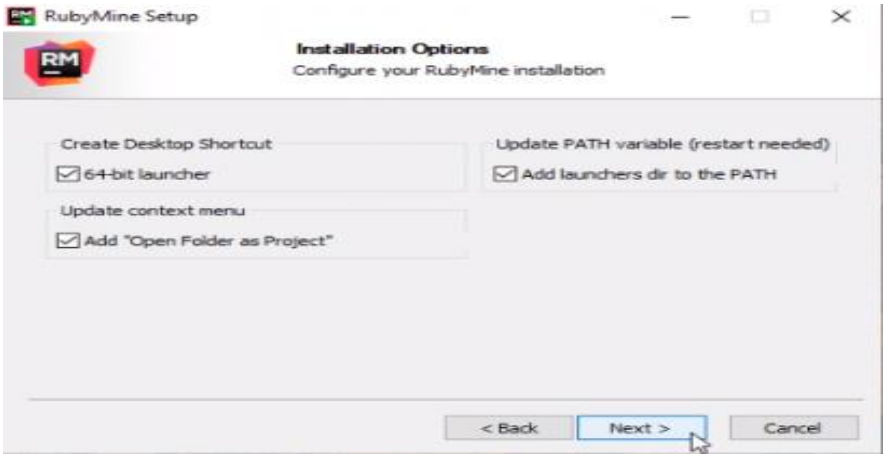
- **Passo 7:** Assim que iniciar o Setup a tela abaixo aparecerá. Para continuar a instalação clique em *next*:



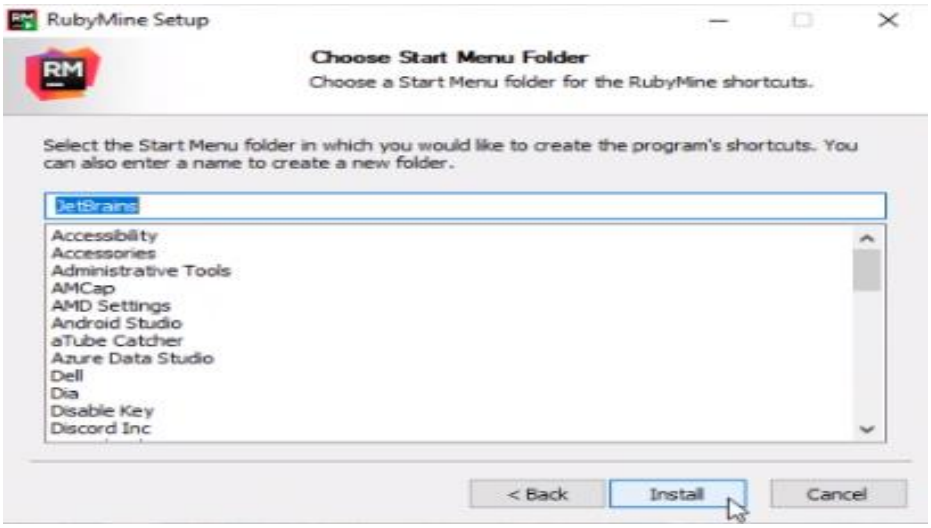
- **Passo 8**: Escolha o diretório onde ficará todos os arquivos do programa, clique em *next* novamente:



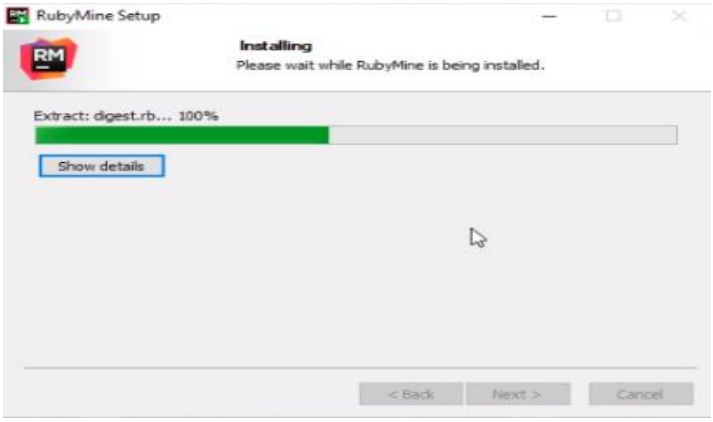
- **Passo 9:** Marque todas as "caixinhas", elas instalam os arquivos que você precisará para dar continuidade ao curso, e claro, clique em *next* novamente:



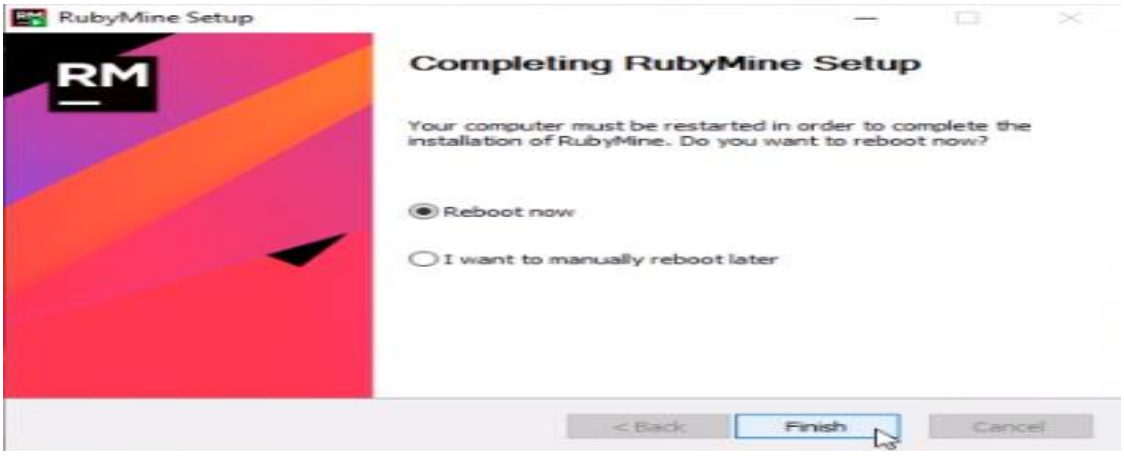
- **Passo 10:** escolha a pasta que deseja inserir os programas, e claro clique em *install* para continuar:



- **Passo 11**: Agora o programa será instalado de fato:

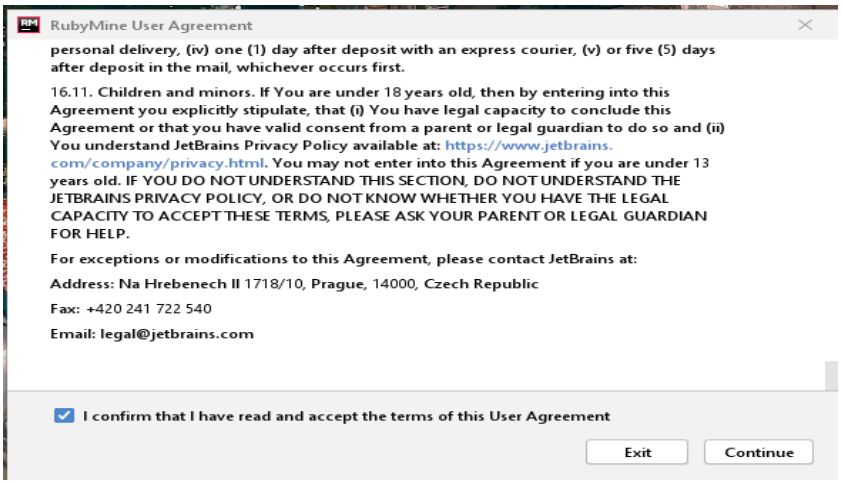


- **Passo 12**: Para terminar o processo do setup é necessário que reinicie a máquina, marque a opção e clique em *finish:*

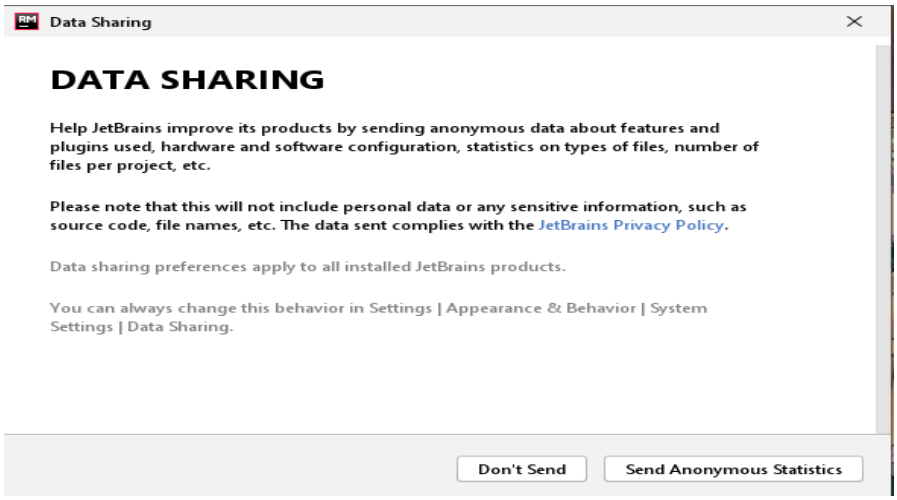


- **Passo 13:** Agora você já tem o programa de fato na sua área de trabalho, o arquivo fica da seguinte forma:

- **Passo 14**: ao iniciar o programa você precisa aceitar novamente os termos de uso e serviço:

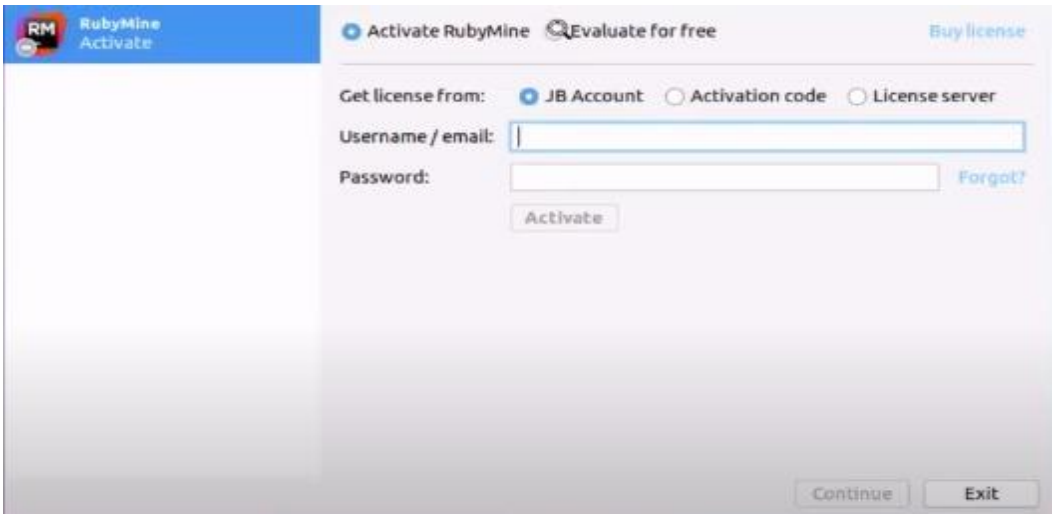


- **Passo 15:** Dando sequência você verá a seguinte tela, nela você poderá escolher se deseja compartilhar seus dados com a empresa JetBrains, você terá livre arbítrio para escolher como prosseguir:

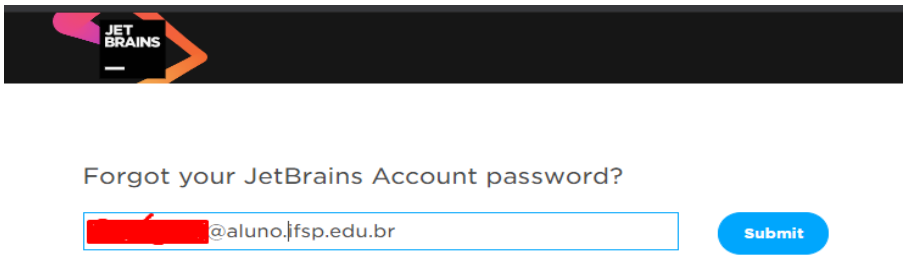


- **Passo 16:** Lembra da sua conta que você cadastrou anteriormente? Em nenhum momento uma senha foi inserida, para isso vamos ter que retornar ao site para

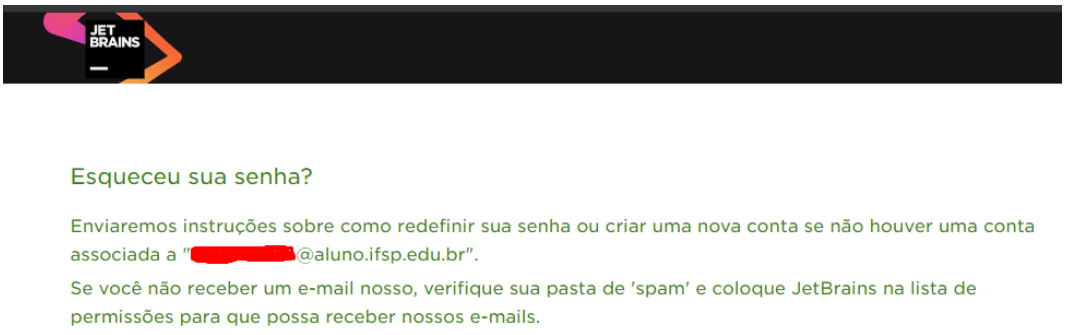
estabelecermos a sua conta e posteriormente usar o pacote educacional. Clique em *Forgo*t para fazer este retorno:



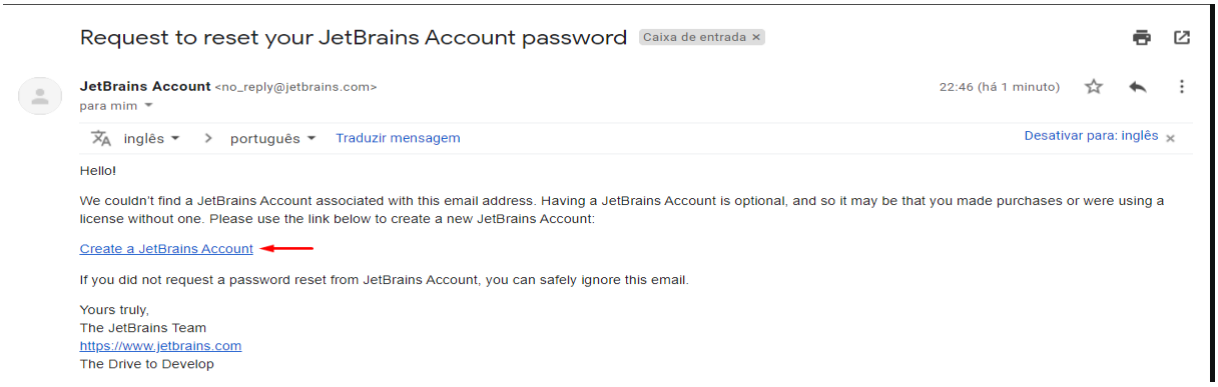
- **Passo 17:** Ao apertar o *forgot* uma guia será aberta no seu navegador. Insira o e-mail que será utilizado no cadastro do JetBrains:



- **Passo 18:** A equipe do JetBrains te enviará um e-mail, verifique sua caixa de entrada e a caixa de spam do seu e-mail:

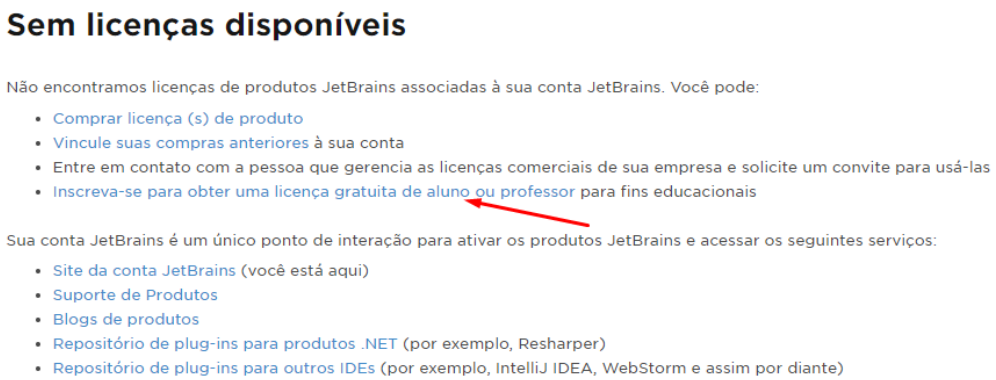


- **Passo 19:** Dentro desse e-mail você receberá um link anexado que é exclusivo para o seu cadastro:



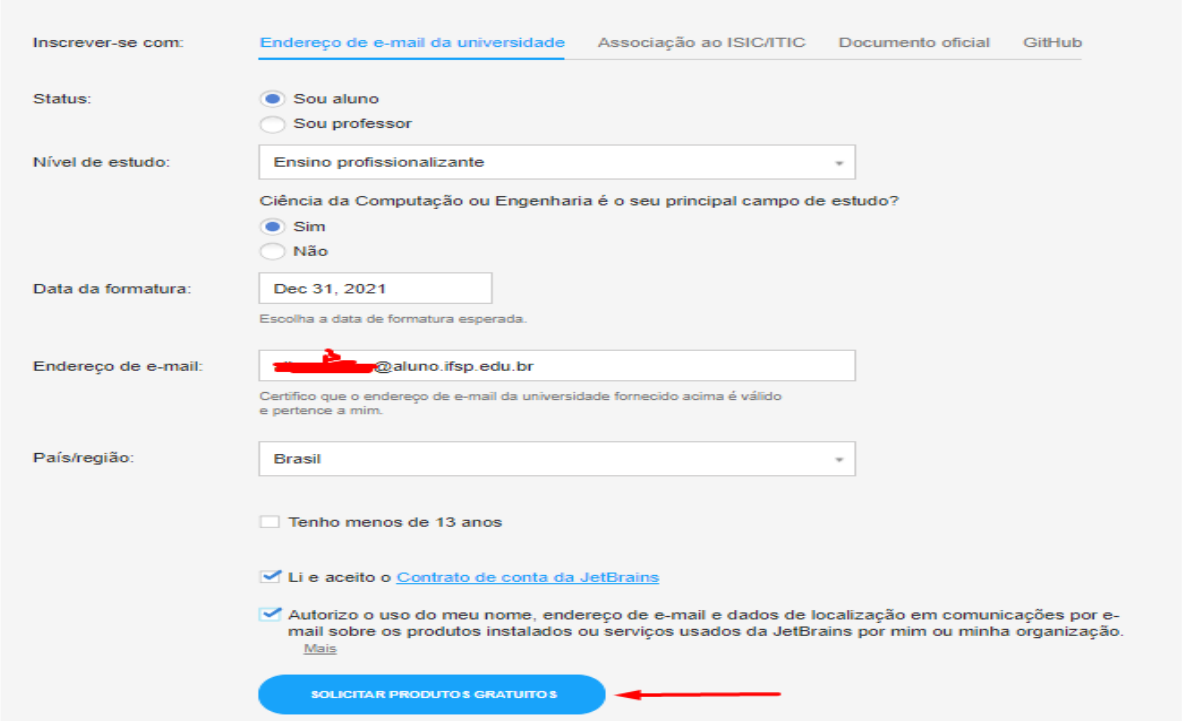
- **Passo 20:** Preencha os dados e crie a sua conta Oficial:

- **Passo 21:** Agora você chegou na tela para adquirir as licenças para usar o programa de forma gratuita, em seguida clique no link indicado, uma nova guia será aberta:

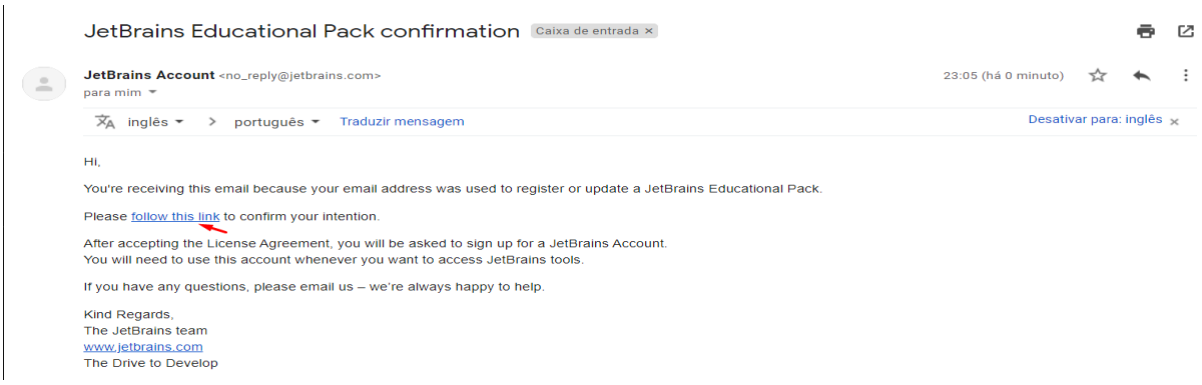


- **Passo 22:** Prossiga clicando no botão indicado no *print* abaixo:

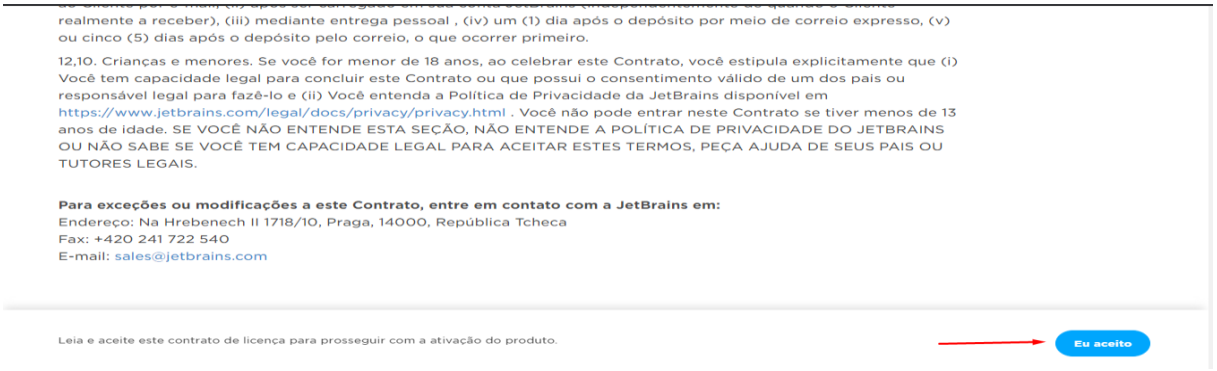
- **Passo 23:** Faça o seu cadastro acadêmico, certifique-se que o e-mail que você usará é um e-mail institucional, neste caso utilizaremos o e-mail institucional fornecido pelo IFSP:



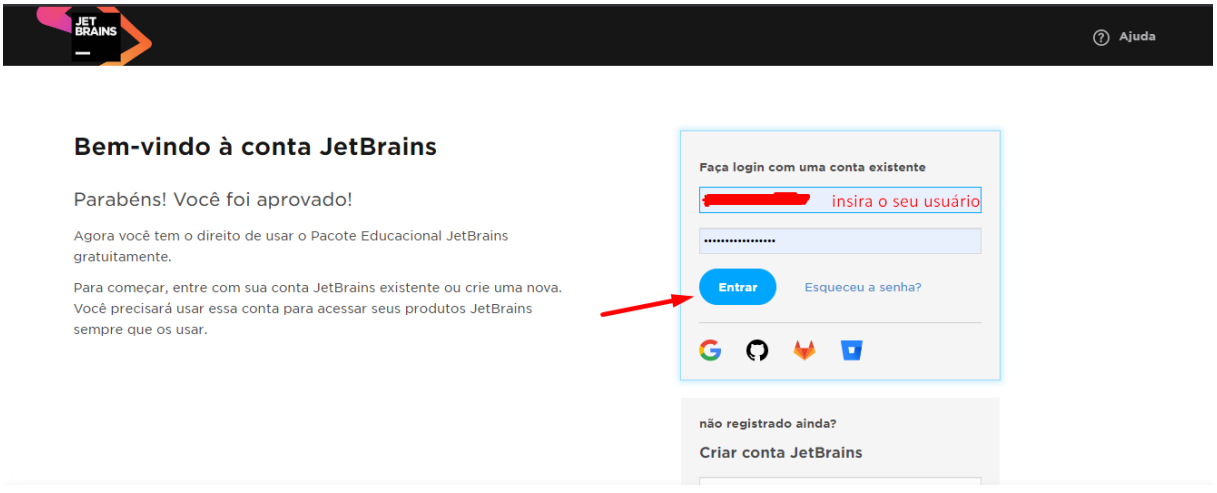
- **Passo 24**: Outro e-mail será enviado para a sua caixa de entrada, nele terá outro link exclusivo para o seu cadastro do pacote acadêmico:



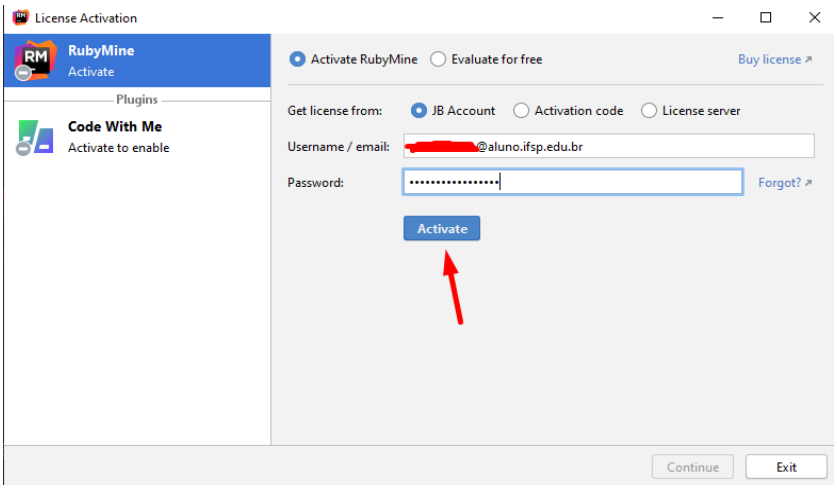
- **Passo 25:** Leia os termos de uso e serviço e no final da página clique em Eu aceito, como indicado na foto:



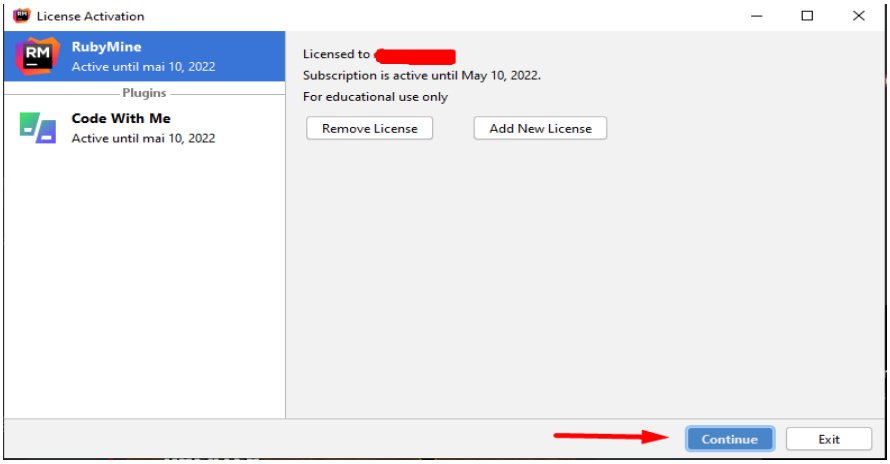
- **Passo 26**: Agora você fará o Log-in oficial no site do JetBrains para verificar se está tudo nos conformes:



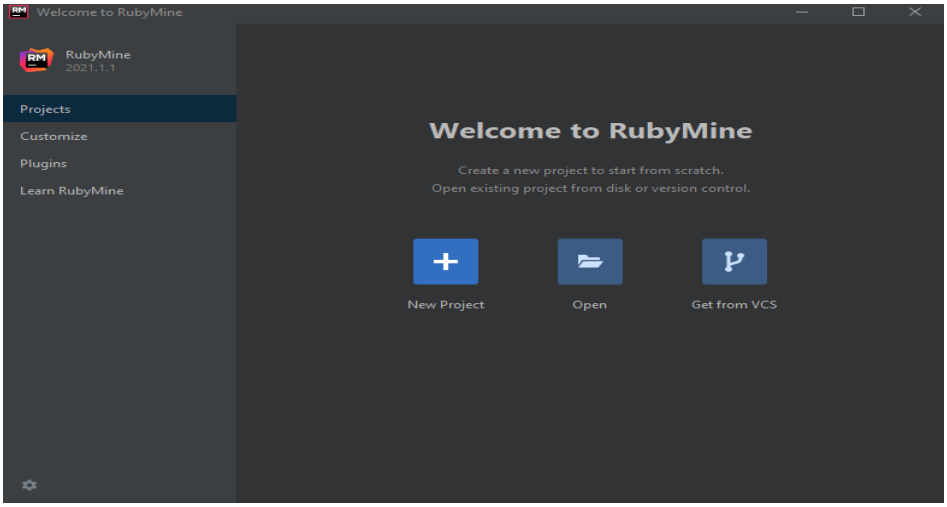
- **Passo 27:** Retornando ao aplicativo insira o log-in que você criou durante o processo, clique em *activate* para confirmar a sua licença:



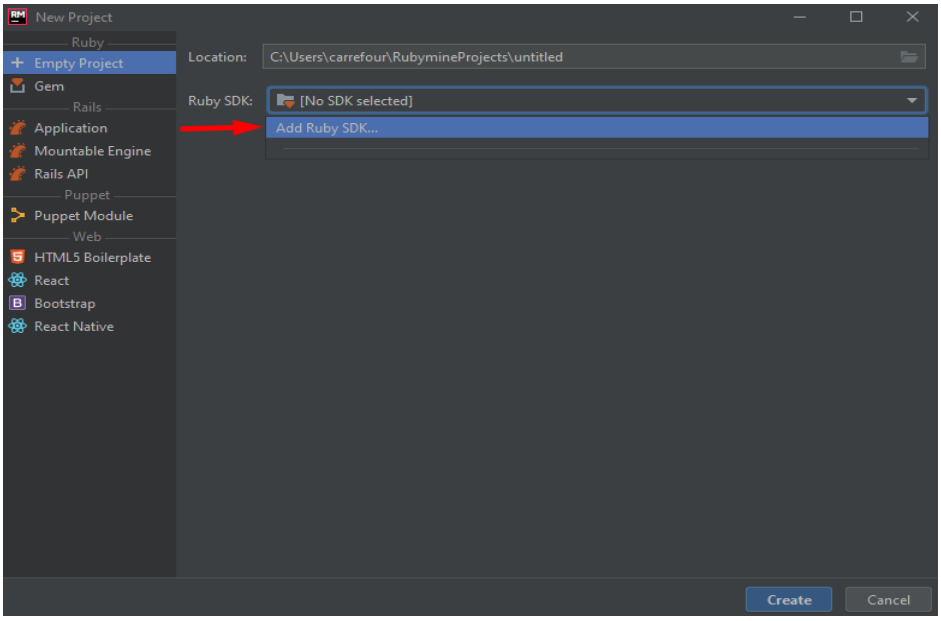
- **Passo 28:** Licença confirmada agora é só apertar o *continue* e abrir a IDE:



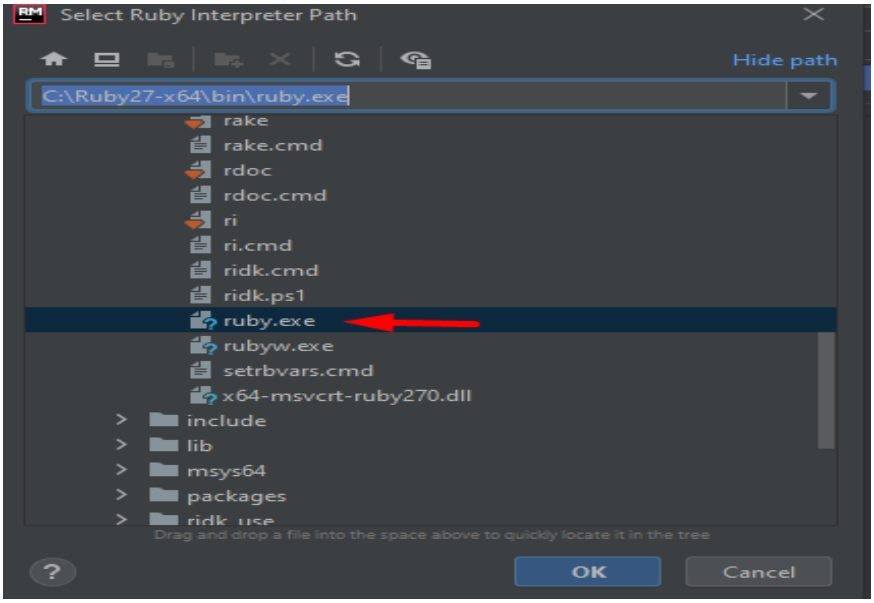
- **Passo 29:** Com a IDE aberta clique em new Project para começar os trabalhos:



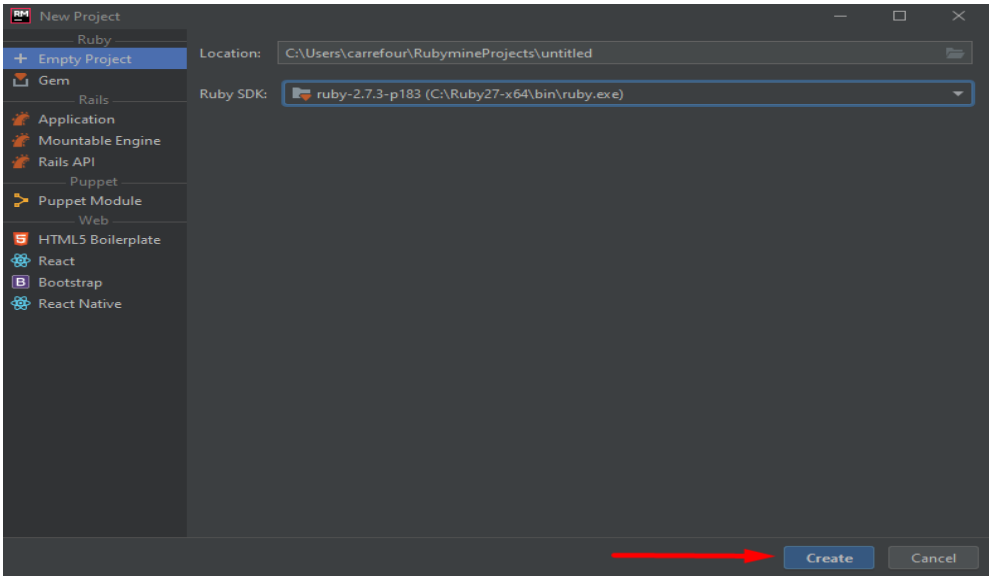
- **Passo 30:** Ao apertar em New Project aparece uma janela ao lado com as funcionalidades da IDE e um acesso rápido aos seus projetos, como ainda não nada em mãos clique em Empty Project e depois em Add Ruby SDK:



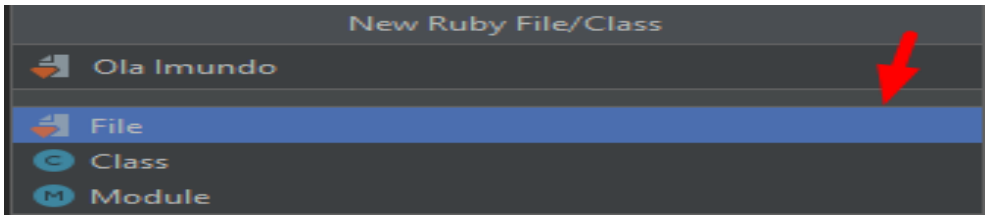
- **Passo 31:** Selecione o Ruby.exe para começar a utilizar a linguagem Ruby:



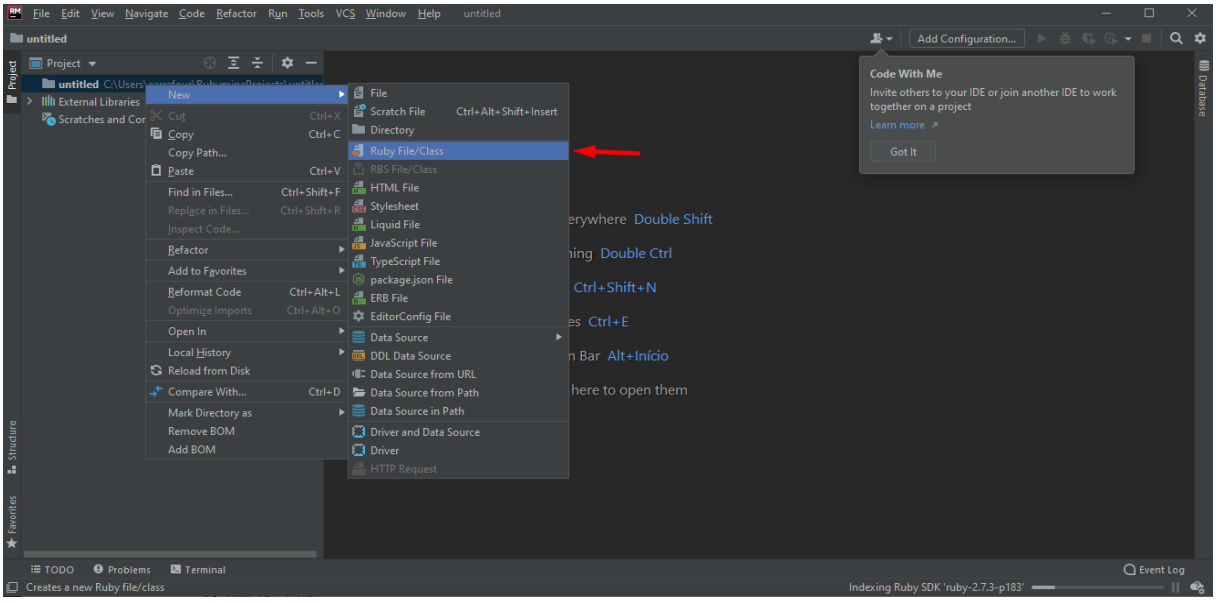
- **Passo 32:** Crie o seu Projeto clicando em *create:*



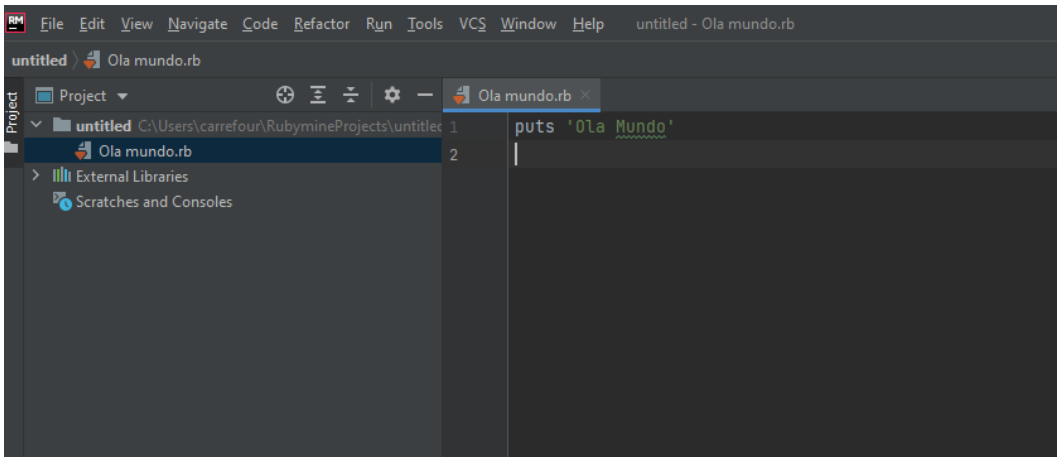
- **Passo 33:** Aperte com o botão direito do mouse em untitled, mantenha o mouse em cima do New e prossiga para o Ruby File/Class:



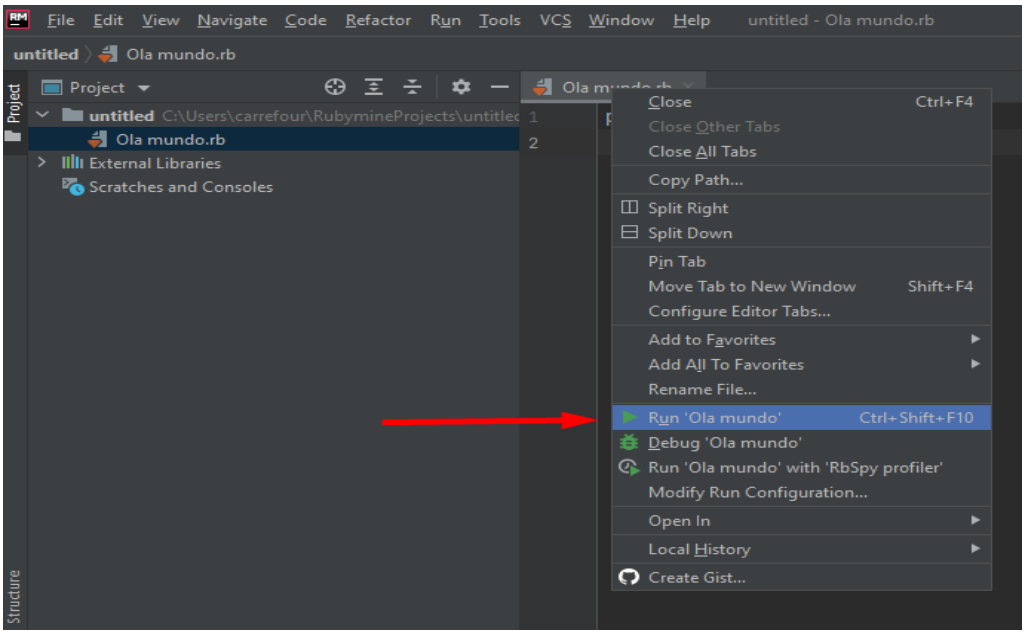
- **Passo 34:** Selecione o File para abrirmos um arquivo editável:



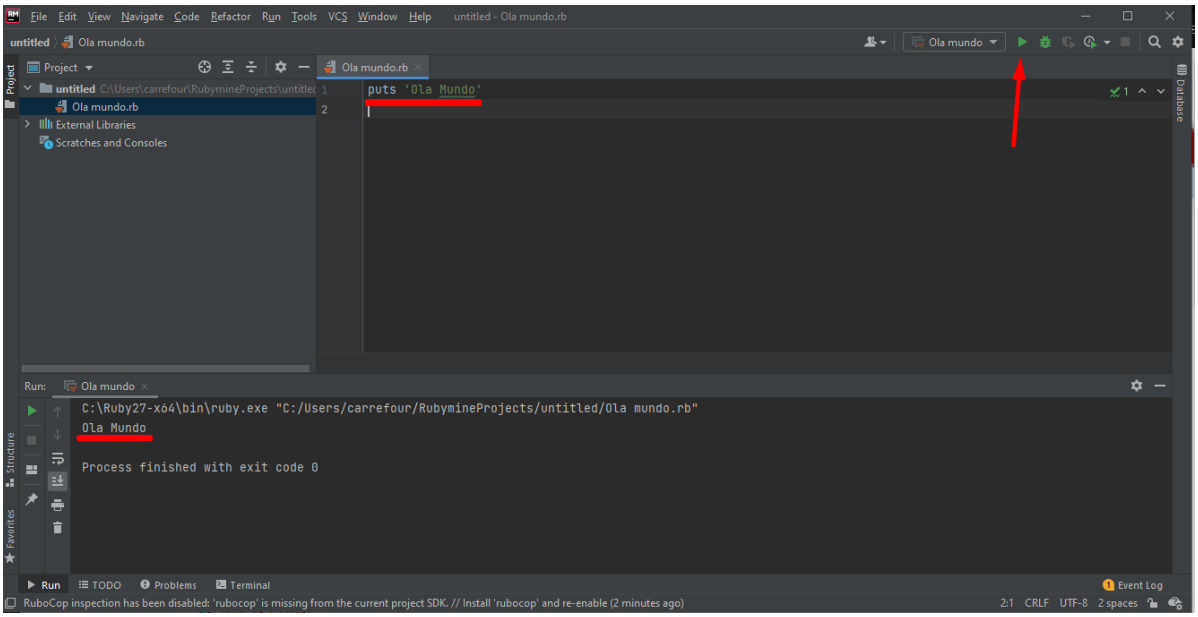
- **Passo 35:** Para criamos o nosso primeiro ‘Ola Mundo’ você precisa usar o puts para inserir algo no terminal:



- **Passo 36:** Logo de cara você perceberá que o arquivo não executará de primeira, para isso clique com o botão direito do mouse no arquivo e prossiga para a opção Run ’Ola mundo’:



- **Passo 37:** Agora você já pode executar os seus códigos a vontade, observe que o que foi digitado no *puts* aparece no terminal logo a baixo:



3. CAPÍTULO 2: FUNÇÕES BÁSICAS NA LINGUAGEM RUBY

Neste capitulo 2 vamos ver as funções básicas e mais utilizadas no mundo da programação para a nossa linguagem. Ao longo do capitulo veremos sobre: os comandos E/S (entrada e saída); operadores aritméticos; funções matemáticas; decisão logica; laços repetição; e vetores. E claro não tem forma melhor de entender o assunto se não observando a pratica, então a seguir temos diversos programas que mostrar muitas das funções que falaremos sobre em funcionamento.

Outro detalhe importante é que como sabemos que os alunos já estão situados com a linguagem C# faremos analogias do Ruby em C# para você, caro leitor, ter entendimento com base em algo que já conhece.

4. PARTE I -COMANDOS E/S, OPERADORES ARITMÉTICOS, FUNÇÕES MATÉMATICAS E DECISÃO LÓGICA

4.2. PROGRAMA 1: ÁREA DO TRIÂNGULO

A partir da digitação da base e altura de um triângulo, o programa deverá calcular sua área e exibi-la no monitor.

```
#Declaração das variáveis que utilizaremos
base = 0
altura = 0
area_t = 0

#nos comandos puts será inserido uma mensagem no terminal
puts("Insira a base")
#nos comandos gets pegaremos o que o usuário digita
base = gets
puts("Insira a altura\n")
altura = gets
#calculo
area_t = base.to_f * altura.to_f
#exibição do calculo
puts("Área do triângulo: #{area_t}")
```

Neste programa, para calcular o valor de seno e cosseno de um triângulo, inicialmente, utilizamos o *puts* para pedir ao usuário que digite o ângulo do triângulo que deseja calcular. Em C# podemos usar o comando *Console.Write* ou *Console.WriteLine* para exibirmos uma mensagem.

O comando *gets* recebe o valor inserido para a variável criada.

Com isso, utilizamos comandos de cálculos em duas variáveis criadas, a “sem” e a “cos”. Nesse caso, as variáveis recebem os comandos *Math.sin* e *Math.cos*, para calcular o seno e cosseno. Os mesmos comandos existem dentro da linguagem C# e são utilizados para mesma funcionalidade.

Utilizamos o *puts* novamente para exibir o valor na tela, juntamente com o “#{}”, onde o mesmo exibe o valor da variável.

4.3. PROGRAMA 2: DESCOBRIR O SENO E COSSENO

O programa deverá pedir para você digitar o valor de um ângulo em graus e na sequência mostrar o valor do seno e cosseno deste ângulo.

```
#Coleta de dados sobre o seno e o cosseno.
puts("Agora digite um dos ângulos do triângulo.")
angulo = gets

#Cálculo final para saber o seno e o cosseno do ângulo
sen = Math.sin(angulo.to_f * Math::PI/180)
cos = Math.cos(angulo.to_f * Math::PI/180)
puts("O cos. do desse ângulo é: #{cos}\nEnquanto o seno é:#{sen}\n")
```

Neste programa, para calcular o valor de seno e cosseno de um triângulo, inicialmente, utilizamos o *puts* para pedir ao usuário que digite o ângulo do triângulo que deseja calcular. Em C# podemos usar o comando *Console.Write* ou *Console.WriteLine* para exibirmos uma mensagem.

O comando *gets* recebe o valor inserido para a variável criada.

Com isso, utilizamos comandos de cálculos em duas variáveis criadas, a “sem” e a “cos”. Nesse caso, as variáveis recebem os comandos *Math.sin* e *Math.cos*, para calcular o seno e cosseno. Os mesmos comandos existem dentro da linguagem C# e são utilizados para mesma funcionalidade.

Utilizamos o *puts* novamente para exibir o valor na tela, juntamente com o “#{}”, onde o mesmo exibe o valor da variável.

4.4. PROGRAMA 3: APROVADO OU REPROVADO PELA MÉDIA

O programa deverá solicitar a digitação de suas 4 notas bimestrais, feito isso deverá calcular e exibir a sua média final (média aritmética entre as 4 notas). Feito isso, deverá também mostrar as mensagens: “Você está aprovado!”, “Você está reprovado!” ou “Você está de exame” de acordo com o seguinte critério: Média final maior ou igual a seis o aluno está aprovado, menor que três reprovado, entre 3 e 6 de exame.

```
#Váriavel usada para divir strings
#Inserção das notas.
puts("Insira a primeira nota.")
nota1 = gets.chomp.to_f

puts("Insira a segunda nota.")
nota2 = gets.chomp.to_f
```

```
puts("Insira a terceira nota.")
nota3 = gets.chomp.to_f

puts("Insira a quarta nota.")
nota4 = gets.chomp.to_f

#Cálculo da média.
media = (nota1 + nota2 + nota3 + nota4) / 4

#Condições para definir se estará aprovado, reprovado ou de
exame.
if(media >= 6)
  puts ("O aluno está... APROVADO! Com uma bela média de
#{media}, parabéns!")
else
  if(media < 3)
    puts ("O aluno está... REPROVADO! Um #{media} não foi o
suficiente, mas não desanime, estude ainda mais!")
  else
    puts ("O aluno está... DE EXAME! Sua média foi um
#{media}, mas ainda há esperanças!")
  end
end
```

Primeiramente, nós pedimos para o usuário digitar as suas notas através do comando *puts*, e utilizamos o *gets.chomp.to_f* para puxar o valor deferido.

O comando *chomp* serve para não contar espaços vazios, como o *enter* no final da frase. Em C#, existe um método semelhante chamado *trim*, que remove espaços em branco no começo e fim da *string*. E juntamente com o *gets.chomp*, há o .to_f, que serve para que o valor receba a conversão para um valor *float*, ou seja, um valor quebrado, como média, peso, etc. Em C# poderíamos usar o comando *float.Parse*. Lembrando que os dados podem ser convertidos em outros tipos além do to_f, para especificar qual você deseja usar um Ruby é necessário colocar o .to_inicialDoTipoDesejado.

Após essas perguntas, nós criamos a variável média e fizemos o cálculo nela, dessa forma, ela obteve o valor para si. Com esse valor em mãos, criamos a condição *If* e *else* para ver se o aluno foi aprovado ou reprovado. Um exemplo: se a média for maior que 6, exibirá a mensagem do comando *puts*, e com a utilização da "#{}" para exibir o valor da variável média. Diferenciando assim, da estrutura do bloco de código que encontra-se no C#, pois a exibição estaria adentrada entre chaves "{ }" e não precisaria de *end* para que seja finalizado o programa. Exemplificando a utilização em C#:

if(condição)

{

bloco de códigos

}

Observação: If = Se / Else = Se não

4.5. PROGRAMA 4: CALCULADORA SIMPLES (IF)

O programa deverá nos solicitar a digitação de dois números e um caractere, sendo que este poderá ser “+”, “-”, “*” ou “/”. Mediante o caractere digitado, o programa fará o respectivo cálculo e exibir o resultado. Se o caractere não corresponder a nenhum dos 4 caracteres em questão, ele irá exibir mensagem de erro. Este programa obrigatoriamente deverá ser feito usando um ninho de if’s.

```
.#Váriavel usada para dividir strings.
divisores = [' ', '+','-','*','/']

#Insersão de dados
puts("Monte uma conta com dois números, só é permitido usar:
+, -, * ou /")
full_calculo = gets

#Separando a string
calculo = full_calculo.split(Regexp.union(divisores))

#Cálculo apenas com If e Else
if(full_calculo.include? "+")
 result = calculo.first.to_f + calculo.last.to_f
 puts("O resultado de #{calculo.first.to_f} +
#{calculo.last.to_f} = #{result}")
else
 if (full_calculo.include? "-")
  result = calculo.first.to_f - calculo.last.to_f
  puts("O resultado de #{calculo.first.to_f} -
#{calculo.last.to_f} = #{result}")
 else
  if (full_calculo.include? "*")
   result = calculo.first.to_f * calculo.last.to_f
   puts("O resultado de #{calculo.first.to_f} *
#{calculo.last.to_f} = #{result}")
  else
   if (full_calculo.include? "/")
    result = calculo.first.to_f / calculo.last.to_f
```

```
    puts("O resultado de #{calculo.first.to_f} /
#{calculo.last.to_f} = #{result}")
   else
    puts("Use os símbolos indicados, se não quebra o programa
:d")
   end
  end
 end
end
```

Inicialmente, foi instanciado uma variável para receber as operações matemáticas básicas.

O comando *puts* foi utilizado para a entrada de dados que o usuário vai colocar, sendo esses dados uma operação matemática simples.

O *include* é utilizado para verificar se existe um determinado caractere, já o *first* é utilizado para selecionar o primeiro item separado do resultado de uma separação de *string*, enquanto o.*last* retorna o último elemento.

O método *split*, também existente na linguagem C#, é utilizado como método para divisão de *strings*, com base em um único caractere ou sequência estática de caracteres.

A estrutura de seleção utilizada nesse programa foi *If* e *Else* (Se e Senão), que testa as opções e condições possíveis para calcular as operações.

4.6. PROGRAMA 5: CALCULADORA SIMPLES (SWITCH)

Idem ao exercício anterior, porém agora a solução deverá ser desenvolvida usando um switch-case.

```
#Váriavel usada para divir strings.
divisores = [' ', '+','-','*','/']

#Insersão de dados
puts("Monte uma conta com dois números, só é permitido usar:
+, -, * ou /")
calculo = gets.chomp

#Separando a string
sinal = calculo[/\W+/]
sinal = sinal.split(" ").to_s
calculo = calculo.split(Regexp.union(divisores))

#Cálculo apenas com switch-case
case sinal.to_s
when ["+"].to_s
 result = calculo.first.to_f + calculo.last.to_f
```

```
 puts("O resultado de #{calculo.first.to_f} +
#{calculo.last.to_f} = #{result}")
when ["-"].to_s
 result = calculo.first.to_f - calculo.last.to_f
 puts("O resultado de #{calculo.first.to_f} -
#{calculo.last.to_f} = #{result}")
when ["*"].to_s
 result = calculo.first.to_f * calculo.last.to_f
 puts("O resultado de #{calculo.first.to_f} *
#{calculo.last.to_f} = #{result}")
when ["/"].to_s
 result = calculo.first.to_f / calculo.last.to_f
 puts("O resultado de #{calculo.first.to_f} /
#{calculo.last.to_f} = #{result}")
else
 puts("Use os símbolos indicados, se não quebra o programa
:d")
end
```

Inicialmente, foi instanciado uma variável para receber as operações matemáticas básicas.

O comando *puts* foi utilizado para a entrada de dados que o usuário vai colocar, sendo esses dados uma operação matemática simples.

O método *split*, também existente na linguagem C#, é utilizado como método para divisão de *strings*, com base em um único caractere ou sequência estática de caracteres.

A estrutura de seleção utilizada nesse programa foi *Switch* case, que testa as opções e condições possíveis para calcular as operações. Em C#, também podemos usar o *switch-case*, com leves diferenças na sintaxe. Em C# o switch se apresenta da seguinte forma:

```
Switch (expressão ou condição)
{
    case x:
    // code block
    break;
    case y:
    // code block
    break;
    default:
    // code block
    break;
```

}

Ao longo da **PARTE I** desta apostila você foi apresentado a diversos operadores aritméticos e as opções que temos para criar as decisões lógicas que a linguagem nos oferece, então vale uma breve menção para eles no fim dessa parte da apostila:

Dos operadores aritméticos o único que ficou restando para ser apresentado é o '%', conhecido como módulo, ou simplesmente o resto da divisão.

Das condições possíveis ainda temos algumas para apresentar, dentre essas temos: '>' o maior que; '<' menor que;'>=' maior ou igual; '<=' menor ou igual; '!=' diferente de; e '==' igual a. quando se trata de estrutura de decisão lógica estes são os operadores mais utilizados.

5. PARTE II - LAÇOS DE REPETIÇÃO, VETORES E MATRIZES

5.2. PROGRAMA 1: TABUADA

A partir da digitação da base e altura de um triângulo, o programa deverá calcular sua área e exibi-la no monitor.

```
#Declaração das variáveis que utilizaremos
base = 0
altura = 0
area_t = 0

#nos comandos puts será inserido uma mensagem no terminal
puts("Insira a base")
#nos comandos gets pegaremos o que o usuário digita
base = gets
puts("Insira a altura\n")
altura = gets
#calculo
area_t = base.to_f * altura.to_f
#exibição do calculo
puts("Área do triângulo: #{area_t}")
```

Inicialmente, criamos três variáveis para conseguir fazer o cálculo do triângulo, base, altura e área. Ao contrário do C#, no Ruby as variáveis não precisam ter seu tipo previamente declarado. Em C# antes do nome da variável seria necessário digitar o tipo, como *int, double, float, string, boolean* dentre outros.

O *puts* serve para imprimir uma mensagem qualquer na exibição do console.

Após pedirmos para ele inserir os dados, utilizamos o comando *gets* para que a variável receba um valor digitado pelo usuário. Em C#, utilizaríamos o comando *Console.ReadLine*.

É perceptível que estamos utilizando uma conta para descobrirmos o valor da área do nosso triangulo. Para tal, usamos os operadores aritméticos '+' e' =' para fazer o cálculo.

Logo depois, o programa realizará o cálculo da área e o resultado será inserido em uma outra variável. Com o resultado em mãos, utilizamos o "#{}", para exibir o valor que foi adquirido. Esse #{} equivale aos placeholders que existem em C#, geralmente apresentados como {0}.

O programa deve começar nos solicitando a digitação de um número, sendo ele inteiro e positivo. A partir de então o programa deverá exibir na tela a tabuada deste número. A entrada de dados não deverá ser validada, vamos acreditar que o usuário sempre digitará um valor devido.

```
#Coleta de dados
puts"Digite um número positivo e inteiro, confio em você. :D"
num = gets.chomp.to_i

#Conta
puts "================================="
for i in 1..10
 puts "#{num} x #{i} = #{num * i}"
end
puts "================================="
```

Inicialmente, utilizamos o comando *puts* para saber qual o valor ele gostaria de ter uma tabuada, e a variável "num" recebe esse valor por conta do *gets.chomp*, e já recebendo uma conversão para inteiro, a partir do comando *.to_i*.

E sabendo o valor indicado, nós utilizamos a condição for para a variável "i" receber os valores de 1 até 10, e com isso o *puts* exibe os valores recebidos (juntamente com o #{}, onde o mesmo mostra o valor de uma variável) conforme os números dados ao "i" vão se alterando. Em C# também se usa a estrutura de repetição for, mas com diferenças na sintaxe, pois é composta por três expressões separadas por ponto e vírgula, as quais controlam o início do loop e as condições de como cada interação irá executar, e, também, com o bloco de código entre chaves. A sintaxe do for em C# é: for (int contador = 0; contador < 5; contador++). Bem diferente do Ruby, não é mesmo?

5.3. PROGRAMA 2: NÚMEROS EM SÉRIE

O programa deverá exibir na tela os “n” primeiros termos da série: 2, 5, 10, 17, 26... onde o valor de “n” deverá ser inicialmente digitado. Em tempo esclareço que tal série tem como termo geral (x2 + 1) onde x = {1, 2, 3, 4, 5 ...}.

```
#Definição de quantos números irão aparecer
puts"Digite os primeiros números da série."
num = gets.chomp.to_i

#Conta em si
for i in 1..num
 puts "#{i ** 2 + 1}"
end
```

O programa a seguir serve para o usuário indicar quantas vezes o cálculo (x^2+1) será feito com os termos 1,2,3...

Inicialmente, utilizando o *puts*, perguntamos quantas vezes será feito o cálculo, com isso, utilizando o comando *gents.chomp* a variável “num” recebe esse valor e já recebe a conversão para número inteiro (*.to_i*).

Após isso, a variável “i” acaba recebendo os valores do termo até chegar no valor indicado pelo usuário. Com o valor de “i” se alterando, ocorre o cálculo onde “i” é elevado a 2 e somado a 1.

5.4. PROGRAMA 3: FIBONACCI

Temos aqui um clássico, o programa deverá listar no vídeo os termos da série de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21 ...) menores que 1000.

```
x = 0
y = 1
tot = 1

#Fibonacci
while (tot < 1000)
 puts"#{tot}"
 tot = x+y
```

```
 x = y
 y = tot
end
```

Como o programa não tem entrada de dados pelo usuário, e sim apenas visualização de resultados, nós temos um *while* como condição que mantém a variável "*tot*" menor que 1000. A estrutura de repetição *while* também existe em C#, apresentando-se da seguinte forma:

```
while (condição)
{
        //bloco de código
}.
```

5.5. PROGRAMA 4: GARANTINDO QUE O 2º VALOR DIGITADO É O MAIOR

Entrar com dois valores via teclado, onde o segundo deverá ser maior que o primeiro. Caso contrário solicitar novamente a digitação do segundo valor, o que deve ser repetido até que o usuário atenda a condição definida.

```
#Delcaração das variaveis a serem utilizadas
v1=0
v2=0

#inserção de dados
puts("Digite dois valores distintos, o segundo devera ser
maior que o primeiro pra encerrar o programa")
puts("\nDigite o Primeiro valor")
v1=gets.to_f
puts("Digite o Segundo Valor")
v2 = gets.to_f

#condição while para garantir que o segundo numero seja o
maior
while v1>=v2
 puts("Digite dois valores distintos, o segundo devera ser
maior que o primeiro pra encerrar o programa")
 puts("\nDigite o Primeiro valor")
 v1=gets.to_f
 puts("Digite o Segundo Valor")
 v2=gets.to_f
end
```

O programa serve para solicitar para o usuário digitar 2 valores, onde o segundo valor tem que ser, obrigatoriamente, maior que o primeiro.

Inicialmente, criamos duas variáveis recebendo o valor de 0.

Posteriormente, utilizando o comando *puts*, pedimos para ele digitar os 2 valores. As variáveis “v1” e “v2”, utilizando o comando *gets* acabam recebendo esses valores digitados e já convertidos em *float* (.to_*f*), um valor que possui casas decimais.

Após isso, utilizamos o *While* (enquanto), ou seja, enquanto o “v1” for maior que o “v2”, ele terá que digitar novamente os valores, e isso se repetirá até o usuário digitar os valores corretamente. Em c#, sua sintaxe declara a condição entre parênteses com o bloco de código entre chaves. A estrutura de repetição *while* apresentando-se da seguinte forma:

while (condição)
{
    //bloco de código
}.

5.6. PROGRAMA 5: VALIDAÇÃO DO SEXO DIGITADO

Entrar via teclado com o sexo de determinado usuário, aceitar somente “F” ou “M” como respostas válidas, caso o valor digitado seja indevido, o processo será repetido até que se digite um dos dois caracteres válidos.

```
#inserção dos dados
puts("Seu nome:")
nome = gets.chomp
puts("Digite o seu sexo F ou M")
sexo = gets.chomp

#condição para digitar o sexo entre F ou M
while sexo != 'F' && sexo != 'M'
  puts("Digite o seu sexo F ou M")
  sexo = gets.chomp
end

#saida de dados
puts ("Nome: #{nome}\nSexo: #{sexo}")
```

O programa dá início com o pedido da entrada de dois dados por parte do usuário, sendo eles nome e sexo.

A condição *while* vai impedir toda entrada de dados que seja diferente de “F” ou “M”. Em c#, sua sintaxe declara a instrução *while* e a condição entre parênteses com o bloco de código entre chaves, já sua exibição é por meio do comando *console.write*.

Caso o usuário insista em colocar dados diferentes de “F” ou “M”, a mesma mensagem pedindo pela entrada do sexo será exibida.

Por fim, na saída de dados, o programa apresenta o nome colocado pelo usuário e o sexo.

5.7. PROGRAMA 6: FATORIAL DO NÚMERO DIGITADO

O programa deverá nos permitir a digitação de um número inteiro e positivo, essa entrada deverá ser repetida até que o usuário satisfaça a condição. Feito isso o programa deverá calcular e exibir o fatorial deste número. Ao fim questionar se desejamos continuar ou não, e essa entrada só deverá aceitar como resposta “S” ou “N” e a pergunta deverá ser repetida até que o usuário responda corretamente. Se a resposta for “S” então deveremos voltar ao início do programa e repetir tudo novamente, caso contrário o programa deverá ser encerrado.

```
#Pede o Número, faz o fatoral, e pergunta se quer coninuar.
def verificar
 puts "Digite um número positivo.(Caso digite alguma letra,
será consideirado como 0)"
 num = gets.to_i

 #Verifica se o número corresponde ao pedido.
 if (num.to_i < 0)
  while (num < 0)
   puts "O número tem que ser positivo, meu anjo. Digite
novamente, mas positivo."
   num = gets.to_i
  end
 end

 #Fatorial
 if (num == 0)
  puts"Fatorial: 0"
 else
  puts"Fatorial: #{(1..num).inject(:*)}"
 end

 #"Arruma" o pedido para o próximo passo
 puts "Deseja continuar?"
 resp = gets.upcase
```

```
 resp = resp[0..0]
 return resp
end

#Responsável por sair do programa ou voltar a fazer a conta.
def Resposta(resp)
 while (resp.include? "N" or resp.include? "S") == false
  puts "Digite apenas 'S' caso queira repetir, e 'N' caso não
queira."
  resp = gets.upcase
  resp = resp[0..0]
 end
 if (resp.include? "N") == true

 else if (resp.include? "S") == true
       all
      end
 end
end

#Serve para entrar no loop, caso o usuário queira repetir.
def all
 resposta = verificar
 Resposta(resposta.to_s)
end

#Apenas inicia o programa
resposta = verificar
Resposta(resposta.to_s)
```

Neste programa, ele perguntará ao usuário um valor positivo e inteiro, caso o usuário não digite de acordo, o programa ficará pedindo para ele insira. Após isso, o programa irá calcular o fatorial desse número inserido. Ao fim, o programa irá perguntar se o usuário deseja fazer novamente ou não, digitando apenas “S” ou “N”, caso a resposta solicitada seja indevida, ficará pedindo para digitar novamente.

Inicialmente, através do comando *puts*, pedimos para o usuário digitar um valor positivo e inteiro, e caso ele digite alguma letra, o valor é considerado como 0. Com o valor inserido, a variável “num” irá receber, através do comando *get*, o valor e já convertido, por conta do comando *.to_i*.

Após o número inserido, foi criado um *if* (Se), juntamente com um *While* (Enquanto), para verificar se o número indicado é maior que 0, caso contrário, irá perguntar novamente. Depois da verificação, e o número ser maior do que 0, a variável “num” recebe o valor, e

será feito, através de um *If* e *Else*, o fatorial deste número, juntamente o comando *.inject*, que combina todos os elementos de "num" aplicando uma operação binária, especificada por um bloco ou símbolo que nomeia um método ou operador.

Partindo dessa etapa, o programa irá perguntar se o usuário deseja continuar ou fazer novamente, e com isso, a variável "resp" acaba recebendo essa resposta, juntamente ao comando *.upcase*, o qual transforma todos os caracteres em maiúsculos. Porém, em c#, se utiliza *String.ToUpper*.

Logo após, criamos uma função que servirá para retornar ou sair do programa, e para isso, foi utilizado um *While*, caso ele digitasse "S", o programa voltaria para o começo, já no caso que digitasse "N", o programa se encerraria.

Logo abaixo, podemos ver a função que foi criada para entrar em loop, caso o usuário aceitasse repetir novamente o programa.

Os próximos programas terão foco na utilização de *Arrays*, preste bastante atenção!

5.8. PROGRAMA 7: VALORES DA MATRIZ EM ORDEM INVERSA À DA DIGITAÇÃO

O programa deverá nos permitir digitar e armazenar dez números na memória do computador. Feito isso exibirá os valores digitados em tela na ordem inversa à da digitação.

```
#array para armazenar os dados
x = Array.new(10)
#inserção dos dados
for a in 0..9 do
 puts("Digite o #{a+1}°:")
 x[a] = gets.chomp.to_i
end

#metodo do Ruby feito para mostrar os dados do array
inversamente ordenados a digitição
x.reverse!
x.each do | valor |
 puts valor
end
```

O programa a seguir serve para o usuário digitar 10 valores, e os valores serão armazenados no computador e exibidos a digitação inversa.

Primeiramente, criamos um vetor com 10 índices, após isso, utilizamos de um “for” que ficara repetindo para o usuário digitar um número, até preencher o vetor, assim como em C#, mas alterando que há três expressões separadas por ponto e vírgula e o bloco de código dentro de chaves.

Com isso, armazenamos esses valores no *array*, utilizamos do comando “get” para pegar e armazenar o valor, e o *chomp* para pegar precisamente o valor (sem contar espaços etc.).

Finalizando com as utilizações do comando *reverse* para inverter os valores e, também, o comando *each*, que retorna os itens da coleção. E por último, a exibição com o *puts*.

5.9. PROGRAMA 8: SOMAR E DESCOBRIR A MÉDIA DOS VALORES DO VETOR

O programa deverá nos permitir digitar e armazenar dez números na memória do computador. Feito isso criar um laço capaz de calcular a somatória desses 10 valores e então exibir a média desses valores.

```
Array onde os números estão armazenados
values = Array.new(10)

#inserção de dados
for a in 0..9 do
 puts("Digite o #{a+1}°:")
 values[a] = gets.to_f
end

#saida da somatoria
puts("Somatória: #{values.sum}\nMédia#{values.sum/10}")
```

Inicialmente, foi instanciado um *array* para criar uma partição e armazenar 10 dados. Em C#, o programador também disfruta de a opção de poder criar uma variável homogênea composta, o *array*, popularmente conhecido como vetor.

Logo após isso, entramos em um for que vai forçar o usuário a colocar 10 valores, um por um. Quando o último valor for adicionado, o usuário irá sair do for e entrar na função de somatória e apresentar a média entre eles, juntamente com o *sum*, que soma os valores dentro de um vetor, assim como no c#.

Em C# também utilizamos o for, mas com diferenças na sintaxe, pois a estrutura de repetição é composta por três expressões separadas por ponto e vírgula, que comandam o começo do loop e as condições de como cada interação irá executar, e, também, com o bloco de código entre chaves.

5.10. PROGRAMA 9: DESCOBRINDO O MENOR VALOR DENTRO DO VETOR

O programa deverá nos permitir digitar e armazenar dez números na memória do computador. Feito isso criar um laço capaz de identificar o maior e o menor dos números digitados e exibi-los ao final.

```
#array para armazenar os numeros
x = Array.new(10)

#inserção dados
for a in 0..9 do
 puts("Digite o #{a+1}°:")
 x[a] = gets.chomp.to_f
end

#condição para verificar o maior
maior = x[0];
for ma in 0..9 do
if (x[ma] > maior)
 maior = x[ma]
end
end

#condição para verificar o menor
menor = x[0];
for me in 0..9 do
 if (menor>x[me])
  menor = x[me]
 end
end

#saida dos dados
puts("Maior: #{maior}\nMenor: #{menor}")
```

O programa serve para pedir ao usuário que digite 10 números, os quais serão armazenados na memória do computador. Após a digitação, o programa irá mostrar o maior e menor valor digitado.

Inicialmente, foi criado um *array* (vetor) para armazena os números.

Posteriormente, nós criamos a inserção de dados, onde criamos a condição de for para ficar pedindo os valores até que chegasse ao 9. Lembrando que um vetor sempre começa do 0. Após isso, o vetor armazena os números com a utilização do *gets.chomp* e já convertido para *float*, com a utilização do .to_f.

Depois, criamos a condição para verificar o maior valor indicado. E logo após, outra condição para verificar o menor valor. E por fim, a saída de dados, mostrando o maior valor digitado e o menor

5.11. PROGRAMA 10: FAZENDO PESQUISAS DENTRO DO VETOR

O programa deverá nos permitir digitar e armazenar o nome e idade de dez pessoas. Feito isso, devera-os solicitar a digitação de um nome e então proceder a pesquisa e informar a idade do sujeito pesquisado caso ele se encontre armazenado, caso contrário informar o fato através de mensagem “Pessoa não localizada”. Ao final, verificar se uma nova consulta é desejada, validar a resposta do usuário no sentido de só aceitar “S” ou “N”. Obviamente que no caso de nova consulta a digitação dos dez nomes/idades não deverá ser repetido.

```
def consulta
  divisores = [',', ' ', ';', "e"]
  i = 0
  nome = []
  idade = []

  while (i < 10)
    puts("Insira o primeiro o nome e depois a idade da #{i + 
1}° pessoa")
    info_tot = gets.chomp.upcase
    info_tot = info_tot.split(Regexp.union(divisores))

    nome[i] = info_tot.first.to_s
    idade[i] = info_tot.last.to_i

    i = i+1
  end
```

```
  puts"Qual o nome que o sr. deseja encontrar?"
  nome_ver = gets.chomp.to_s.upcase

  for i in 0..9
    nome2 = nome[i]
    if(nome_ver == nome2)
      puts "Existe uma pessoa chamada #{nome2}, e ela tem
#{idade[i]} anos de idade."
      nome_certo = nome_ver
    end
  end

  if (nome_certo != nome_ver)
    puts"Puts... infelizmente não achei alguém com esse nome,
fica pra próxima!"
  end
  puts"Quer fazer mais uma vez? Hehe."
  resp = gets.upcase
  resp = resp[0..0]
  return resp
end

def Resposta(resp)
  while (resp.include? "N" or resp.include? "S") == false
    puts "Digite apenas 'S' caso queira repetir, e 'N' caso
não queira."
    resp = gets.upcase
    resp = resp[0..0]
  end
  if (resp.include? "N") == true

  else if (resp.include? "S") == true
         all
       end
  end
end

def all
  resposta = consulta
  Resposta(resposta.to_s)
end

resposta = consulta
Resposta(resposta.to_s)
```

A primeira parte desse código temos a declaração de variáveis, após isso nós entramos em um loop *while* para a entrada de dados pelo usuário, pedindo para que entre com o nome e idade de dez pessoas.

Depois de sair do loop, o programa realizará uma rotina de busca de dados, onde o usuário é pedido para que entre com os nomes que ele gostaria de procurar, lembrando que serão encontrados apenas os 10 nomes que foram digitados anteriormente.

Visto que não foram encontrados quaisquer dos nomes, o programa encerra a consulta e oferece ao usuário para que realize uma nova utilização do programa.

A todo momento que é exigido as respostas digitadas, o comando *upcase* entra em cena para identificar todos os casos em que o usuário final utilizar letras maiúsculas e minúsculas.

Caso a resposta para reiniciar o programa seja “S”, será reiniciado, senão for, o programa se encerrará.

Nesse código, percebemos o uso de funções em Ruby, que podem ser definidas apenas através da palavra reservada def. Em C#, o processo é um tanto mais complicado, pois segue um sintaxe: static Tipo_de_retorno Nome_da_função(Argumentos).

5.12. PROGRAMA 11: SALA DE CINEMA UTILIZANDO MATRIZ

Determinado cinema tem 20 fileiras (de 1 a 20) com 15 cadeiras (de 1 a 15) cada uma, portanto estamos falando de uma sala com 300 assentos, que deve ser reproduzida através de um *array* de 20 linhas por 15 colunas. O programa deve começar solicitando do usuário a digitação de seu nome, o número da fileira e cadeira em que deseja se sentar, se o assento estiver vazio reservá-lo registrando no *array* seu nome, caso contrário informar que o assento está ocupado. Feito isso o programa deverá nos questionar se desejamos nova reserva, validando nossa resposta e repetindo todo o processo.

```
  #Criação de variáveis globais.
$divisores = [',', ';', ' ', 'e', '[', ']', '"']
$assentos = Array.new(20){Array.new(15)}

  #Responsável por verificar os dados, coletar os dados, e
preparar para uma possível repetição do programa.
  #Coleta de dados
def consulta
puts"Digite seu nome."
nome = gets.chomp.to_s

puts"Agora digite primeiro a fileira depois a cadeira
desejada."
info_total = gets.chomp
```

```
info_total = info_total.split(Regexp.union($divisores))

  i = info_total.first.to_i
  j = info_total.last.to_i

  #Verifica se o assento está ocupado.
if ($assentos[i][j].to_s == "")
  puts "A cadeira #{j} na fileira #{i} agora te pertence!"
  $assentos[i][j] = nome
  else
  puts"Essa cadeira nessa fileira está ocupada."
  end

  #Preparando uma possível repetição do programa.
puts"Deseja repetir o processo?"
resp = gets.upcase
resp = resp[0..0]
return resp
end

  #Responsável por repetir o programa.
def Resposta(resp)
  while (resp.include? "N" or resp.include? "S") == false
    puts "Digite apenas 'S' caso queira repetir, e 'N' caso
não queira."
    resp = gets.upcase
    resp = resp[0..0]
  end
  if (resp.include? "N") == true

  else if (resp.include? "S") == true
         all
       end
  end
end

  #Chama os programas.
def all
  resposta = consulta
  Resposta(resposta.to_s)
end

  #Início do programa
all
```

O programa se inicia instanciando os divisores e a “matriz” para consulta, no Ruby, segundo pesquisas, não existe matriz, então a “matriz”, em questão, é um vetor, dentro de outro vetor, um que representa as linhas, e outro, as colunas. Essas

variáveis são globais, e isso é representado pelo símbolo “$”, antes do nome da variável

Após isso, criamos uma função, que pede a entrada de nome por parte do usuário e, ocasionalmente, o usuário deverá dar entrada com dois valores numéricos para que possa reservar a fileira e a cadeira do cinema, as quais serão armazenados nas variáveis: i = fileira, j = cadeira.

Posteriormente, é dada a entrada na estrutura de repetição if, que irá consultar se esse assento estaria ocupado. Caso esteja, o programa oferece ao usuário a realização de uma nova inserção de dados e consulta, e caso não esteja, ele avisa que está ocupada.

Ainda na mesma função (def consulta) perguntamos se o usuário quer repetir, ou não, o programa. A resposta será armazenada em uma variável local, e será retornada como valor da função, finalizando a função “consulta”.

Agora entramos na segunda função do programa, a função “Resposta”, que será responsável por repetir o todo o processo já dito. Começamos verificando se a resposta que o usuário deu, bate com a resposta que esperamos, com um “while”. Se a resposta for “S”(sim), a função “Resposta” chama a função “all”, Se for “N”(não), ele finaliza o programa, caso não for nenhum dos dois, evidentemente não passará pelo “while”.

Agora estamos na função “all”, que é responsável por chamar todas as outras funções (“consulta” e “Resposta”). Nela, nós criamos uma variável local chamada “resposta”, que armazena o valor da função “consulta”, e já iniciando a função consulta, depois ele passa o valor para a função “Resposta”.

E, assim, nós apenas chamamos a função “all”, e toda a “mágica” acontece.

6. CAPÍTULO 3: INTERFACE GRÁFICA

Neste capitulo 6 vamos ver as funções básicas e mais utilizadas no mundo da programação para a nossa linguagem. Ao longo do capitulo veremos sobre: A criação de aplicações *form* em Ruby, desde a criação de variáveis para os botões, caixas de texto e caixas de seleção. E claro não tem forma melhor de entender o assunto se não observando a pratica, então a seguir temos diversos programas que mostrar muitas das funções que falaremos sobre em funcionamento.

Outro detalhe importante é que como sabemos que os alunos já estão situados com a linguagem C# faremos analogias do Ruby em C# para você, caro leitor, ter entendimento com base em algo que já conhece.

6.2. INTEGRANDO A INTERFACE GRÁFICA À IDE

A fim de produzir esse capitulo, por unanimidade, optamos por utilizar o framework FXRuby, devido a facilidade de instalação. Para instalar o framework é necessário abrir o prompt de comando (cmd), digitar "gem install fxruby" e aguardar.

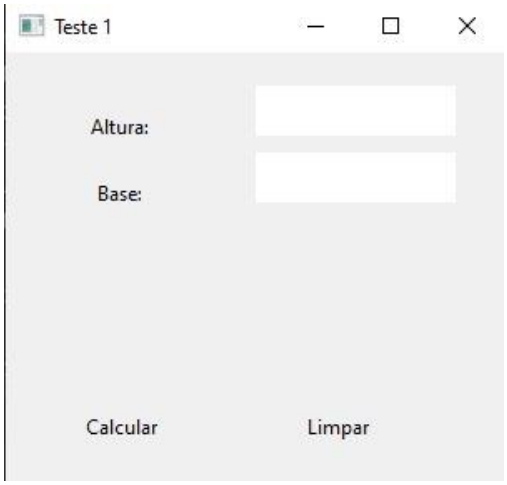
Assim que instalado só precisa abrir a IDE, digitar os comandos require 'fox16' e include Fox, com isso já está pronto para utilizar.

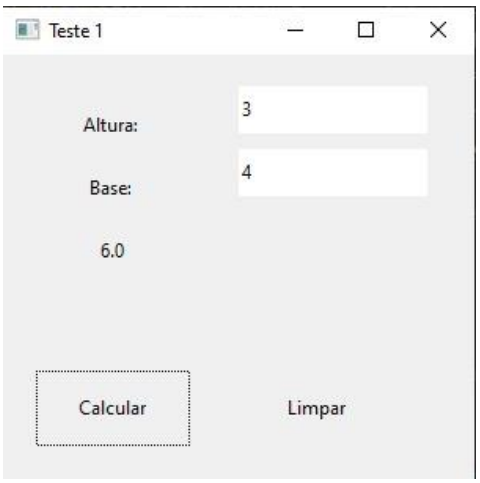
7. APRESENTANDO OS PROGRAMAS

7.2. PROGRAMA 1: ÀREA DO TRIÂNGULO

```
require 'fox16'
include Fox

class GUI < FXMainWindow
  def initialize(app)
    super(app,"Teste 1",:width=>300,:height=>300)
    label1 = FXLabel.new(self,"Altura: ",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>20)
    label2 = FXLabel.new(self,"Base: ",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>60)
    @resultado = FXLabel.new(self,"",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>100)
```

```
    altura = FXTextField.new(self, 30,
:opts=>LAYOUT_EXPLICIT, :width=>120, :height=>30, :x=>150,
:y=>20)
    base = FXTextField.new(self, 30, :opts=>LAYOUT_EXPLICIT,
:width=>120, :height=>30, :x=>150, :y=>60)
    botaoCalcular = FXButton.new(self, "Calcular",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>200)
    botaoLimpar = FXButton.new(self, "Limpar",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>150,
:y=>200)

    botaoCalcular.connect(SEL_COMMAND) do

      alturabtn = altura.to_s
      basebtn = base.to_s
      @resultado.text = ((alturabtn.to_f *
basebtn.to_f)/2).to_s
    end
    botaoLimpar.connect(SEL_COMMAND) do
      altura.text=""
      base.text=""
      @resultado.text = ""
    end
  end


def create
  super
  show(PLACEMENT_SCREEN)
end
end

app = FXApp.new
GUI.new(app)
app.create
app.run
```





- **Programa 1, legenda**

Primeiramente chamamos o "fox", que funciona como uma biblioteca. E Após isto, criamos a classe "GUI", com isso, criamos as "label", onde foi declarado o tipo dela, o tamanho e localização (essa parte foi feita igualmente para os outros, como o "@resultado", "altura", "base", etc.". O "@resultado" só irá aparecer após a conta ser concluída, e a "altura" e "base" são os campos que utilizamos para digitar os valores.

E com isso, virá o botão calcular e o limpar. O botão calcular irá pegar os valores e fazer a conta, além de estar transferindo os valores para "string" (to_s) e também para "float" (to_f).

O botão limpar, irá esvaziar os blocos, deixando-os utilizáveis novamente.

A parte do "def create", é onde cria a tela que irá aparecer, e com isso, temos a parte onde irá executar o programa ao todo, basicamente, ele cria e executa.

- **Programa 1, comparação com C#**

Em C#, a criação de um programa em modo janela torna-se muito mais fácil devido ao fato de a linguagem possuir um framework que nos possibilita criar os forms (formulários/interface gráfica) de forma interativa. Já em relação à linguagem Ruby, a formatação e a inclusão dos forms são feitas de forma manual.

Em Ruby, para que os componentes do form sejam incluídos, é necessário que o programador especifique algumas informações e coloque-as manualmente, como a largura,

altura e posição dentro do form, podemos observar com a declaração da variável que irá armazenar o resultado dos valores inseridos pelo usuário após o cálculo:

```
@resultado = FXLabel.new(self,"", :opts=>LAYOUT_EXPLICIT, :width=>100,
:height=>50, :x=>20, :y=>100)
```

Para definir o tipo do componente, foi utilizado FX (responsável pelo poder das funções) seguido de Label, usado para colocar um texto, juntamente a ".new" que é para adicionar um novo campo no *form*. Dentro dos parênteses encontra-se o self significando o rosto da interface, depois duas aspas indicando que inicialmente o campo está vazio. Antes de declarar o tamanho (*height*), altura (*width*), e as posições (x e y), há também "*:opts=>LAYOUT_EXPLICIT*", código esse que representa a estrutura plana física do *form*.

Em C#, o trabalho torna-se mais prático pelo fato de não ser necessário programar os atributos dos componentes. Isso porque através do mouse, o programador decide onde posicionar cada componente do seu *form* e o tamanho de cada componente. Entretanto, para utilizarmos um componente assim que clicamos nele, precisamos criar um evento. Veja um exemplo de um evento de um botão sendo criado em C#:

```
private void button1_Click(object sender, EventArgs e)   {          COMANDOS          }
```

Dentro das chaves, inserimos os comandos a serem executados ao clicar no botão. Em Ruby, isso se dá a partir da seguinte forma (exemplo: botão utilizado para armazenar o cálculo e exibi-lo:

```
botaoCalcular.connect(SEL_COMMAND) do
    alturabtn = altura.to_s
    basebtn = base.to_s
    @resultado.text = ((alturabtn.to_f * basebtn.to_f)/2).to_s
  end
```

Nessa etapa, as variáveis “alturabtn” e “basebtn” recebem os dados informados pelo usuário e os convertem para o tipo string, através do comando “.to_s”. Por fim, temos o cálculo e exibição da área do triângulo, como vemos em “@resultado.text = ((alturabtn.to_f * basebtn.to_f)/2).to_s”.

Em C#, para programarmos um botão que realize as mesmas operações, faríamos da seguinte forma:

```
private void button1_Click(object sender, EventArgs e)
    {
        float alturatriangulo = float.Parse(textBox1.Text);
        float basetriangulo = float.Parse(textBox2.Text);
        float areatriangulo = (alturatriangulo * basetriangulo ) / 2;
        MessageBox.Show("Area do Triângulo = " + areatriangulo);
    }
```

Nesse ponto, temos as definições das variáveis “alturatriangulo”, “basetriangulo” e “areatriangulo”, todas do tipo *float*. As duas primeiras recebem os dados já convertidos através do comando *float.Parse*(). Já a última é a variável que recebe o valor da área do triângulo. Por fim, usamos o comando *MessageBox.Show*() para exibir uma mensagem na tela assim que o usuário clicar no botão.

Em Ruby, a limpeza dos campos é dada da seguinte forma:

```
botaoLimpar.connect(SEL_COMMAND) do
    altura.text = ""
    base.text = ""
    @resultado.text = ""
  end
```

Atribuímos aos campos altura, base e resultado *strings* vazias, para que estes não exibam valores assim que o usuário clicar no botão limpar. Em C#, é feito assim:

```
private void button2_Click(object sender, EventArgs e)
    {
        textBox1.Clear();
        textBox2.Clear();
        textBox1.Focus();
    }
```

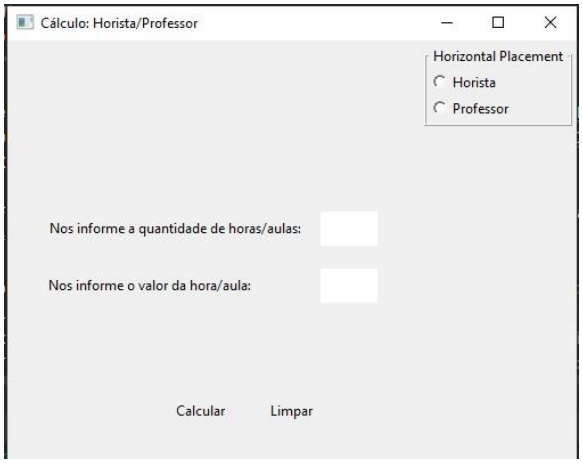
Usando o comando .*Clear()*, os campos ficam limpos automaticamente. Já o comando .*Focus*(), para selecionar o primeiro campo novamente.

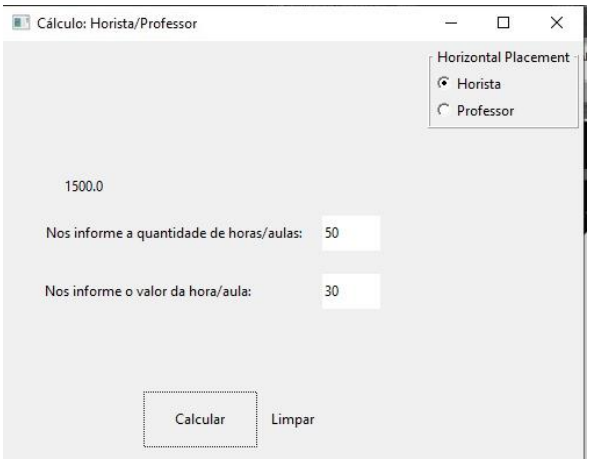
7.3. PROGRAMA 2: HORISTA/PROFESSOR (RADIO BUTTON)

```
require 'fox16'
include Fox

class GUI < FXMainWindow
  def initialize(app)
    super(app,"Cálculo:
Horista/Professor",:width=>500,:height=>400)

    controls = FXVerticalFrame.new(self,
LAYOUT_SIDE_RIGHT|LAYOUT_FILL_Y|PACK_UNIFORM_WIDTH)
    group1 = FXGroupBox.new(controls, "Horizontal
Placement", GROUPBOX_TITLE_CENTER|FRAME_RIDGE)
    @group1_dt = FXDataTarget.new(2)

    chk_horista = FXRadioButton.new(group1, "Horista",
@group1_dt, FXDataTarget::ID_OPTION)
    chk_Professor = FXRadioButton.new(group1, "Professor",
@group1_dt, FXDataTarget::ID_OPTION+1)

    lbl_qtd = FXLabel.new(self,"Nos informe a quantidade de
horas/aulas:", :opts=>LAYOUT_EXPLICIT, :width=>250,
:height=>50, :x=>23, :y=>140)
```

```
    lbl_valor = FXLabel.new(self,"Nos informe o valor da
hora/aula:", :opts=>LAYOUT_EXPLICIT, :width=>200,
:height=>50, :x=>25, :y=>190)
    txt_qtd = FXTextField.new(self, 30,
:opts=>LAYOUT_EXPLICIT, :width=>50, :height=>30, :x=>275,
:y=>150)
    txt_valor = FXTextField.new(self, 30,
:opts=>LAYOUT_EXPLICIT, :width=>50, :height=>30, :x=>275,
:y=>200)

    @resultado = FXLabel.new(self,"",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>100)

    botaoCalcular = FXButton.new(self, "Calcular",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>120,
:y=>300)
    botaoLimpar = FXButton.new(self, "Limpar",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>200,
:y=>300)

    botaoCalcular.connect(SEL_COMMAND) do

      qtd = txt_qtd.to_s
      valor = txt_valor.to_s
      result = 0

      if (chk_horista.checked?) == true
        result = valor.to_f * qtd.to_f
      else
        result = (valor.to_f * qtd.to_f) * 1.25
      end

      @resultado.text = result.to_s

    end

    botaoLimpar.connect(SEL_COMMAND) do
      txt_qtd.text=""
      txt_valor.text=""
      @resultado.text = ""
    end
  end


  def create
    super
    show(PLACEMENT_SCREEN)
  end
end
```

```
app = FXApp.new
GUI.new(app)
app.create
app.run
```





- **Programa 2, legenda**

Não diferente dos outros programas, a linha "def initialize(app)" é o que permite o aplicativo ser executado.

Enquanto o comando "super", logo abaixo, cria limites para a janela que será aberta no sistema operacional, colocando limite no nome do executável, e no tamanho da janela que será apresentada, desta vez a largura é 500 e a altura é 400.

No código principal, são instancializados os botões, caixas de texto etc, para posteriormente receberem suas devidas funções e variavéis. Os botões, labels e caixas de texto recebem seus tamanhos e são alocados em posições por "x" e "y", para serem alinhados e não sobreporem nenhum objeto.

O radio button é declarado para criar apenas duas opções, entre horista e professor, onde apenas uma delas poderá ser permitida a marcação.

Caso o radio button não estivesse dentro do group 1, mais de uma opção de radio button poderia ser marcada.

Inicialmente as variaveis "qtd" e "valor" são convertidas para string.

Após isso entramos no if presente caso o botão calcular seja pressionado com os valores em suas respectivas caixas de texto. os resultados são convertidos para float, e após isso é nos apresentado o texto contendo o resultado.

A função do botão limpar continua a mesma, que é de remover os caracteres presentes nos campos de escrita.

- **Programa 2, comparação com C#**

Em Ruby, quando comparado com o C# na questão de praticidade na inclusão de itens que serão utilizados na interface, acaba tornando-se menos do que a outra linguagem, pois tudo é preciso declarar manualmente, como foi abordado anteriormente. Sendo assim, é necessário possuir conhecimento de cada um para que o form esteja funcionando, como vê-se nesse exemplo, onde há o comando *FXVerticalFrame*, responsável por posicionar automaticamente janelas-filhas verticalmente:

```
Controls = FXVerticalFrame.new(self,
LAYOUT_SIDE_RIGHT|LAYOUT_FILL_Y|PACK_UNIFORM_WIDTH)
```

Para limitar a quantidade de itens que aparecem no radiobutton, usa-se *FXDataTarget* seguido do ".new(2)", ou seja, só aparecerá 2 itens.

```
@group1_dt = FXDataTarget.new(2)
```

Aqui pode-se observar algumas diferenças na declaração da variável, pois dentro dos parênteses já inicia incluindo que tal variável participa de um grupo e diferente do programa 1, o *radiobutton* está sendo nomeado como "Professor". O *ID_OPTION* é responsável por identificar, quando é unido a +1 ou outro valor, serve para que as outras variáveis não tenham o mesmo identificador.

```
Chk_Professor = FXRadioButton.new(group1, "Professor", @group1_dt,
FXDataTarget::ID_OPTION+1)
```

O cálculo em Ruby é semelhante ao C#, já que os dois iniciam passando os valores digitados pelo usuário as variáveis. As únicas diferenças é que não precisa colocar o comando dos *ifs* entre chaves e precisa colocar "== *true*" fora dos parênteses para que o if só execute caso seja verdadeiro e, também, está exibindo por uma variável e não por *MessageBox*.

Ruby:

```
botaoCalcular.connect(SEL_COMMAND) do
    qtd = txt_qtd.to_s
    valor = txt_valor.to_s
    result = 0
    if (chk_horista.checked?) == true
        result = valor.to_f * qtd.to_f
    else
        result = (valor.to_f * qtd.to_f) * 1.25
    end
    @resultado.text = result.to_s
  End
```

C#:

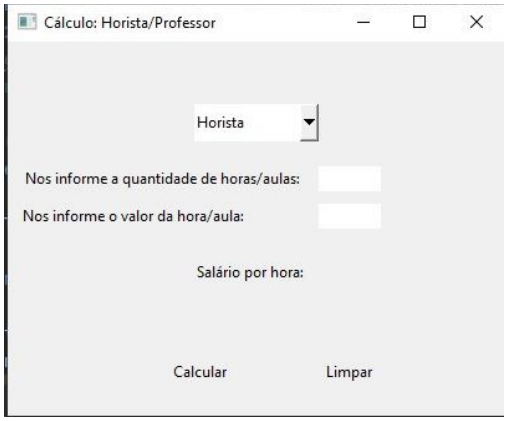
```
    Private void button1_Click(object sender, EventArgs e)
    {
        Float qtd = float.Parse(textBox1.Text);
        Float valor = float.Parse(textBox2.Text);
        Float sb;
        If (radioButton1.Checked)
        {
            Sb = qtd * valor;
        }
        Else
        {
            Sb = (qtd * valor) * 1.25f;
        }
```

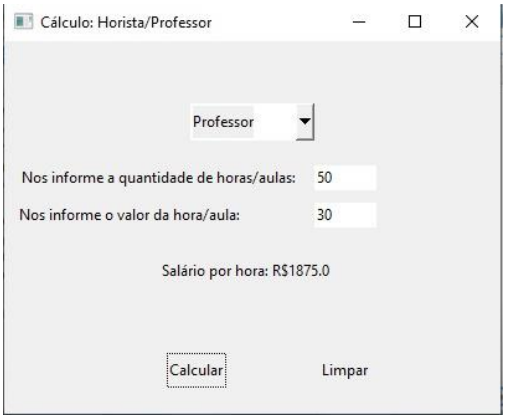
```
        MessageBox.Show("Salário Bruto = " + sb);
    }
```

7.4. PROGRAMA 3: HORISTA/PROFESSOR (COMBO BOX)

```
require 'fox16'
include Fox

class GUI < FXMainWindow
  def initialize(app)
    super(app,"Cálculo:
Horista/Professor",:width=>400,:height=>300)

    cmb_box = FXComboBox.new(self , 20, :opts =>
LAYOUT_EXPLICIT, :width=>100, :height=>30, :x=>150, :y=>50)
    cmb_box.numVisible = 2
    cmb_box.appendItem("Horista")
    cmb_box.appendItem("Professor")

    lbl_qtd = FXLabel.new(self,"Nos informe a quantidade de
horas/aulas:", :opts =>   LAYOUT_EXPLICIT, :width=>230,
:height=>20, :x=>10, :y=>100)
    lbl_valor = FXLabel.new(self,"Nos informe o valor da
hora/aula:", :opts =>  LAYOUT_EXPLICIT, :width=>183,
:height=>20, :x=>10, :y=>130)
    txt_qtd = FXTextField.new(self, 30, :opts =>
LAYOUT_EXPLICIT, :width=>50, :height=>20, :x=>250, :y=>100)
    txt_valor = FXTextField.new(self, 30, :opts =>
LAYOUT_EXPLICIT, :width=>50, :height=>20, :x=>250, :y=>130)

    @resultado = FXLabel.new(self,"Salário por hora:", :opts
=>   LAYOUT_EXPLICIT, :width=>150, :height=>50, :x=>120,
:y=>160)

    botaoCalcular = FXButton.new(self, "Calcular", :opts =>
LAYOUT_EXPLICIT,:width=>50, :height=>30, :x=>130, :y=>250)
    botaoLimpar = FXButton.new(self, "Limpar", :opts =>
LAYOUT_EXPLICIT,:width=>50, :height=>30, :x=>250, :y=>250)

    botaoCalcular.connect(SEL_COMMAND) do

      qtd = txt_qtd.to_s
      valor = txt_valor.to_s
      result = 0
```

```
      if cmb_box.to_s == "Horista"
        result = valor.to_f * qtd.to_f
      end
      if cmb_box.to_s == "Professor"
        result = (valor.to_f * qtd.to_f) * 1.25
      end

      @resultado.text = "Salário por hora: R$" + result.to_s

    end

    botaoLimpar.connect(SEL_COMMAND) do
      txt_qtd.text=""
      txt_valor.text=""
      @resultado.text = ""
    end
  end


  def create
    super
    show(PLACEMENT_SCREEN)
  end
end

app = FXApp.new
GUI.new(app)
app.create
app.run
```





- **Programa 3, legenda**

Não diferente dos outros programas, a linha "def initialize(app)" é o que permite o aplicativo ser executado. enquanto o comando "super", logo abaixo, cria limites para a janela que será aberta no sistema operacional, colocando limite no nome do executável, e no tamanho da janela que será apresentada, desta vez a largura é 400 e a altura é 300.

Entre esses comandos citados anteriormente, existe o código principal, para o aplicativo não ser apenas uma janela vazia.

Novamente são inicializados os botões, caixas de texto etc para posteriormente receberem suas devidas funções e variáveis. Os botões, labels e caixas de texto recebem seus tamanhos e são alocados em posições por "x" e "y", para serem alinhados e não sobreporem nenhum objeto.

Dessa vez nós temos a presença de combo box, que é alocada em separado das labels, a combo box precisa receber o número de itens que vai existir, para depois ser criados os itens.

Inicialmente as variáveis "qtd" e "valor" são convertidas para string.

Após isso entramos no if presente caso o botão calcular seja pressionado com os valores em suas respectivas caixas de texto. os resultados são convertidos para float, e após isso é nos apresentado o texto contendo o resultado.

A função do botão limpar continua a mesma, que é a de remover os caracteres presentes nos campos de escrita.

- **Programa 3, comparação com C#**

Em relação ao c#, que é trabalhado em windows form, nós temos a criação de botões de forma automática (clicar e arrastar), enquanto em ruby toda a criação foi manual, desde a nomenclatura, até o deslocamento de posição dos mesmos.

Enquanto em C#, obtém-se uma interface ao clicar no botão dentro do *form*, podendo localizar a parte onde escrevemos o código. Em Ruby, não tem essa facilidade de localização, o que pode dificultar caso o programa seja mais extenso.

A semelhança mais notória entre C# e Ruby ocorre durante o evento de clique:

C#:

```
private void button2_Click(object sender, EventArgs e)
```

Ruby (sua função é realizar a ação programada logo após clicar no botão):

```
botaoCalcular.connect(SEL_COMMAND) do
```

Para apresentar o resultado das operações, tanto em C# quanto em Ruby, nós temos a mesma essência, porém com semântica diferente:

```
MessageBox.Show("Salário Bruto = " + sb);
```

Em C#, enquanto em ruby temos a linha:

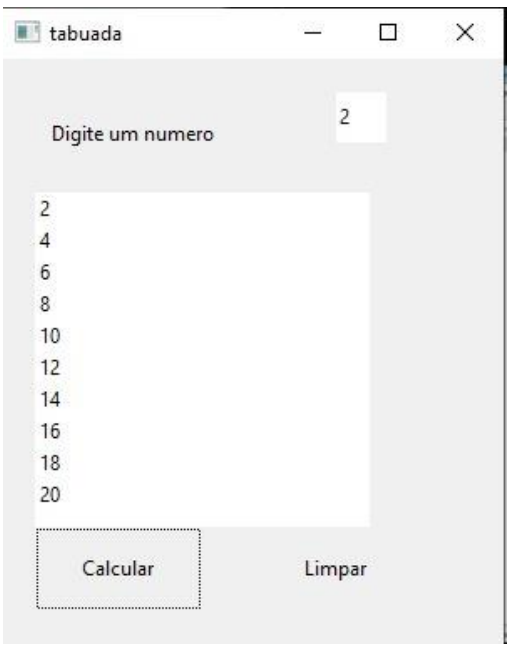
```
@resultado.text = "Salário Bruto: R$" + result.to_s
```

Outra diferença está no botão limpar, onde em C# há o comando .*Clear*, que serve para limpar os campos selecionados, enquanto em Ruby, essa opção não existe, sendo necessário retornar o campo para vazio após o evento do botão "Limpar".

7.5. PROGRAMA 4: TABUADA

```
require 'fox16'
include Fox

class GUI < FXMainWindow
  def initialize(app)
    super(app,"tabuada",:width=>300,:height=>350)

    label1 = FXLabel.new(self,"Digite um numero ",
:opts=>LAYOUT_EXPLICIT, :width=>150, :height=>50, :x=>5,
:y=>20)
    numero = FXTextField.new(self, 30,
:opts=>LAYOUT_EXPLICIT, :width=>30, :height=>30, :x=>200,
:y=>20)

    @listbox = FXList.new(self,  :opts=>LAYOUT_EXPLICIT,
:width=>200, :height=>200, :x=>20, :y=>80)
    botaoCalcular = FXButton.new(self, "Calcular",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>20,
:y=>280)
    botaoLimpar = FXButton.new(self, "Limpar",
:opts=>LAYOUT_EXPLICIT, :width=>100, :height=>50, :x=>150,
:y=>280)
```

```
    botaoCalcular.connect(SEL_COMMAND) do

      conta = numero.to_s

      for a in 1..10 do
        @listbox.appendItem((conta.to_i * a).to_s)
      end
    end
    botaoLimpar.connect(SEL_COMMAND) do
      @listbox.clearItems()
      numero.text=""
    end
  end


  def create
    super
    show(PLACEMENT_SCREEN)
  end
end

app = FXApp.new
GUI.new(app)
app.create
app.run
```



- **Programa 4, legenda**

Com a explicação já dita dos programas anteriores, resumidamente, podemos dizer que na classe "GUI" foi criado o "label1", "numero", "@listbox", entre outros (juntamente com o tamanho, o local, e o formato). E com isso, temos também o botão calcular, onde irá transformar o "numero" em "string", e após isso, criar um for, onde é declarado que o "a" ficar multiplicando até chegar a 10, ou seja, irá calcular: a * conta.to_i (o valor declarado pelo usuário), resumidamente, é uma tabuada.

O botão limpar, serve para esvaziar os campos.

E os comandos feitos logo em seguida, já foram explicados logo nos exercícios acima. Mas resumidamente, podemos dizer que, o "def create" acaba gerando o form, centralizado na tela. E logo abaixo, podemos dizer que é a parte onde iremos inicializar o programa ao todo.

- **Programa 4, comparação com C#**

Comparando o botão de cálculo em C# e em Ruby, é perceptível que em ambos é necessária a conversão para *string*, sendo em C# escrita por:

```
int tabuada = int.Parse(textBox1.Text)
```

E em Ruby:

```
conta = numero.to_s
```

Ainda no botão do cálculo, em ambas linguagens o *for* possui uma estrutura similar, sendo visível apenas uma pequena diferença na condição representada em C# por:

```
for (int i = 1; i < 11; ++i)
```

Já em Ruby:

```
for a int 1..10 do
```

O cálculo se dá a partir do *for* para apresentar os números da tabuada do valor digitado pelo usuário, sendo assim, pode-se analisar a diferença na inclusão do *listbox* das duas linguagens, pois a C# precisa botar a lista entre chaves e colocar o número que a variável irá iniciar, depois declarar que é menor do que o valor 11 e finalizando com um código responsável por sempre acrescentar mais 1 e, assim, completar o laço, como é perceptível a seguir:

```
Private void button1_Click(object sender, EventArgs e)
    {
        Int tabuada = Int.Parse(textBox1.Text);
        For (int i = 1; i < 11; ++i)
        {
            listBox1.Items.Add(tabuada + " x " + i + " = " + tabuada * i);
        }
    }
```

Em Ruby, é preciso informar o primeiro até o último número que será multiplicado pelo valor do usuário e já exibi-lo, podendo observar que não é preciso o uso de chaves para que o cálculo seja executado:

```
botaoCalcular.connect(SEL_COMMAND) do
    conta = numero.to_s
    for a in 1..10 dd
```

```
      @listbox.appendItem((conta.to_i * a).to_s)
    End
  End
```

Com o programa 4, o modo de limpar os campos modifica e em Ruby é preciso usar um comando "*.clearItems*" para deixar os itens vazios.

```
botaoLimpar.connect(SEL_COMMAND) do
    @listbox.clearItems()
    Numero.text=""
  End
End
```

O mesmo procedimento ocorre em C#, porém com códigos em ordens diferentes "*.Items.Clear*()", como a seguir:

```
Private void button2_Click(object sender, EventArgs e)
    {
        textBox1.Clear();
        listBox1.Items.Clear();
        textBox1.Focus();
    }
```

8. CAPÍTULO 4: EXERCÍCIOS EM POO (PROGRAMAÇÃO ORIENTADA A OBJETOS

Programação orientada a objetos é um paradigma de programação baseado no conceito de "objetos", que podem conter dados na forma de campos, também conhecidos como atributos, e códigos, na forma de procedimentos, também conhecidos como métodos.

Nesse capítulo iremos ver algumas aplicações práticas da programação orientada a objetos.

8.2. PROGRAMA 1: HORISTA

Nessa classe, temos a captura e alocação dos dados que serão usados, através das funções “get” e “set”:

```
class Horista


  def setQtd(_b)
    @qtd = _b
  end


  def getQtd()
    return @qtd
  end
```

Em relação ao C#, “get” e “set” são palavras reservadas usadas para capturar e alocar os dados dos campos:

```
        private String qtd;
        private String valor;


        public void setQtd(string _qtd) { qtd = _qtd; }
        public String getQtd() { return qtd; }
```

Em C#, devemos definir os níveis de acesso dos métodos. Nesse caso, temos um método público e um vazio. Sabemos disso através da palavra reservada “public” e “void” (vazio, ou seja, não retorna um tipo específico).

BLL

Na classe bll, foram utilizados testes condicionais para realizar as validações dos dados nos campos do formulário:

```
if horista.getQtd().nil? == true
```

```
      return "O campo QUANTIDADE DE HORAS é de preenchimento obigatório"
```

O mesmo caso foi repetido para verificar outras condições, como para saber se o valor inserido era de fato um número, se era menor do que zero etc.

Em C# teríamos algo muito parecido:

```
Erro.setErro(false);

      if (_umhorista.getQtd().Length == 0)

      {

          Erro.setErro("O campo QUANTIDADE DE HORAS é de preenchimento obrigat
ório...");

          return;

      }
```

Notamos algumas pequenas diferenças no código em C#. Usamos o comando "Erro.setErro" e "return", para retornar uma mensagem de erro caso fosse necessário.

```
      else

      {

          try

          {

              float.Parse(_umhorista.getQtd());

          }

          catch

          {

              Erro.setErro("O campo QUANTIDADE DE HORAS deve ser numérico...")
;

              return;

          }
```

Mais além no código, usamos o comando "try...catch" para capturar um possível erro e, mediante a ocorrência dele, os comandos Erro.setErro e return foram usados para exibir

uma mensagem de erro na tela. Essa mesma sequência foi usada para verificar os outros campos.

HORISTA IHM

Finalizamos com a classe Horista IHM, onde instanciamos as classes e verificamos a base de cálculo. É feita a verificação para a usabilidade dos números informados e depois os valores são passados para a classe “bll” para realização dos cálculos:

```
horista = Horista.new()
bll=HoristaBLL.new()


basecal = qtdHoras.text
alturacal = vlrHoras.text


if ((basecal =~ /[-0-9]/).to_s) == "0"
  basecal = basecal.to_f
else
  basecal = basecal.to_s
end


if ((alturacal =~ /[-0-9]/).to_s) == "0"
  alturacal = alturacal.to_f
else
  alturacal = alturacal.to_s
end


if basecal.to_s == ""
  basecal = nil
end
if alturacal.to_s == ""
  alturacal = nil
end
```

```
horista.setQtd(basecal)
horista.setValor(alturacal)

bll.validacao(horista)
```

Finalizamos limpando os campos do formulário:

```
qtdHoras.text= ""
vlrHoras.text= ""
resultado.text = ""
```

Em C#, o código poderia ser escrito de uma maneira mais simples, onde podemos usar as classes com mais eficácia para produzir os resultados desejados:

```
            Horista umhorista = new Horista();

            umhorista.setQtd(tbqtdhoras.Text);
            umhorista.setValor(tbvalorhora.Text);
            HoristaBLL.validaDados(umhorista);
            if (Erro.getErro())
            {
                MessageBox.Show(Erro.getMens());
            }
            else
            {
                tbsalariobruto.Text = umhorista.getSalarioBruto();
                tbqtdhoras.Enabled = false;
                tbvalorhora.Enabled = false;
            }
```

Aplicamos os métodos relativamente aos campos onde são inseridos os dados, e finalizamos com a exibição do salário bruto. Por fim, desabilitamos os campos para não ser mais possível que o usuário os use, exceto caso ele escolha limpá-los antes.

8.3. PROGRAMA 2: ÁREA TRIÂNGULO

Nessa classe, temos a captura e alocação dos dados que serão usados, através das funções “get” e “set”. Assim podemos receber e utilizar os dados necessários para calcular a área do triângulo, conforme podemos ver abaixo:

```
def setB(_b)
        @b = _b
      end


      def getB()
        return @b
      end
```

Esses mesmos procedimentos foram feitos para adquirir e alocar os valores da altura do triângulo e para a realização do cálculo.

Em C#, poderíamos fazer da seguinte forma:

```
      private String b;
      private String h;


      public void setB(string _b) { b = _b; }
      public String getB() { return b; }
```

Criamos os campos “b” e ”h”, cada qual sendo do tipo String e, posteriormente, criamos os métodos “set” e ”get” para alocação e captura dos dados, respectivamente. Vale notar que sempre que em C#, sempre que criamos um método, precisamos definir o seu nível de acesso, por isso a palavra reservada “Public” no começo de cada método.

Após isso, efetuamos os mesmos procedimentos para a altura do triângulo e para o cálculo da área dele.

BLL

Na classe bll, foram utilizados testes condicionais para realizar as validações dos dados nos campos do formulário:

```
      if cal.getB().nil? == true
        return "O campo BASE é de preenchimento obrigatório..."
      else
        if (cal.getB().class.to_s == "String" )
          return "O campo BASE deve ser numérico..."
        end
```

Nesse caso, verificamos algumas condições relativas ao preenchimento correto dos campos do formulário.

Em C#:

```
public static void validaDados(Triangulo umtriangulo)
        {
            Erro.setErro(false);
            if (umtriangulo.getB().Length == 0)
            {
                Erro.setErro("O campo BASE é de preenchimento obrigatório...");
                return;
            }
            else
            {
                try
                {
                    float.Parse(umtriangulo.getB());
                }
```

```
            catch
            {
                Erro.setErro("O campo BASE deve ser numérico...");
                return;
            }
```

Usamos o comando try...catch para capturar um possível erro e informar uma mensagem de erro para o usuário.

TRIÂNGULO IHM

Finalizamos com a classe Triângulo IHM, onde instanciamos as classes e verificamos a base de cálculo. É feita a verificação para a usabilidade dos números informados e depois os valores são passados para a classe “bll” para realização dos cálculos:

```
    cal = Cálculo.new
    bll = BLL.new

    basecal = base.text
    alturacal = altura.text

    if ((basecal =~ /[-0-9]/).to_s) == "0"
      basecal = basecal.to_f
    else
      basecal = basecal.to_s
    end

    if ((alturacal =~ /[-0-9]/).to_s) == "0"
      alturacal = alturacal.to_f
    else
      alturacal = alturacal.to_s
```

```
    end
```

Semelhantemente, em C# poderíamos fazer da seguinte maneira:

```
private void button1_Click(object sender, EventArgs e)
    {
        Triangulo meutriangulo = new Triangulo();


        meutriangulo.setB(tbBase.Text);
        meutriangulo.setH(tbAltura.Text);
        TrianguloBLL.validaDados(meutriangulo);
        if (Erro.getErro())
        {
            MessageBox.Show(Erro.getMens());
        }
        else
        {
            tbArea.Text = meutriangulo.getArea();
            tbBase.Enabled = false;
            tbAltura.Enabled = false;
        }
    }
```

Instanciamos a classe “meutriangulo” e, a partir daí, realizamos as operações por meio da utilização dos métodos.

A programação do botão limpar é ainda mais fácil, bastando apenas conhecer um pouco de cada linguagem para conseguir realizar a limpeza dos campos.

Em Ruby:

```
botaoLimpar.connect(SEL_COMMAND) do
    altura.text= ""
```

```
      base.text= ""
      @resultado.text = ""


    End
```

Em C#:

```
private void button2_Click(object sender, EventArgs e)
  {
      tbBase.Clear();
      tbAltura.Clear();
      tbArea.Clear();
      tbBase.Enabled = true;
      tbAltura.Enabled = true;
      tbBase.Focus();
  }
```

Notamos que, na primeira linguagem é necessário atribuir vazio (= “”) aos campos para que eles possam ser limpos. Já em C#, contamos como método “Clear”, que realiza a limpeza automaticamente.

8.4. PROGRAMA 3: EQUAÇÃO DO SEGUNDO GRAU

Para que o programa seja iniciado em Ruby, é necessário, já que todas as classes FXRuby são definidas no módulo *Fox*, referi-las. E também, devemos adicionar uma instrução *include Fox* para que todos os nomes no módulo *Fox* sejam "incluídos" no *namespace* global. Exemplo disto, observa-se na classe “EquacaoForm”:

```
require 'fox16'

require './cálculo'
```

```
require './bll'

include Fox
```

Para definir um novo método e iniciar as funções do programa, é preciso, na mesma classe que está como “ponto de entrada principal para o aplicativo”, utilizarmos:

```
  def initialize(app)
```

Já em C#, utiliza-se de duas classes, tanto da entrada principal (“Program”):

```
 static class Program

    {

        /// <summary>

        /// Ponto de entrada principal para o aplicativo.

        /// </summary>

        [STAThread]

        static void Main()

        {

            Application.EnableVisualStyles();

            Application.SetCompatibleTextRenderingDefault(false);

            Application.Run(new Equacao2GIHM());

        }
```

Quanto a “Equacao2GIHM”, que armazena os dados pedidos ao usuário e apresentar o resultado logo em seguida, ou seja, a classe Pai:

```
public partial class Equacao2GIHM : Form

    {

        public Equacao2GIHM()
```

```
        {
            InitializeComponent();
        }
```

Em C#, para iniciar o cálculo, é preciso criar um novo form desta maneira abaixo, referindo-se a classe em que estão as funções matemáticas:

```
Equacao2G umaequacao = new Equacao2G();
```

Já em Ruby, no botão calcular já está "chamando" os dois outros forms, tanto do cálculo, quanto da validação de dados:

```
botaoCalcular.connect(SEL_COMMAND) do
      cal = Cálculo.new
      bll = BLL.new
```

A classe "Cálculo" que está na linguagem de programação Ruby, é composta inicialmente com *setters*, os quais servem para atribuir valores aos atributos do objeto que são guardadas em variáveis de instâncias que sempre são começadas com @. E também, há a presença logo em seguida de *getters*, que leem os valores dessas variáveis e permitem acessá-los sempre que necessário retornando-os nas variáveis de instância, como é perceptível abaixo:

```
class Cálculo


    def setA(_a)
      @a = _a
    end


    def getA()
      return @a
    end
```

```
    [...]
```

No código abaixo, está sendo apresentado o cálculo que retornará o resultado de “Delta” e dos “x1” e “x2”. Como pode observar, há no final de cada parênteses “.to_s”, que está convertendo o valor final por uma *string*, a qual será exibida na tela. Já no “getX1”, o mesmo vale para “getX2”, no final da variável de instância “@delta”, está “.to_f”, o qual retorna o resultado da interpretação dos caracteres *iniciais* em *str*, ou seja, caracteres estranhos após o final de um número válido são ignorados:

```
def getDelta()

      return ((@b * @b) - (4 * @a * @c)).to_s

    end



    def getX1()

      @delta = getDelta().to_f;

      return ((-@b + Math.sqrt(@delta)) / (2 * @a)).to_s

    end

    [...]
```

Em C#, está ocorrendo o mesmo no bloco de código, porém, a estrutura está divergente, pois as variáveis antes de passarem nos comandos *get* e *set*, são privados e, posteriormente, é preciso declarar novamente os tipos das variáveis e deixá-las públicas. E igualmente em Ruby, no final dos cálculos está com o comando que retorna em *string*, porém, está “.*ToString*”, e na variável “auxa”, inicia com *float* e termina com “*float.Parse*(a)”.

```
class Equacao2G

    {

        private String a;

        [...]
```

```
        public void setA(string _a) { a = _a; }

        public String getA() { return a; }

        [...]



        public String getDelta()

        {

            float auxa = float.Parse(a);

            [...]

            return ((auxb * auxb) - (4 * auxa * auxc)).ToString();

        }

        public String getX1()

        {

            float auxa = float.Parse(a);

            [...]

            return ((-auxb + Math.Sqrt(auxdelta)) / (2 *
auxa)).ToString();

        }
```

Em C#, depois de “chamar” o formulário para o cálculo, é utilizado o comando *setter*, pois serve para manipular as informações do objeto, ou seja, os atributos, acarretando na atribuição dos valores em variáveis do tipo texto.

```
            umaequacao.setA(tbA.Text);

            [...]

            Equacao2GBLL.validaDados(umaequacao);
```

Já em Ruby, além de criar variáveis de instância para adquirir valores no mesmo formato de texto, é necessário que seja declarado, utilizando *if* e *else*, o tipo do objeto, por exemplo, caso seja igual a 0, seu tipo será *float*, do contrário, será *string*. E caso seja *string*, se o valor for “”, ou seja, vazio, será atribuído “nil”, que significa nada.

```
      acal = a.text

      [...]

_

      if ((acal =~ /[-0-9]/).to_s) == "0"

        acal = acal.to_f

      else

        acal = acal.to_s

      end

_

      if acal.to_s == ""

        acal = nil

      end

_

      cal.setA(acal)

      [...]

      bll.validaDados(cal)
```

Posteriormente, em C#, com o comando *if* e *else*, irá verificar a validação dos dados apresentados pelo usuário. Caso houver algum dado não correspondente, aparecerá uma mensagem de erro, do contrário, será mostrado na tela os resultados a partir dos códigos “umaequacao.getX1” e “umaequacao.getX2”, e os campos para inserir os 3 valores serão inabilitados, pois sua habilitação está igual a *false*:

```
if (Erro.getErro())

            {

                MessageBox.Show(Erro.getMens());
```

```
            }
            else
            {
                tbX1.Text = umaequacao.getX1();
                tbX2.Text = umaequacao.getX2();
                tbA.Enabled = false;
                tbB.Enabled = false;
                tbC.Enabled = false;
            }
```

Diferente em C#, é necessário que haja duas classes, a qual irá manipular os valores com os comandos *set* e get, retornando-os posteriormente para as outras classes que irão "chamá-lo", e colocando-os em tipo bool, como é observável nos comandos abaixo na classe "Erro":

```
class Erro
    {
        private static bool erro;
        private static String mens;

        public static void setErro(bool _erro) { erro = _erro; }
        public static void setErro(String _mens) { erro = true; mens =
_mens; }
        public static bool getErro() { return erro; }
        public static String getMens() { return mens; }
    }
```

E a que irá validar esses valores que passaram pela classe "Erro", a qual está nomeada como "Equacao2GBLL". Para que apareça a mensagem de erro, o comando "*Erro.setErro*" precisa ser dado como *false* ao passar no *if* da classe "Equacao2GIHM". Com isso, caso os

valores dos campos sejam iguais a 0, irá aparecer a mensagem de preenchimento obrigatório; e utilizando *try* e *catch*, para caso seja digitado valores não numéricos. E também, se o valor de delta digitado for menor que 0, aparecerá que não há delta negativo.

```
class Equacao2GBLL

    {

        public static void validaDados(Equacao2G _umaequacao)

        {

            Erro.setErro(false);

            if (_umaequacao.getA().Length == 0)

            {

                Erro.setErro("O valor de A é de preenchimento
obrigatório...");

                return;

            }

            else

            {

                try

                {

                    float.Parse(_umaequacao.getA());

                }

                catch

                {

                    Erro.setErro("O valor de A deve ser numérico...");

                    return;

                }

            }

            [...]
```

```
            if (float.Parse(_umaequacao.getDelta()) < 0)

            {  Erro.setErro("Delta negativo, não existem raízes reais
para esta equação...");

                return; }
```

Em Ruby, diferente em C#, a classe “BLL” a qual fará a validação, está sendo “chamada” para que apareça a mensagem de erro referenciando a variável que sofreu atribuição “cal”, seguida de “.nil?”, o que verifica se o objeto é vazio. Se não houver problemas de verificação, os resultados serão passados igualmente em C#, porém, não há comandos de inabilitação.

```
if (bll.validaDados(cal).nil? == false )

        FXMessageBox.error(app, MBOX_OK, 'Error',
bll.validaDados(cal).to_s)

      else

        result1.text = cal.getX1().to_s

        result2.text = cal.getX2().to_s

      end

    end
```

BLL

Na classe “BLL”, ocorre o mesmo que explicado em C#, mudando apenas os comandos dentro do *if*, por exemplo, “.nil? == true” diferente de 0, e sendo iniciada por “cal”, que recebe os valores das variáveis de todos os campos; e a ausência dos comandos *try* e *catch*. E também, ao invés de terminar os comandos entre chaves, é usado *end* para finalizar cada possibilidade.

```
class BLL



    def validaDados(cal = Cálculo.new)



      if cal.getA().nil? == true

        return "O campo A é de preenchimento obrigatório..."

      else
```

```
        if (cal.getA().class.to_s == "String" )
          return "O campo A deve ser numérico..."
        end
      end
      [...]
       if(cal.getDelta().to_f < 0)
          return "Delta negativo, não existem raízes reais para esta
equação..."
        end
```

No botão de limpar em C#, é utilizado “.Clear()” para limpar os campos e “.Focus()” para que o primeiro valor a ser digitado seja do campo que está recebendo a variável “tbA”:

```
 private void button2_Click(object sender, EventArgs e)
        {
            tbA.Clear();
            [...]
            tbX1.Clear();
            [...]
            tbA.Enabled = true;
            [...]
            tbA.Focus();
```

Em Ruby, é declarado as variáveis seguidas de ".text = "" " ou dentro das aspas, caso queira que inicie com frases, esteja, por exemplo, "Valor de x1".

```
botaoLimpar.connect(SEL_COMMAND) do
      a.text= ""
```

```
      [...]

      result1.text = "Valor de x1"

      [...]

    end

  end
```

Enquanto em C#, há um sistema automático no windows que possibilita o usuário fazer os ajustes e inclusões na tela. O código do design não é colocado manualmente, mas observa-se que desde o tipo de ferramenta, até a posição que está no form, encontram-se declaradas separadamente, já que cada um pertence a um tipo de função:

```
 private void InitializeComponent()

        {

            this.label1 = new System.Windows.Forms.Label();

            [...]

            this.label1.AutoSize = true;

            this.label1.Font = new System.Drawing.Font("Microsoft Sans
Serif", 18F, System.Drawing.FontStyle.Regular,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));

            this.label1.Location = new System.Drawing.Point(28, 20);

            this.label1.Name = "label1";

            this.label1.Size = new System.Drawing.Size(128, 29);

            this.label1.TabIndex = 0;

            this.label1.Text = "Valor de a:";
```

Já em Ruby, tudo é armazenado entre parênteses e organizados em uma sequência padronizada, pois cada lugar tem seu tipo de função, e diferente da linguagem C#, é preciso declará-las manualmente:

```
label1 = FXLabel.new(self,"Valor de a: ", :opts=>LAYOUT_EXPLICIT,
:width=>100, :height=>50, :x=>20, :y=>20)
```

8.5. PROGRAMA 3: CADASTRO LIVRO

Em C#, o formulário é “chamado” antes do programa ser inicializado, fazendo com que o programa não dê acesso ao banco de dados, pois está criando a partir da classe responsável pela manipulação dos dados, a “Livro”:

```
public partial class LivroIHM : Form

    {

        Livro umlivro = new Livro();

        public LivroIHM()

        {

            InitializeComponent();

        } [...]
```

O que ocorre igualmente em Ruby, porém, não apenas a classe “Livro”, mas também a de cálculo (“BLL”), como pode-se analisar:

```
def initialize(app)

[...]

 @livro = Livro.new

 @bll = BLL.new
```

A classe “Livro” em Ruby, primeiramente, atribui uma variável, por exemplo, “_codigo”, a uma instância que será utilizada, posteriormente, com o comando *get*, para retornar o valor.

```
class Livro
```

```
  def setCodigo(_codigo)
    @codigo = _codigo
  end
[...]
  def getCodigo()
    return @codigo
  end
[...]
```

Em C#, inicialmente, é preciso declarar uma variável, por exemplo a que está nomeada "codigo", sendo ela privada e com o tipo (como: *String)*. Está sendo usado antes dos comandos *set* e *get*, ao invés de *def*, a situação (public) e o tipo, por exemplo, *void* - vazio - e *String*. E utilizando para isolar os comandos as chaves, diferentemente, de Ruby, que usa *end* para finalizar.

```
class Livro
    {
        private String codigo;
        [...]

        public void setCodigo(String _codigo) { codigo = _codigo; }
        [...]

        public String getCodigo() { return codigo; }
        [...]
    }
```

No botão para salvar as informações dadas ao usuário, variáveis de instância são declaradas para adquirir valores no mesmo formato de texto, como “@codigo2”. É necessário que seja declarado, utilizando *if* e *else*, o tipo do objeto, por exemplo, caso seja igual a 0, seu tipo será float, do contrário, será *string*. E se for *string*, caso o valor seja igual a “”, ou seja, vazio, será atribuído “nil”, que significa nada. Depois de sair do “laço”, a informação do objeto será atribuída à instância, e com isso, levada a classe “Livro”. E, posteriormente, os atributos serão verificados a partir do comando “@bll.validadaDados(@Livro)”.

```
btn_salva.connect(SEL_COMMAND) do


      @codigo2 = codigo.text
      [...]


      if ((@ano2 =~ /[-0-9]/).to_s) == "0"
        @ano2 = @ano2.to_f
      else
        @ano2 = @ano2.to_s
      end


      if @codigo2.to_s == ""
        @codigo2 = nil
      end
     [...]
      @livro.setCodigo(@codigo2)
      [...]
      @bll.validaDados(@livro)
```

Em C#, não é feito o “laço” verificando as informações e passando os resultados delas para algum tipo de declaração para ser, então, conduzida a instância. Já está referenciando a

manipulação sofrida na classe “Livro” e utilizando “.Text” para deixar em formato de texto. E, posteriormente, os atributos serão verificados a partir do comando, terminado em ponto e vírugula, “LivroBLL.validaDados(umlivro);”.

```
private void button1_Click(object sender, EventArgs e)

        {

            umlivro.setCodigo(textBox1.Text);

            [...]



            LivroBLL.validaDados(umlivro);
```

Em Ruby, diferente em C#, a classe “BLL” a qual fará a validação, está sendo “chamada” para que apareça a mensagem de erro referenciando a instância “@livro”, seguida de “.nil?”, o que verifica se o objeto é vazio. Se não houver problemas de verificação, os resultados serão passados através do comando “FXMessageBox.information”, a mensagem informando que os dados foram inseridos.

```
if (@bll.validaDados(@livro).nil? == false )

        FXMessageBox.error(app, MBOX_OK, 'Error',
@bll.validaDados(@livro).to_s)

      else

        FXMessageBox.information(app, MBOX_OK,'Sucesso', "Dados
inseridos com sucesso!")

      end

    end
```

BLL

Na classe “BLL”, inicialmente, é declarado duas vezes, quando o usuário não digita o código, o “laço”, utilizando-se a instância que está recebendo os atributos do código pela classe “Livro”, seguida de “.getCodigo”, que está retornando a mensagem de preenchimento obrigatório. Embora seja declarado o mesmo para as outras variáveis, não é posto duas vezes.

Também é observável que utilizando “.class.to.s == “String””, aparecerá que o valor do ano deve ser numérico e caso o ano for dado como igual ou menor que 0, apresentará que o valor deve ser positivo.

```
class BLL


  def validaCodigo(livro = Livro.new)
    if livro.getCodigo().nil? == true
      return "O CÓDIGO é de preenchimento obrigatório!"
    end
  end


  def validaDados(livro = Livro.new)
    if livro.getCodigo().nil? == true
      return "O CÓDIGO é de preenchimento obrigatório!"
    end [...]


 if (livro.getAno().class.to_s == "String" )
      return "O valor do ANO deve ser numérico!"
    end


    if (livro.getAno() <= 0 )
      return "O valor do ANO deve ser positivo!"
    end
```

Em C#, com o comando *if* e *else*, irá verificar a validação dos dados apresentados pelo usuário. Caso houver algum dado não correspondente, aparecerá uma mensagem de erro, do contrário, será mostrado na tela a mensagem de dados inseridos.

```
if (Erro.getErro())
                MessageBox.Show(Erro.getMsg());
            else
                MessageBox.Show("Dados inseridos com sucesso!");
        }
```

Diferente em Ruby, na linguagem C#, é necessário que haja duas classes, a qual irá manipular os valores com os comandos *set* e get, retornando-os posteriormente para as outras classes que irão “chamá-lo”, e colocando-os em tipo bool, como é observável nos comandos abaixo na classe “Erro”:

```
class Erro
    {
        private static String msg;
        private static bool erro;

        public static void setMsg(String _msg)
        {
            erro = true;
            msg = _msg;
        }
        public static void setErro(bool _erro) { erro = _erro; }
        public static String getMsg() { return msg; }
        public static bool getErro() { return erro; }
    }
```

```
}
```

E a que irá validar esses valores que passaram pela classe “Erro”, a qual está nomeada como “Equacao2GBLL”. Para que apareça a mensagem de erro, o comando “*Erro.setErro*” precisa ser dado como *false* ao passar no *if* da classe “Equacao2GIHM”. Igualmente em Ruby, será declarado duas vezes o “laço” para o usuário digitar o código, caso este esteja vazio, ocorrendo o mesmo para as demais variáveis. Para o valor do ano, é utilizado *try*, para declarar o tipo (int) na variável, e *catch* - seguido de “(Exception), usado em casos isolados” -, para que o valor digitado não seja diferente de númerico.

```
class LivroBLL

    {



        public static void validaCodigo(Livro umlivro)

        {

            Erro.setErro(false);

            if (umlivro.getCodigo().Equals(""))

            {

                Erro.setMsg("O código é de preenchimento
obrigatório!");

                return;

            }

        }



        public static void validaDados(Livro umlivro)

        {

            Erro.setErro(false);

            if (umlivro.getCodigo().Equals(""))

            {
```

```
                Erro.setMsg("O código é de preenchimento
obrigatório!");
                return;
            }
            [...]
             try
            {
                int.Parse(umlivro.getAno());
            }
            catch (Exception)
            {
                Erro.setMsg("O valor do ano deve ser numérico!");
                return;
            }

            if (int.Parse(umlivro.getAno()) <= 0)
            {
                Erro.setMsg("O valor do Ano deve ser numérico e
positivo!");
                return;
            }[...]
```

Em C#, o botão de limpar ocorre a partir do comando “.Text = “”; “, seguindo assim para todos os outros campos.

```
private void button2_Click(object sender, EventArgs e)
        {
            textBox1.Text = "";
```

```
            [...]

        }
```

O mesmo ocorre em Ruby, porém, tudo em minúsculo, sem ponto e vírgula e terminado em *end*, e não, entre chaves.

```
btn_limpa.connect(SEL_COMMAND) do

      codigo.text= ""

      [...]

    end

  end
```

No botão de consulta em Ruby, é pedido ao usuário digitar o código que foi, anteriormente, armazenado, utilizando uma instância para o valor da variável "*.text*" seja atribuído. Depois será levado às classes "Livro" e "BLL" para verificar a existência de tal valor, e caso seja igual a vazio ou "nil", aparecerá a mensagem de obrigatoriedade até que seja digitado. Se for digitado corretamente, aparecerá os dados na tela.

```
btn_consulta.connect(SEL_COMMAND) do


      @codigo2 = codigo.text


      if @codigo2.to_s == ""

        @codigo2 = nil

      end


      @livro.setCodigo(@codigo2)

      @bll.validaDados(@livro)
```

```
      if (@bll.validaCodigo(@livro).nil? == false )

        FXMessageBox.error(app, MBOX_OK, 'Error',
@bll.validaCodigo(@livro).to_s)

      else

        codigo.text = @livro.getCodigo().to_s

        [...]

      end

    end
```

Em C#, não é declarado uma instância, pois já referência o campo em formato de texto. É pego o código digitado pelo usuário, levado as duas classes, igualmente em Ruby, e se não corresponder com os critérios, a mensagem persistirá, do contrário, os dados aparecerão na tela. Diferenciando com o uso do ponto e vírgula no final dos comandos, e o "laço" ocorrendo entre chaves.

```
private void button3_Click(object sender, EventArgs e)

        {

            umlivro.setCodigo(textBox1.Text);

            LivroBLL.validaCodigo(umlivro);



            if (Erro.getErro())

                MessageBox.Show(Erro.getMsg());

            else

            {

                textBox1.Text = umlivro.getCodigo();

                [...]

            }
```

9. CAPÍTULO 5: ACESSO AO BANCO DE DADOS EM RUBY

Bancos de dados ou bases de dados são conjuntos de arquivos relacionados entre si com registros sobre pessoas, lugares ou coisas. São coleções organizadas de dados que se relacionam de forma a criar algum sentido e dar mais eficiência durante uma pesquisa ou estudo científico.

Nesse capítulo vamos ver a aplicação do Banco de dados dentro da linguagem Ruby.

9.2. PROGRAMA LIVRO COM BANCO DE DADOS

```
require 'sqlite3'

class LivroDAL

  def conecta
    begin #begin e rescue equivalem ao try catch
      @db = SQLite3::Database.open "livraria.db" #abre um
banco, se não existir cria e abre ele
      @db.execute "CREATE TABLE IF NOT EXISTS Tablivros2(id
INT PRIMARY KEY,titulo varchar(30), autor varchar(30), editora
varchar(30), ano varchar(30))"
      #o comando execute manda as linhas de sql pro cmd,
atraves dele que temos os comandos sendo executados
    rescue SQLite3::Exception => e
      puts "Exception occurred"
      puts e #se der erro exibe qual foi
    ensure
      @db.close if @db #isso fecha o banco
    end
    end

  def inseriLivro(livro=Livro.new)
    begin
      @db = SQLite3::Database.open "livraria.db"
      id=livro.getCodigo().to_i
    titulo=livro.getTitulo().to_s
    autor=livro.getAutor().to_s
    editora=livro.getEditora().to_s
    ano=livro.getAno().to_s

    @db.execute "INSERT INTO TabLivros2
(id,titulo,autor,editora,ano) VALUES
(#{id},'#{titulo}','#{autor}','#{editora}','#{ano}')"
    rescue SQLite3::Exception => e
      puts "Exception occurred"
      puts e
    ensure
      @db.close if @db
    end
    end

  def consultaLivro(livro=Livro.new)
    begin
      @db = SQLite3::Database.open "livraria.db"
      id= livro.getCodigo().to_i
      stm = @db.prepare "SELECT * FROM TabLivros2 WHERE
id=#{id}"
```

```
      rs = stm.execute

      rs.each do |row|
        puts row.join "\s"

      rescue SQLite3::Exception => e
        puts "Exception occurred"
        puts e
  end
    end
  end
end
```

O método require permite carregar uma biblioteca e impede que ela seja carregada mais de uma vez. O método require retornará 'false' se você tentar carregar a mesma biblioteca após a primeira vez.

Começamos colocando o comando "require", onde o método permite carregar uma biblioteca e impede que ela seja carregada mais de uma vez. O método require retornará 'false' se você tentar carregar a mesma biblioteca após a primeira vez. Depois disso, criamos a classe "LivroDAL".

E com isso, usamos o comando "begin", que acabam sendo iguais a um "try catch". Logo em seguida, abrimos o banco de dados, e caso não exista, o programa vai forçar a criação de um e irá abrir. Fazemos uso do comando "@db.execute" que acaba mandando as linhas de sql para o cmd, ou seja, através dele que temos os comandos sendo executados.

A partir do momento em que entramos no comando “def inseriLivro(livro=Livro.new)”, nós temos a adição de mais um livro no banco de dados, onde nós podemos perceber que logo em seguida é exigido a entrada de dados, como nome do autor, editora, nome do livro, ano de lançamento, entre outros.

Depois entramos na verificação de dados, onde ‘rescue’ está fazendo o mesmo papel do try catch em C#. Podemos notar que a partir do momento em que ocorrer algum erro, o programa irá retornar uma mensagem de erro de processamento, e logo em seguida fecha encerra o processo do programa.

Caso a consulta seja bem sucedida, o programa oferece a opção de fazer uma nova consulta dentro do banco de dados.

Inicialmente, para conectar-se com o banco de dados, precisará que o sqlite3 e a gem do sqlite3 no cmd sejam instalados. Após isso, cria-se uma classe para interagir com o banco de dados e que contenha os comandos necessários para o funcionamento do programa.

Em Ruby, a classe LivroDAL, primeiramente, irá utilizar o comando def para definir um novo método para conectar-se ao banco, cujo possui um comando - SQLite3::Database.open - dentro da variável "@db", para que abra o bdd ou que crie um e assim, abra-o, caso não exista ainda. Usando ainda a mesma variável, irá utilizar o comando "execute", seguidamente, o qual manda as linhas de sql para o cmd e através dele que obtêm-se os comandos em execução.

```
require 'sqlite3'

class LivroDAL

  def conecta
    begin #begin e rescue equivalem ao try catch
      @db = SQLite3::Database.open "livraria.db"

      @db.execute "CREATE TABLE IF NOT EXISTS Tablivros2(id INT PRIMARY
KEY,titulo varchar(30), autor varchar(30), editora varchar(30), ano
varchar(30))"
```

Em java, na classe "LivroDAL", está utilizando o comando privado e fora do comando que irá exibir os códigos que compoem a execução, o *Connection*, está estabelecendo conexão com bdd através da variável "con".

```
public class LivroDAL {
   private static Connection con;
```

Assim como o *begin* em Ruby, em java é utilizado o comando *try*, que possui entre chaves o comando *Class.ForName* que está se conectando com o banco e, também, o comando *DriverManager.getConnection*, o qual está começando a execução.

```
public static void conecta(String _bd)
{
   Erro.setErro(false);
   try
   {
      Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
      con = DriverManager.getConnection("jdbc:ucanaccess://" + _bd);
   }
```

Ainda definindo o método "conecta" em Ruby, pode-se observar que o comando "rescue", que tem o funcionamento de resgatar algum erro eminente, irá colocar o sqlite para executá-lo e assim, obter qual seria o possível incoveninente e, com isso, exibir qual foi.

```
      rescue SQLite3::Exception => e
      puts "Exception occurred"
      puts e
```

Em java, o mesmo ocorre, porém, com a utilização de *catch (Exception e)*, o qual procura e identifica o erro e, entre chaves, o *Erro.setErro(e.getMessage())*, está exibindo com uma mensagem qual foi o erro.

```
catch (Exception e)
    {
      Erro.setErro(e.getMessage());
    }
```

Em uma outra *public* "desconecta" em java, está usando *try catch*, comando que irá desconectar o programa com o banco de dados. Entre chaves, está uma variável “con” juntamente ao “.close”, comando que irá finalizar a comunicação. Caso haja algum erro no processo de execução, irá aparecer uma mensagem que dirá qual erro está se manifestando.

```
public static void desconecta()
  {
    Erro.setErro(false);
    try
    {
      con.close();
    }
    catch (Exception e)
    {
      Erro.setErro(e.getMessage());
    }
```

Diferentemente, emRuby, é feita a desconexão dentro de "conecta", onde estão os comandos de conexão com o banco de dados, do contrário de Java, que utilizou outra *public static void* para o ocorrido. É utilizado, em vez de *try catch*, o comando *ensure*, que irá garantir que o programa desconecte com o bdd. E igualmente em Java, há a presença da variável junto com o comando “.close”, mas sem os parênteses e ponto e vírgula no final da declaração.

```
    ensure
      @db.close if @db
end
```

```
    end
```

Em Ruby, para a definição dos códigos para que o usuário consiga inserir os livros, é necessário, utilizar “(livro=Livro.new)”, que está criando um novo espaço no bdd. A partir disto, é usado uma variável para chamar o banco e já localizar o espaço que será ocupado. São criadas novas variáveis para inserir todos os dados do livro que será incluído, assim como acontece na declaração do id: “id=livro.getCodigo().to_i”, estando este com “to.i”, diferente dos demais, pois seria uma variável com números inteiros e não string. Após isso, serão utilizados os mesmos comandos para execução e armazenamento, seguido de desconexão.

```
  def inseriLivro(livro=Livro.new)
    begin
      @db = SQLite3::Database.open "livraria.db"
      id=livro.getCodigo().to_i
    titulo=livro.getTitulo().to_s
    [...]
    @db.execute "INSERT INTO TabLivros2 (id,titulo,autor,editora,ano)
VALUES (#{id},'#{titulo}','#{autor}','#{editora}','#{ano}')"
    rescue SQLite3::Exception => e
      puts "Exception occurred"
      puts e
    ensure
      @db.close if @db
    end
    end
```

Em Java, ocorre o mesmo, porém com comandos diferentes, por exemplo, *try catch* para inserir o novo livro; na declaração das variáveis que estão recebendo os objetos já criados é utilizado “.setString”, o que transformará os valores em textos. Diferentemente, o processo de execução do banco de dados já inicia antes das novas declarações, e também, não é preciso outra public, como anteriormente visto, para a desconexão e para atualizar com “executeUpdate”.

```
public static void inseriLivro(Livro umlivro)
  {
     Erro.setErro(false);
     try
     {
        PreparedStatement st = con.prepareStatement("insert into TabLivros
(titulo,autor,editora,ano,localizacao) values (?,?,?,?,?)");
        st.setString(1,umlivro.getTitulo());
        st.setString(2,umlivro.getAutor());
      [...]
        st.executeUpdate();
        st.close();
     }
```

```
    catch(Exception e)
    {
      Erro.setErro(e.getMessage());
    }
  }
```

Em Ruby, para a consulta ocorrer é necessário, igualmente nas outras definições, chamar o banco de dados primeiro para que haja comunicação, e em seguida, a consulta será feita a partir da digitação correta do id do livro. Após esse processo, é criado a variável "stm" que irá selecionar o livro arquivado a partir do id, com auxílio do comando ".prepare" e ".execute" em outra variável – "rs". A nova variável "rs", seguida de ".each", o qual irá executar cada elemento do bloco de código que irá fazer a busca; o comando *row.join "/s",* é um array Ruby. Os três campos são unidos com um caractere de espaço para formar uma linha usando o joinmétodo. Sendo assim, será finalizado com o processo de execução para localizar o livro.

```
  def consultaLivro(livro=Livro.new)
    begin
      @db = SQLite3::Database.open "livraria.db"
      id= livro.getCodigo().to_i
      stm = @db.prepare "SELECT * FROM TabLivros2 WHERE id=#{id}"
      rs = stm.execute

      rs.each do |row|
        puts row.join "\s"

      rescue SQLite3::Exception => e
        puts "Exception occurred"
        puts e
  end
    end
  end
end
```

Em Java, inicialmente, é declarado "ResultSet rs", que está guardando os dados do banco de dados dentro da variável "rs". Usando os comandos *try catch*, inicialmente, será localizado o espaço que está o livro no arquivo, usando *PreparedStatement.* Em seguida, assim como Ruby, é declarado as informações dentro do objeto de procura, o título, e iniciando a execução com ".executeQuery". o If presente será necessário para conferir todos os objetos que há em todos os livros, se estiverem corretos, a consulta será e o processo finalizará, caso contrário, o livro não será localizado e retornará para que o usuário coloque o título certo. Caso exista algo indevido, aparecerá uma mensagem informando o ocorrido.

```
public static void consultaLivro(Livro umlivro)
  {
```

```
        ResultSet rs;

        try
        {
           PreparedStatement st = con.prepareStatement("SELECT * FROM TabLivros WHERE titulo=?");
           st.setString(1,umlivro.getTitulo());
           rs = st.executeQuery();
           if (rs.next())
           {
              umlivro.setAutor(rs.getString("autor"));
              [...]
           }
           else
           {
              Erro.setErro("Livro nao localizado.");
              return;
           }
           st.close();
        }
        catch(Exception e)
        {
           Erro.setErro(e.getMessage());
        }
     }
  }
```

10. CAPÍTULO 6: INTEGRAÇÃO DA LINGUAGEM COM RECURSOS DE TERCEIROS

Para esse capitulo, elaboramos um programa que pega os dados da previsão do tempo da região mais próxima da localização atual.

O que precisou:

Instalação “gem install json” e “gem install rest-client” no prompt de comando, o json é o formato que os dados vão ser disponibilizados pelo site e o rest-client é responsável pela conexão com o site

Criar uma conta no site hgBrasil, desse site que é retirada as informações necessárias. Ao criar a conta você ganha uma chave de acesso que precisa ser inserida junto ao link da API, uma forma de se autenticar pra pegar os dados

Programa:

```
require 'json'
require 'rest-client'

url = "https://api.hgbrasil.com/weather?format=json-cors&key=63c1c270"

response = RestClient.get url
result = JSON.parse response.to_s

temp = result['results']['temp']
date= result['results']['date']
description= result['results']['description']

puts "Temperatura: #{temp}"
puts "Data: #{date}"
puts "Descrição: #{description}"
```

Começamos o programa com o comando "require", para importar as bibliotecas. E logo após, armazenamos o link do hgbbrasil, em uma variavel, declarada como "url".

Em seguida, utilizamos o comando "response", para acessar e conseguir os dados que existão dentro daquele link. Sendo assim, os dados foram armazenados no formato JSON.

Com os dados armazenados no foormato JSON, criamos 3 variaveis para armazenar os dados. O comando "result" acaba buscando o dado que desejamos, dentro do site, sendo assim, buscando "temp", "date" e "description".

E por fim, o utilizamos o comando "puts" para exibir os resultados coletados.