**Guia de Consulta C#**

**Escola SENAI de Informática**

**Professor Jorge Rabello**

**2017**

**São Paulo – SP**

Sumário

[1. Variáveis, Sintaxe e Tipos de Dados Primitivos 6](#_Toc498112901)

[Variáveis 6](#_Toc498112902)

[Sintaxe 7](#_Toc498112903)

[Exercícios 8](#_Toc498112904)

[2. Tipos de Dados Primitivos 10](#_Toc498112905)

[3. Constantes 13](#_Toc498112906)

[4. Operadores 14](#_Toc498112907)

[Operadores Aritméticos 15](#_Toc498112908)

[Expressões 15](#_Toc498112909)

[Exercícios 16](#_Toc498112910)

[Conversão estática 18](#_Toc498112911)

[Formatando os resultados 21](#_Toc498112912)

[Evitando erros 21](#_Toc498112913)

[Exercícios 22](#_Toc498112914)

[Operadores de Comparação 22](#_Toc498112915)

[Operadores Lógicos 23](#_Toc498112916)

[Operador Lógico && (e) 24](#_Toc498112917)

[Operador Lógico || (ou) 24](#_Toc498112918)

[Operador Lógico ! (não) 24](#_Toc498112919)

[Operadores de Atribuição 25](#_Toc498112920)

[Operadores Especiais 26](#_Toc498112921)

[Operador Ternário 26](#_Toc498112922)

[Chamada de método 27](#_Toc498112923)

[Separador de expressões 27](#_Toc498112924)

[5. Desafio I Calculo de IMC 28](#_Toc498112925)

[6. Vetores 31](#_Toc498112926)

[Declarando um vetor 31](#_Toc498112927)

[Contando quanto item há no vetor 32](#_Toc498112928)

[Declarando vetores, inicializando e acessando os dados 33](#_Toc498112929)

[7. Matrizes 33](#_Toc498112930)

[Declarando uma matriz 34](#_Toc498112931)

[Acessando os dados de uma matriz 34](#_Toc498112932)

[Contando quantos item há no total na matriz 34](#_Toc498112933)

[8. ArrayList 37](#_Toc498112934)

[Declarando um ArrayList 38](#_Toc498112935)

[Adicionando items ao ArrayList 38](#_Toc498112936)

[Obtendo o total de item do ArrayList 38](#_Toc498112937)

[Exibindo os dados do ArrayList em uma ComboBox 38](#_Toc498112938)

[Removendo elementos do ArrayList e atualizando os valores 38](#_Toc498112939)

[Verificando se um elemento está no ArrayList 38](#_Toc498112940)

[Otendo o índice de um elemento do ArrayList 38](#_Toc498112941)

[Ordenando os elementos do ArrayList 38](#_Toc498112942)

[9. Estrutura de Decisão Condicional 39](#_Toc498112943)

[Exercícios 44](#_Toc498112944)

[10. Estrutura de Seleção Condicional 45](#_Toc498112945)

[Exercícios 46](#_Toc498112946)

[11. Desafio II – Desafio dos Dados 47](#_Toc498112947)

[12. Estruturas de Repetição 48](#_Toc498112948)

[Laço de repetição for 49](#_Toc498112949)

[Laço de repetição while 51](#_Toc498112950)

[Laço de repetição do... while 53](#_Toc498112951)

[Diferença entre while e do...while 54](#_Toc498112952)

[Laço de repetição foreach 55](#_Toc498112953)

[Obtendo valores de um ArrayList 55](#_Toc498112954)

[Exercícios 55](#_Toc498112955)

[13. Exercícios Complementares 56](#_Toc498112956)

[14. Orientação a objetos 59](#_Toc498112957)

[Abstração 59](#_Toc498112958)

[Atributos 60](#_Toc498112959)

[Métodos 61](#_Toc498112960)

[Classe 62](#_Toc498112961)

[Exercícios 67](#_Toc498112962)

[15. Métodos 68](#_Toc498112963)

[Métodos sem retorno 68](#_Toc498112964)

[Métodos com retorno 73](#_Toc498112965)

[VarArgs – Argumentos variáveis para métodos 76](#_Toc498112966)

[16. Desafio: Extrair a raiz quadrada de um número utilizando a Equação de Pell 78](#_Toc498112967)

[Métodos sobrecarregados 79](#_Toc498112968)

[17. Namespaces 81](#_Toc498112969)

[18. Modificadores de Acesso e Encapsulamento 81](#_Toc498112970)

[Exercício 86](#_Toc498112971)

[19. Variáveis e métodos estáticos 87](#_Toc498112972)

[Exercícios 90](#_Toc498112973)

[20. Construtores 91](#_Toc498112974)

[Exercícios 94](#_Toc498112975)

[21. Composição 95](#_Toc498112976)

[Acessando os atributos da composição 97](#_Toc498112977)

[Exercícios 98](#_Toc498112978)

[22. Enumerações 99](#_Toc498112979)

[23. Herança 100](#_Toc498112980)

[23.1. Herança e construtores 103](#_Toc498112981)

[24. Polimorfismo 105](#_Toc498112982)

[24.1. Virtual e Override 105](#_Toc498112983)

[24.2. Abstract 106](#_Toc498112984)

[24.3. Métodos e classes sealed 107](#_Toc498112985)

[24.4. Interfaces 108](#_Toc498112986)

[24. Banco de Dados 113](#_Toc498112987)

[24.2. Diferença entre dados e informação 113](#_Toc498112988)

[24.2. Formas de Armazenamento de Dados 113](#_Toc498112989)

[24.3. Sistema de banco de dados 115](#_Toc498112990)

[25. Linguagem SQL 115](#_Toc498112991)

[26. Subdivisão de Instruções SQL 115](#_Toc498112992)

[26.1. DDL (Data Definition Language) 115](#_Toc498112993)

[26.2. DML (Data Manipulation Language) 116](#_Toc498112994)

[26.3. DCL (Data Control Language) 116](#_Toc498112995)

[26.4. DTL (Data Transaction Language) 116](#_Toc498112996)

[26.5 Exercícios DDL 117](#_Toc498112997)

[26.6. Exercícios DML 118](#_Toc498112998)

[26.6.1. SELECT 118](#_Toc498112999)

[26.6.2. INSERT 118](#_Toc498113000)

[26.6.3. UPDATE 119](#_Toc498113001)

[26.6.4. DELETE 119](#_Toc498113002)

[26.6.5. Relatórios 119](#_Toc498113003)

[26.6.6. Relatórios com JOIN 120](#_Toc498113004)

[27. Construindo uma aplicação para cadastrar funcionários 122](#_Toc498113005)

[27.1. Diagrama de Banco de Dados 122](#_Toc498113006)

[27.2. Diagrama de Classes 122](#_Toc498113007)

[27.3. Mockups das Telas 123](#_Toc498113008)

[27.4. Cadastro de Funcionários 123](#_Toc498113009)

[27.5. Cadastro de Dependências 124](#_Toc498113010)

[27.6. Cadastro de Dependentes 124](#_Toc498113011)

[28. Código Fonte: 125](#_Toc498113012)

[28.1.1. Senai.RH.Model.Dependente.cs 125](#_Toc498113013)

[28.1.2. Senai.RH.Model.Dependencia.cs 127](#_Toc498113014)

[28.1.3. Senai.RH.Model.Funcionario.cs 128](#_Toc498113015)

[28.1.4. Senai.RH.Dao.ConnectionFactory.cs 130](#_Toc498113016)

[28.1.5. Senai.RH.Dao.IDao.cs 131](#_Toc498113017)

[28.1.6. Senai.RH.Dao.FuncionarioDao.cs 132](#_Toc498113018)

[28.1.7. Senai.RH.Dao.DependenciaDao.cs 137](#_Toc498113019)

[28.1.8. Senai.RH.Dao.DependenteDao.cs 140](#_Toc498113020)

[28.1.9. Senai.RH.Forms.HomeForm.cs 144](#_Toc498113021)

[28.1.10. Senai.RH.Forms.FuncionarioForm.cs 146](#_Toc498113022)

[28.1..11 Senai.RH.Forms.DependenteForm.cs 150](#_Toc498113023)

[29. Gerando relatórios 154](#_Toc498113024)

[30. Delegates e Eventos 160](#_Toc498113025)

[30.1. Eventos 160](#_Toc498113026)

[30.1. Delegates na prática 161](#_Toc498113027)

[31. Consumindo WebServices 164](#_Toc498113028)

[Bibliografia 169](#_Toc498113029)

# Variáveis, Sintaxe e Tipos de Dados Primitivos

## Variáveis

As variáveis servem para que possamos armazenar e processar informações. Quando utilizamos uma calculadora por exemplo, em algum momento a calculadora precisará armazenar os números para a operação, a operação que queremos realizar e o resultado, em um sistema bancário, precisaríamos armazenar dados como o saldo da conta, nome do cliente dono da conta, entre outras coisas.

Para que esses dados sejam armazenados teremos de fazer com que o C# reserve regiões na memória, essas regiões são conhecidas como variáveis.

As variáveis guardam informações de tipos específicos, podemos armazenar o número de uma conta bancária em um tipo numérico ou o nome do correntista em um tipo de texto.

Provavelmente na escola você deve já ter se deparado com algo parecido com o seguinte:

a = 7

b = 10

c = a + b

Nesse exemplo a, b, e c é o que podemos chamar de variáveis, e consequentemente cada uma dessas letras representam um valor, sendo que a letra c representaria o valor de 17 (total da soma de 7 mais 10).

Acima fizemos algumas operações com as variáveis, por exemplo, quando fazemos a=7 estamos atribuindo à variável a o valor de 7, então se lê-se a recebe 7.

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

string nome = "Jorge";

int idade = 29;

bool casado = true;

MessageBox.Show("Nome: " + nome);

MessageBox.Show("Idade: " + idade);

MessageBox.Show("Casado: " + casado);

}

## Sintaxe

Sempre que desejarmos criar variáveis devemos seguir a seguinte sintaxe:

tipo\_de\_dados identificador = valor;

O tipo de dados serve para informar se vamos armazenar um texto, um número ou um valor lógico (verdadeiro ou falso).

O identificador serve para que seja possível recuperar, posteriormente o valor armazenado na variável.

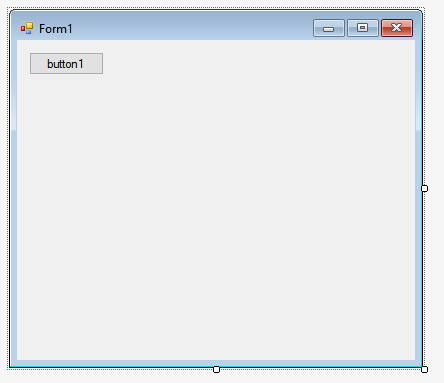
O operador de atribuição (operador de igual) serve para dizer qual a informação que vamos armazenar na variável.

Para se nomear as variáveis existem algumas regras:

1. Utilize apenas caracteres do alfabeto
2. Não inicie o nome da variável com números
3. Não utilize espaços
4. Utilize um nome significativo
5. Não utilize palavras reservadas da linguagem C#[[1]](#footnote-1)

## Exercícios

1. Crie um novo projeto do tipo Aplicativo do Windows Forms e dê a ele o nome de OlaMundo.
2. Construa a seguinte interface gráfica



1. Dê um duplo clique sobre o botão e programe com as instruções seguintes:

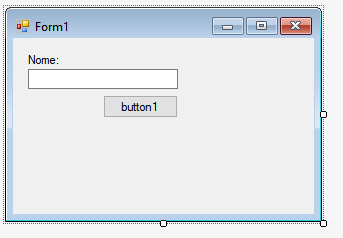
private void button1\_Click(object sender, EventArgs e)

{

MessageBox.Show("Ola Mundo");

}

1. Execute o programa e faça um teste
2. Agora vamos modificar a nossa interface gráfica para que o usuário consiga digitar seu nome e receba uma saudação. Para isso utilizaremos o componente TextBox e o componente Label



Label

TextBox

Button

1. Clique novamente duas vezes no botão e altere o código fonte

private void button1\_Click(object sender, EventArgs e)

{

string nome = textBox1.Text;

MessageBox.Show("Ola " + nome);

}

1. Faça um teste novamente

Explicando o código:

string nome = textBox1.Text;

Nessa linha estamos declarando uma variável do tipo string (esse tipo é utilizado para armazenar texto), o identificador da variável é nome, os identificadores servem para que seja possível recuperar posteriormente o valor que a variável armazenou.

Utilizamos o operador de atribuição (sinal de igual) para dizer que desejamos armazenar um valor na variável.

O valor a ser armazenado é o texto digitado pelo usuário na TextBox textBox1.

MessageBox.Show("Ola " + nome);

Nessa linha nós exibimos uma caixa de mensagem unindo as palavras Olá e o conteúdo armazenado na variável.

# Tipos de Dados Primitivos

Os tipos de dados são utilizados para que o computador saiba quanto de memórias precisa reservar para a variável, por exemplo quando você declarar uma variável do tipo int (para armazenar um número do tipo inteiro o computador reserva 16 bits (16 espaços com zeros e uns) na memória.

Os tipos de dados podem ser divididos basicamente em:

* Tipos de Texto ou Alfanuméricos: Utilizados para armazenar palavras são conhecidos também como tipos literais.
* Tipos Numéricos: Utilizados para armazenar números, que podem ser inteiros ou com casas decimais.
* Tipo Lógico: Utilizado para armazenar valores lógicos (verdadeiro ou falso)

A linguagem C# possui seus próprios tipos de dados:

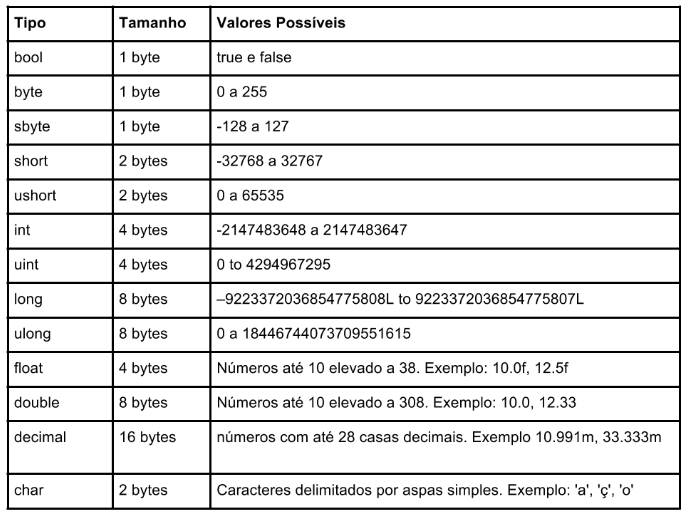


Figura 1Tabela de tipos de dados

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

// tipo lógico

// 1 Byte (8 bits)

bool logico = false;

// byte (8 bits)

// 0 até 255

byte meuByte = 255;

byte byteBinario = 0b01010101;

// sbyte (short byte - byte pequeno)

// 1 byte (8 bits)

// -128 até 127

sbyte meuSByte = 127;

sbyte sbyteBinario = 0b01010101;

// short

// -32768 até 32767

// 2 bytes (16 bits)

short meuHShort = 32\_767;

short shortBinario = 0b01010101\_01010101;

// ushort (unsigned - sem sinal)

// 0 65\_535

// 2 bytes (16 bits)

ushort meuUShort = 65\_535;

ushort ushortBinario = 0b01010101\_01010101;

// int

// -2\_147\_843\_648 até 2\_147\_843\_647

// 4 bytes ou 32 bits

int meuInt = 2\_000\_000\_000; // 2 bilhões

int intBinario = 0b01010101\_01010101\_01010101\_01010101;

// long

//9\_233\_372\_036\_854\_775\_807L

// o L indica que esse valor é do tipo long

// 8 bytes (64 bits)

// 9 quintilhões

long meuLong = 9\_000\_000\_000\_000\_000\_000L;

// ulong

// 0 até 18\_446\_744\_073\_709\_551\_615

// 8 bytes (64 bits)

ulong meuULong = 18\_446\_744\_037\_709\_551\_616;

// decimais (R$ 12.75)

// float

// 10 elevado a 308 1E308 (12.75, -15.85...)

// 4 bytes

float meuFloat = 15.43F; // R$ 15,43

// double

// 10 elevado a 308 1E308

// 8 bytes

double meuDouble = 1E+308; // IEEE 754

// decimal

// números com até 28 casas decimais

// 16 bytes

decimal meuDecimal = 10.991M;

// representa um caractere

// 2 bytes

char meuChar = 'M'; // unicode

// texto

char numero = '7'; // unicode

// tipo especial de dados para textos

string curso = "Programação C#";

MessageBox.Show("Valor: " + meuChar);

}

# Constantes

Alguns dados não devem ser modificados em uma aplicação, alguns valores devem permanecer fixos, e para estes casos utilizamos as constantes.

Uma constante é uma variável que não pode ser modificada, ou seja uma vez atribuído o seu valor não se poderá modifica-lo.

Para se criar constantes na linguagem C# utilizamos a palavra reservada const.

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

// mutável

double pi = 3.1415926543;

// imutável (constante)

const double VALOR\_PI = 3.1415926543;

// VALOR\_PI = 28;

MessageBox.Show("PI " + VALOR\_PI);

}

# Operadores

Os operadores são utilizados para que seja possível manipular e fazer operações com as variáveis.

Os operadores são classificados de acordo com os operandos que manipulam e pode ser dos seguintes tipos:

* Unários – Manipulam apenas um operando  
  -1
* Binários – Manipulam dois operandos

1. + 5

* Ternários – Manipulam três operandos

5 > 6 : “Maior que 6” : “Menor que 6”

Uma lista completa de operadores pode ser encontrada em <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/>.

## Operadores Aritméticos

Os operadores aritméticos ou matemáticos servem para que seja possível realizar cálculos.

|  |  |
| --- | --- |
| OPERADOR | UTILIDADE |
| + | Adição |
| - | Subtração |
| \* | Multiplicação |
| / | Divisão |
| % | Módulo ou resto de divisão |

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

int x = 10;

int y = 2;

int resultado = 0;

// adição +

// resultado = 10 + 2

resultado = x + y;

// subtracao -

// resultado = 10 - 2

resultado = x - y;

// multiplicação \*

// resultado = 10 \* 2

resultado = x \* y;

// divisão /

// resultado = 10 / 2

resultado = x / y;

// módulo %

// resultado = 10 % 2

resultado = x % y;

int a = 3;

int b = 2;

// resultado = 3 % 2

resultado = a % b;

MessageBox.Show("Resultado: " + resultado);

}

## Expressões

Na escola aprendemos a solucionar expressões matemáticas resolvendo os cálculos sempre da esquerda para a direita, também aprendemos que há prioridades entre os operadores

Assim sendo

5 + 3 \* 2 =11

Isso por que 2 \* 3 resulta em 6 e 6 somado à 5 resulta em 11.

Porém podemos alterar a prioridade adicionando parênteses as expressões:

(5 + 3) \* 2 =16

Nesse caso a adição será realizada primeiro então 5 + 3 resulta em 8 e 8 vezes 2 resultam e 16.

Exemplo:

resultado = 5 + 3 \* 2; // 11

MessageBox.Show("Resultado: " + resultado);

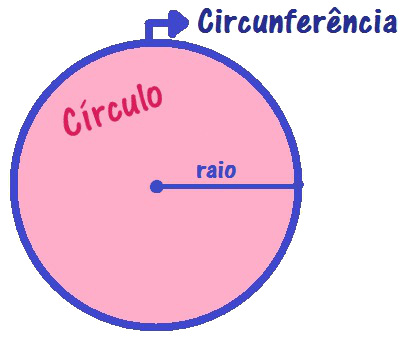
resultado = (5 + 3) \* 2; // 16

MessageBox.Show("Resultado: " + resultado);

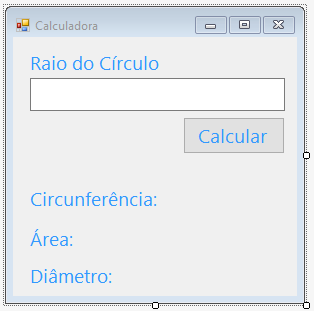
## Exercícios

Vamos desenvolver um programa que faz alguns cálculos com um círculo, vamos calcular o diâmetro, a circunferência e a área de um círculo:

´Todos esses cálculos são realizados utilizando o raio do circulo:



Para começar crie um novo projeto do Windows Forms e desenhe a seguinte interface gráfica:



lblDiametro

lblArea

lblCircunferencia

btnCalcular

txtRaio

lblRaio

**Propriedades dos componentes:**

|  |  |
| --- | --- |
| **lblRaio** | |
| Name | lblRaio |
| FontName | Segoe UI |
| FontSize | 14 |
| ForeColor | Highlight |
| **txtRaio** | |
| Name | txtRaio |
| FontName | Segoe UI |
| FontSize | 14 |
| **btnCalcular** | |
| Name | btnCalcular |
| FontName | Segoe UI |
| FontSize | 14 |
| **lblCircunfrencia** | |
| Name | lblCircunfrencia |
| FontName | Segoe UI |
| FontSize | 14 |
| **lblArea** | |
| Name | lblArea |
| FontName | Segoe UI |
| FontSize | 14 |
| **lblDiametro** | |
| Name | lblDiametro |
| FontName | Segoe UI |
| FontSize | 14 |

## Conversão estática

Para que o nosso programa funcione será necessário realizarmos algumas conversões, pois os valores do TextBox sempre serão do tipo string.

Para podermos fazer cálculos temos que converter o tipo string em algum tipo numérico, essas conversões podem ser realizadas por meio do método Parse, por exemplo:

double numero = Double.Parse("123");

A linha acima converte o texto “123” no tipo numérico double.

int numero = int.Parse("123");

A linha acima converte o texto “123” no tipo numérico int.

byte numero = Byte.Parse("123");

A linha acima converte o texto “123” no tipo numérico byte.

E assim por diante.

Sendo assim clique duas vezes no botão calcular e vamos começar.

O primeiro calculo que vamos fazer é o da circunferência, esse cálculo é realizado por meio da fórmula: circunferência = 2 \* pi \* raio.

Para obter o valor de PI podemos utilizar uma classe[[2]](#footnote-2) da linguagem C# chamada Math, que possui diversos métodos para cálculos matemáticos.

Na classe Math há uma constante chamada PI (Math.PI).

Assim sendo a primeira coisa que precisamos fazer é obter o valor do raio que o usuário digitou no TextBox, para isso devemos obter a propriedade Text da TextBox e armazenar o valor em uma variável do tipo string.

string raioTxt = txtRaio.Text;

Feito isso deveremos converter a variável string raioTxt em um valor numérico.

double raio = Double.Parse(raioTxt);

E agora podemos calcular a circunferência e guardar o resultado do calculo em uma variável.

double circunferencia = 2 \* Math.PI \* raio;

Por fim devemos atribuir o valor calcular ao texto da label lblCircunferencia

lblCircunferencia.Text = "Circunferência " + circunferencia;

O código completo fica assim:

private void btnCalcular\_Click(object sender, EventArgs e)

{

// calculo da circunferência: 2 \* PI \* raio

string raioTxt = txtRaio.Text;

double raio = Double.Parse(raioTxt);

double circunferencia = 2 \* Math.PI \* raio;

lblCircunferencia.Text = "Circunferência " + circunferencia;

}

Para fazer os demais cálculos já temos os valores de que precisamos

string raioTxt = txtRaio.Text;

double raio = Double.Parse(raioTxt);

Sendo assim basta que realizemos os cálculos necessários:

Para calcular a área vamos utilizar a seguinte fórmula: área = PI \* raio2.

Pra realizar a operação de potência podemos mais uma vez recorrer a classe Math, nela temos um método chamado Pow(base, expoente) com o qual podemos realizar cálculos de potenciação:

double area = Math.PI \* Math.Pow(raio, 2);

Sendo assim para calcular a área e atribuir o valor calculado à sua label o código completo fica:

// calculo da área PI \* raio2

double area = Math.PI \* Math.Pow(raio, 2);

lblArea.Text = "Área: " + area;

O nosso código fonte completo será:

private void btnCalcular\_Click(object sender, EventArgs e)

{

string raioTxt = txtRaio.Text;

double raio = Double.Parse(raioTxt);

// calculo da circunferência

// 2 \* PI \* raio

double circunferencia = 2 \* Math.PI \* raio;

lblCircunferencia.Text = "Circunferência: " + circunferencia;

// calculo da área

// PI \* raio2

double area = Math.PI \* Math.Pow(raio, 2);

lblArea.Text = "Área: " + area;

}

Para o cálculo do diâmetro da circunferência utilizaremos a seguinte fórmula 2 \* PI \* raio, assim sendo o código para este cálculo fica assim:

double diametro = 2 \* raio;

lblDiametro.Text = "Diâmetro: " + diametro;

O código fonte completo ficou assim:

private void btnCalcular\_Click(object sender, EventArgs e)

{

string raioTxt = txtRaio.Text;

double raio = Double.Parse(raioTxt);

// calculo da circunferência

// 2 \* PI \* raio

double circunferencia = 2 \* Math.PI \* raio;

lblCircunferencia.Text = "Circunferência: " + circunferencia;

// calculo da área

// PI \* raio2

double area = Math.PI \* Math.Pow(raio, 2);

lblArea.Text = "Área: " + area;

// calculo do diametro

// 2 \* PI \* raio

double diametro = 2 \* raio;

lblDiametro.Text = "Diâmetro: " + diametro;

}

## Formatando os resultados

Para formatar os resultados com duas casas decimais podemos utilizar o método ToString(), passado como parâmetro a quantidade seguida de casas decimais que queremos exibir, por exemplo:

lblCircunferencia.Text = "Circunferência: " + circunferencia.ToString("N2");

Formata o valor da variável para duas casas decimais

## Evitando erros

Se por algum acaso o usuário digitar uma letra no campo onde deveria digitar o raio da circunferência, o nosso programa vai travar e fechar.

Para evitar que isso ocorra vamos utilizar o método TryParse no lugar do método Parse.

O método TryParse faz com que ocorra uma validação e evita que o programa encerre caso haja algum erro.

Sendo assim vamos alterar essa parte do nosso programa

string raioTxt = txtRaio.Text;

double raio = Double.Parse(raioTxt);

Altere o código da seguinte forma:

string raioTxt = txtRaio.Text;

double raio;

Double.TryParse(raioTxt, out raio);

A linha Double.TryParse(raioTxt, out raio); converte a string raioTxt em double e armazena o valor convertido na variável raio.

Dessa forma o nosso programa não fechará caso o usuário cometa um erro.

## Exercícios

1. Solicitar que o usuário digite uma temperatura em ºC e apresentá-la convertida em ºF. A fórmula de conversão é: **f = (9 \* celsius + 160) / 5**, sendo F a temperatura em Fahrenheit e C a temperatura em Celsius.
2. Solicitar que o usuário digite uma temperatura em ºF e apresentá-la convertida em ºC. A fórmula de conversão é: **c = ((fahrenheit – 32 ) \* 5 ) / 9**, sendo F a temperatura em Fahrenheit e C a temperatura em Celsius.
3. Calcular e apresentar o volume de uma lata de óleo, utilizando a fórmula **volume = 3.141592654 \* raio2 \*altura**.
4. Elaborar um programa que calcule e apresente o volume de uma caixa retangular, utilizando a fórmula **volume = comprimento \* largura \* altura**.

## Operadores de Comparação

Como o próprio nome já diz os operadores de comparação servem para comparar valores e sempre resultam em um valor lógico (verdadeiro ou falso).

|  |  |
| --- | --- |
| OPERADOR | UTILIDADE |
| > | Maior |
| < | Menor |
| >= | Maior ou igual |
| <= | Menor ou igual |
| != | Diferente |
| == | Igual |

Exemplos:

private void button1\_Click(object sender, EventArgs e)

{

int x = 6;

MessageBox.Show("Resultado: " + (x == 6)); // true

MessageBox.Show("Resultado: " + (x != 6)); // false

MessageBox.Show("Resultado: " + (x != 7)); // true

MessageBox.Show("Resultado: " + (x > 7)); // false

MessageBox.Show("Resultado: " + (x > 5)); // true

MessageBox.Show("Resultado: " + (x >= 5)); // true

MessageBox.Show("Resultado: " + (x >= 6)); // true

MessageBox.Show("Resultado: " + (x >= 7)); // false

}

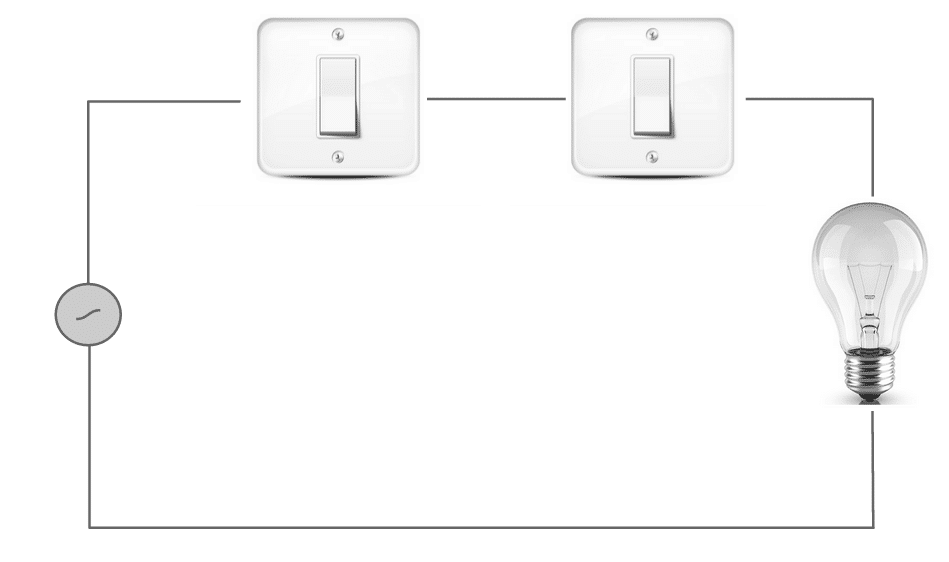
## Operadores Lógicos

Os operadores lógicos são capazes de avaliar duas expressões de comparação e com base nos resultados da avaliação podem produzir um novo resultado.

|  |  |
| --- | --- |
| OPERADOR | UTILIDADE |
| && | E, and |
| || | OU, or |
| ! | Não, not |

## Operador Lógico && (e)

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **S** |
| F | F | F |
| V | F | F |
| F | V | F |
| V | V | V |

O operador lógico e resultará em verdadeiro caso as duas condições avaliadas sejam verdadeiras.

B

A

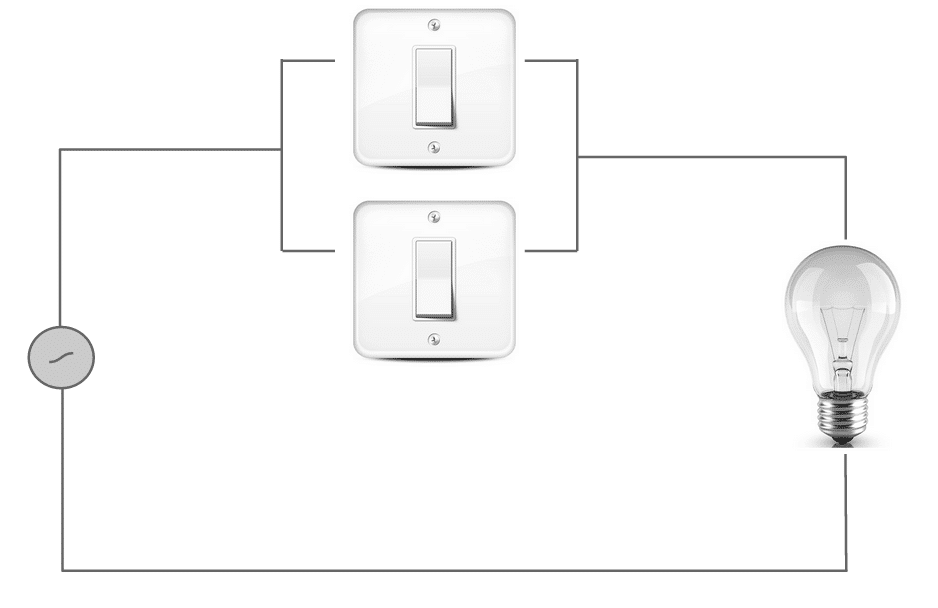
S

## Operador Lógico || (ou)

Por outro lado, há também o operador lógico ou, e nesse caso basta que uma das duas expressões sejam verdadeiras para que o resultado seja verdadeiro.

B

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **S** |
| F | F | F |
| V | F | V |
| F | V | V |
| V | V | V |



A

S

## Operador Lógico ! (não)

O operador lógico não faz com que as operações sejam negadas, isso é, inverte o resultado, então o verdadeiro se torna falso e o falso se torna verdadeiro.

Exemplos :

* 1. Operador lógico && (e):

private void button1\_Click(object sender, EventArgs e)

{

int x = 6;

bool resultado = (x >= 1) && (x <= 10);

// true e true - true

MessageBox.Show("Resultado: " + resultado);

resultado = (x >= 1) && (x <= 5);

// true e false – false

MessageBox.Show("Resultado: " + resultado);

}

* 1. Operador lógico || (ou)

private void button1\_Click(object sender, EventArgs e)

{

int x = 6;

bool resultado = (x >= 1) || (x <= 5);

// true ou false - true

MessageBox.Show("Resultado: " + resultado);

resultado = (x >= 10) && (x <= 1);

// false e false - false

MessageBox.Show("Resultado: " + resultado);

}

* 1. Operador lógico ! (não)

private void button1\_Click(object sender, EventArgs e)

{

int x = 6;

bool resultado = !(x >= 1);

// true só que não então false

MessageBox.Show("Resultado: " + resultado);

resultado = !(x >= 10);

// false só que não então true

MessageBox.Show("Resultado: " + resultado);

}

## Operadores de Atribuição

|  |  |
| --- | --- |
| OPERADOR | UTILIDADE |
| += | Adição |
| -= | Subtração |
| \*= | Multiplicação |

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

int x = 6;

int resultado = x += 3;

MessageBox.Show("Adição: " + resultado); // 9

resultado = x -= 3;

MessageBox.Show("Subtração: " + resultado); // 6

resultado = x \*= 3;

MessageBox.Show("Multiplicação: " + resultado); // 18

}

## Operadores Especiais

|  |  |
| --- | --- |
| OPERADOR | UTILIDADE |
| ? : | Operador ternário |
| ( ) | Chamada de método |
| , | Separador de expressões |

## Operador Ternário

O operador ternário recebe esse nome por que consegue avaliar três operandos.

Este operador pode ajudar o programa a tomar decisões.

Sua sintaxe é expressão\_lógica ? valor\_se\_verdadeiro : valor\_se\_falso;

Exemplo:

private void button1\_Click(object sender, EventArgs e)

{

int idade = 16;

string resultado = idade >= 18 ? "Maior de Idade" : "Menor de Idade";

MessageBox.Show("Resultado: " + resultado);

}

private void button1\_Click(object sender, EventArgs e)

{

double preco = 2\_000\_000;

string decisao = preco <= 1000 ? "Vou comprar" : "Não vou comprar";

MessageBox.Show("Resultado: " + decisao);

}

private void button1\_Click(object sender, EventArgs e)

{

double producao = 200;

string status = producao >= 100 && producao <= 500 ? "OK" : "Erro";

MessageBox.Show("Resultado: " + status);

}

## Chamada de método

O operador de chamada de método é utilizado para indicar que aquela determinada instrução é um método, e serve também para passar argumentos aos métodos.

MessageBox.Show(argumento);

MessageBox.Show("Resultado: " + decisao);

## Separador de expressões

O operador de separação de expressões nos permite declarar variáveis do mesmo tipo em um única linha.

private void button1\_Click(object sender, EventArgs e)

{

string pais = "Brasil";

string nome = "Jorge";

}

private void button1\_Click(object sender, EventArgs e)

{

string pais = "Brasil", nome = "Jorge";

}

# Desafio I Calculo de IMC

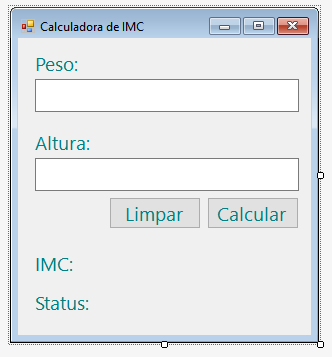
Você foi contratado por uma grande empresa de desenvolvimento de software. Um dos clientes dessa empresa é um hospital e você foi designado para desenvolver um software que calcule o IMC de uma pessoa.

A fórmula para calcular o imc é: imc = altura / peso2.

E uma pessoa está no peso ideal se o imc estiver entre 20 e 25.

O usuário deve digitar o peso e a altura da pessoa e o programa deve calcular e informar o IMC além de informar se a pessoa está dentro ou fora do peso ideal.

Interface Gráfica:



txtAltura

txtPeso

lblStatus

lblImc

lblAltura

lblPeso

btnCalcular

btnLimpar

Vamos começar programando o botão calcular:

Para fazer o cálculo de im primeiro precisaremos obter a altura digitada pelo usuário e converte-la de string para double:

string alturaTxt = txtAltura.Text;

double altura;

Double.TryParse(alturaTxt, out altura);

Depois precisaremos fazer o mesmo com o peso:

string pesoTxt = txtPeso.Text;

double peso;

Double.TryParse(pesoTxt, out peso);

A seguir podemos realizar o cáclulo de imc e armazenar o valor na variável imc:

double imc = peso / Math.Pow(altura, 2);

Tendo a variável imc podemos atribuir seu valor ao texto da label lblImc:

lblImc.Text = "IMC: " + imc.ToString("N2");

Na sequência, o programa deve decidir se o paciente está dentro ou fora do peso ideal, sendo que o imc de um peso ideal varia entre 20 e 25, para essa finalidade, podemos utilizar o operador ternário:

imc >= 20 && imc <= 25 ? "Peso ideal." : "Fora do peso ideal.";

e atribuir o resultado à uma variável chamada status:

string status = imc >= 20 && imc <= 25 ? "Peso ideal." : "Fora do peso ideal.";

para que seja possível utilizar seu valor no label lblStatus

lblStatus.Text = "Status: " + status;

O código completa fica assim:

private void btnCalcular\_Click(object sender, EventArgs e)

{

string pesoTxt = txtPeso.Text;

string alturaTxt = txtAltura.Text;

double peso;

double altura;

Double.TryParse(pesoTxt, out peso);

Double.TryParse(alturaTxt, out altura);

double imc = peso / Math.Pow(altura, 2);

lblImc.Text = "IMC: " + imc.ToString("N2");

string status = imc >= 20 && imc <= 25 ? "Peso ideal." : "Fora do peso ideal.";

lblStatus.Text = "Status: " + status;

}

E a seguir o código do botão limpar.

Para o botão limpar a primeira coisa que deve ser feita é remover o texto das TextBox, isso pode ser feito com o método Clear

txtPeso.Clear();

txtAltura.Clear();

A seguir basta que devolvêssemos o texto original as Labels

lblImc.Text = "IMC: ";

lblStatus.Text = "Status: ";

E por fim colocar o curso no campo de altura:

txtPeso.Focus();

O código completo fica assim:

private void btnLimpar\_Click(object sender, EventArgs e)

{

txtPeso.Clear();

txtAltura.Clear();

lblImc.Text = "IMC: ";

lblStatus.Text = "Status: ";

txtPeso.Focus();

}

# Vetores

Vetores são coleções de dados. Com eles podemos armazenar mais de uma informação do mesmo tipo, por exemplo, podemos armazenar os nomes de 5 alunos, a idade de 10 pessoas e assim por diante.

Podemos representar um vetor da seguinte forma:

|  |  |  |  |
| --- | --- | --- | --- |
| Ricardo | Beatriz | Ana | Pedro |

## Declarando um vetor

private void button1\_Click(object sender, EventArgs e)

{

string[] nomes =

{

"Ricardo",

"Beatriz",

"Ana",

"Pedro"

};

}

Também podemos declarar vetores da seguinte forma:

private void button1\_Click(object sender, EventArgs e)

{

int[] pares = new int[5];

}

## Contando quanto item há no vetor

private void button1\_Click(object sender, EventArgs e)

{

string[] nomes =

{

"Ricardo",

"Beatriz",

"Ana",

"Pedro"

};

MessageBox.Show("Total de Items: " + nomes.Length);

}

Cada elemento do vetor recebe um índice, um número inteiro, para que seja possível recuperar o conteúdo daquela posição:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valor | Ricardo | Beatriz | Ana | Pedro |
| Índice | 0 | 1 | 2 | 3 |

Nesse exemplo para acessarmos o nome Ana teríamos de nos referir à posição 2 do vetor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valor | 2 | 4 | 6 | 8 |
| Índice | 0 | 1 | 2 | 3 |

Nesse outro exemplo para acessarmos o número 4 teríamos de nos referir à posição 1 do vetor.

## Declarando vetores, inicializando e acessando os dados

private void button1\_Click(object sender, EventArgs e)

{

string[] nomes =

{

"Ricardo",

"Beatriz",

"Ana",

"Pedro"

};

MessageBox.Show("Nome: " + nomes[2]);

}

private void button1\_Click(object sender, EventArgs e)

{

int[] pares = new int[5];

pares[0] = 2;

pares[1] = 4;

pares[2] = 6;

pares[3] = 8;

pares[4] = 10;

MessageBox.Show("Par: " + pares[3]);

}

# Matrizes

Matrizes são vetores de mais de uma dimensão. Isso significa que podemos ter uma coleção de dados para armazenar não só o nome mas também o sexo e o estado de uma nome da lista.

|  |  |  |  |
| --- | --- | --- | --- |
| Linha/Coluna | 0 | 1 | 2 |
| 0 | Ricardo | M | SP |
| 1 | Ana | F | RJ |
| 2 | Pedro | M | MG |
| 3 | Beatriz | F | SP |

Note que no caso da matriz temos uma estrutura com linhas e colunas.

## Declarando uma matriz

private void button1\_Click(object sender, EventArgs e)

{

string[,] pessoas =

{

{"Ricardo", "M", "SP"},

{"Ana", "F", "RJ"},

{"Pedro", "M", "MG"},

{"Beatriz", "F", "SP"}

};

}

## Acessando os dados de uma matriz

Para acessar os dados de uma matriz, utilizamos um esquema de [linha][coluna]

|  |  |  |  |
| --- | --- | --- | --- |
| Linha/Coluna | 0 | 1 | 2 |
| 0 | Ricardo | M | SP |
| 1 | Ana | F | RJ |
| 2 | Pedro | M | MG |
| 3 | Beatriz | F | SP |

private void button1\_Click(object sender, EventArgs e)

{

string[,] pessoas =

{

{"Ricardo", "M", "SP"},

{"Ana", "F", "RJ"},

{"Pedro", "M", "MG"},

{"Beatriz", "F", "SP"}

};

string nome = pessoas[0,0];

string sexo = pessoas[0, 1];

string estado = pessoas[0, 2];

string msg = "Nome: " + nome + "\nSexo: " + sexo + "\nEstado: " + estado;

MessageBox.Show(msg);

}

## Contando quantos item há no total na matriz

private void button1\_Click(object sender, EventArgs e)

{

string[,] pessoas =

{

{"Ricardo", "M", "SP"},

{"Ana", "F", "RJ"},

{"Pedro", "M", "MG"},

{"Beatriz", "F", "SP"}

};

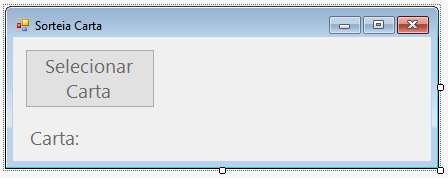
MessageBox.Show("Total: " + pessoas.Length);

}

Exemplo:

Vamos desenvolver um programa que sorteie uma carta de baralho.

Interface Gráfica



btnCarta

lblCarta

Em primeiro lugar devemos declarar dois vetores, um para as faces e outro para os nipes do baralho.

string[] faces =

{

"A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"

};

string[] nipes =

{

"Copas", "Ouros", "Espada", "Paus"

};

Para montar a carta devemos selecionar um índice do vetor de nipes e um índice do vetor de faces, assim poderíamos pensar em algo assim:

string carta = faces[0] + " " + nipes[0];

Porém dessa forma o programa sempre vai sortear a carta A de Copas, então temos que tornar o índice aleatório para isso podemos utilizar a classe Random.

Primeiro precisamos criar uma variável do tipo Random

Random random = new Random();

E depois gerar um número aleatório

random.Next(10);

Nesse caso será sorteado um número de um conjunto de 10 números que vão de 0 (zero) até 9 (nove).

Para que não corramos o risco de ser sorteado um índice que não exista no vetor, vamos definir a quantidade de números baseado no tamanho do vetor

random.Next(faces.Length);

E vamos atribui o número sorteado a uma variável do tipo int

int indiceFace = random.Next(faces.Length);

Para obter a face sorteada, no lugar do número do índice colocaremos a nossa variável indiceFace

faces[indiceFace];

E vamos atribuir a face sorteada à uma variável do tipo string

string face = faces[indiceFace];

Para sortear o nipe vamos fazer do mesmo jeito

int indiceNipe = random.Next(nipes.Length);

string nipe = nipes[indiceNipe];

Por fim para montar a carta basta unir a face e o nipe

string carta = faces + " " + nipe;

Agora basta atribuir a carta à propriedade Text da label lblCarta

lblCarta.Text = "Carta: " + carta;

O código completo ficará da seguinte forma:

private void btnCarta\_Click(object sender, EventArgs e)

{

string[] faces =

{

"A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"

};

string[] nipes =

{

"Copas", "Ouros", "Espada", "Paus"

};

Random random = new Random();

int indiceFace = random.Next(faces.Length);

string face = faces[indiceFace];

int indiceNipe = random.Next(nipes.Length);

string nipe = nipes[indiceNipe];

string carta = faces + " " + nipe;

lblCarta.Text = "Carta: " + carta;

}

# ArrayList

Os ArrayLists são vetores com super-poderes. Os ArrayLists possuem métodos que nos permitem manipular os dados dos vetores de forma mais dinâmica.

Com os ArrayLists podemos adicionar, recuperar, excluir itens do vetor, podemos ainda consultar para saber se encontramos um determinado item no vetor.

O ArrayLists não pertence a api principal do .NET Framework, por tanto para utiliza-lo teremos de declarar que vamos precisar de sua api

using System.Collections;

## Declarando um ArrayList

ArrayList cores = new ArrayList();

## Adicionando items ao ArrayList

cores.Add("Branco");

cores.Add("Vermelho");

cores.Add("Verde");

cores.Add("Azul");

## Obtendo o total de item do ArrayList

MessageBox.Show("Cores: " + cores.Count);

## Exibindo os dados do ArrayList em uma ComboBox

comboBox1.Items.Clear();

comboBox1.Items.AddRange(cores.ToArray());

## Removendo elementos do ArrayList e atualizando os valores

cores.Remove("Verde");

comboBox1.Items.Clear();

comboBox1.Items.AddRange(cores.ToArray());

## Verificando se um elemento está no ArrayList

MessageBox.Show("Tem Verde ? " + cores.Contains("Verde"));

MessageBox.Show("Tem Azul ? " + cores.Contains("Azul"));

## Otendo o índice de um elemento do ArrayList

MessageBox.Show("Posição do Azul: " + cores.IndexOf("Azul"));

## Ordenando os elementos do ArrayList

cores.Sort();

comboBox1.Items.Clear();

comboBox1.Items.AddRange(cores.ToArray());

# Estrutura de Decisão Condicional



As estruturas condicionais permitem que o programa tome decisões baseadas em parâmetros pré-estabelecidos.

A estrutura condicional simples segue a seguinte sintaxe.

if (true)

{

}

Exemplo

int idade = 32;

if (idade >= 18)

{

MessageBox.Show("Adulto");

}

A estrutura condicional ainda pode ser composta, no exemplo acima se a idade for menor que 18 , nada acontecerá, para resolver isso adicionamos a estrutura else

int idade = 15;

if (idade >= 18)

{

MessageBox.Show("Adulto");

}

else

{

MessageBox.Show("Criança");

}

Também podemos utilizar o tipo booleano

bool passou = true;

if (passou)

{

MessageBox.Show("Contratado");

}

Para perguntar se não passou podemos utilizar o operador lógico não

bool passou = true;

if (!passou)

{

MessageBox.Show("Contratado");

}

No teste lógico podemos fazer utilizar expressões

int numero = 5;

if ((numero + 5) == 10)

{

MessageBox.Show("OK");

}

int numero = 10;

if ((numero % 2) == 0)

{

MessageBox.Show("Par");

}

else

{

MessageBox.Show("Ímpar");

}

As estruturas de decisão ainda podem ser encadeadas, veja:

int nota = 6;

if (nota >= 7)

{

MessageBox.Show("Aprovado");

}

else

{

MessageBox.Show("Reprovado");

if (nota == 6)

{

MessageBox.Show("Mas pode fazer a recuperação");

}

}

E podemos contar com os operadores lógicos

int idade = 17;

if (idade <= 11)

{

MessageBox.Show("Criança");

}

else if (idade >= 12 && idade <= 18)

{

MessageBox.Show("Adolecente");

}

else if (idade >= 19 && idade <= 60)

{

MessageBox.Show("Adulto");

}

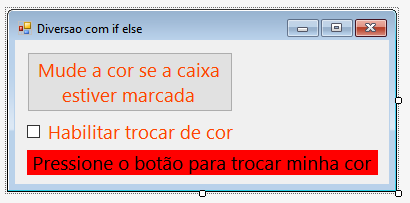
else

{

MessageBox.Show("Melhor idade");

}

Para demonstrar uma o uso, vamos construir um programa simples que permite ao usuário alterar a cor de fundo de um label.



chkTrocar

btnTroca

lblMsg

Para que esse programa funcione vamos seguir a seguinte lógica:

Se o CheckBox estiver marcado

Verifica se a cor do label lblMsg é vermelho se for trocar para azul e se não trocar para vermelho

Se o checkbox estiver desmarcado exibir uma mensagem informando ao usuário que ele precisa marcar o checkbox para habilitar a troca de cor.

Se o CheckBox estiver marcado

if (chkTrocar.Checked)

{

}

Verifica se a cor do label lblMsg é vermelho

if (lblMsg.BackColor == Color.Red)

{

se for trocar para azul

lblMsg.BackColor = Color.Blue;

}

else

{

se não trocar para vermelho  
lblMsg.BackColor = Color.Red;

}

Se o checkbox estiver desmarcado exibir uma mensagem informando ao usuário que ele precisa marcar o checkbox para habilitar a troca de cor.

else

{

MessageBox.Show("Marque a caixa para poder trocar a cor.");

}

Sendo assim o código completo fica assim:

private void button1\_Click(object sender, EventArgs e)

{

if (chkTrocar.Checked)

{

if (lblMsg.BackColor == Color.Red)

{

lblMsg.BackColor = Color.Blue;

}

else

{

lblMsg.BackColor = Color.Red;

}

}

else

{

MessageBox.Show("Marque a caixa para poder trocar a cor.");

}

}

## Exercícios

1. Realizar a leitura de quatro notas e calcular a média de um aluno classificando-o da seguinte forma:

|  |  |
| --- | --- |
| **Média** | **Classificação** |
| Maior que 0 e menor ou igual a 25 | 1 |
| Maior que 25 e menor ou igual a 50 | 2 |
| Maior que 50 e menor ou igual a 75 | 3 |
| Acima de 75 | 4 |

2. Aprimore o programa anterior informando se o aluno está aprovado ou reprovado. O aluno estará aprovado se a média calculada for superior a 50 e reprovado caso contrário, porém se a média for entre 45 e 49 informar que o aluno está de recuperação.

3. Efetuar a leitura do nome e sexo de uma pessoa e caso o sexo seja masculino apresentar uma a mensagem Sr. Fulano, caso o sexo seja feminino apresentar a mensagem Sra. Fulana. NOTA: Substituir Fulano/Fulana pelo nome informado pelo usuário.

4. Leia o valor de um número inteiro e multiplique por dois se o número informado for par multiplicar por 2 ou se o número informado for ímpar multiplicar por 5.

# Estrutura de Seleção Condicional

Se tivermos muitos testes condicionais a fazer precisaremos de muitos ifs...else encadeados o que tornaria nosso código difícil de ler, para solucionar questões como essa podemos contar com a estrutura de seleção switch...case.

char sexo = 'M';

switch (sexo)

{

case 'M':

MessageBox.Show("Masculino");

break;

case 'F':

MessageBox.Show("Feminino");

break;

default:

MessageBox.Show("Outro");

break;

}

}

Podemos tratar mais de um caso da seguinte forma:

string tecnologia = "mysql";

switch (tecnologia)

{

case "java":

case "c#":

case "cobol":

case "c++":

MessageBox.Show("Linguagem de Programação.");

break;

case "oracle":

case "postgree":

case "mysql":

case "ms sql server":

MessageBox.Show("Banco de Dados");

break;

default:

MessageBox.Show("Tecnologia desconhecida.");

break;

}

## Exercícios

1. Desenvolva um programa que receba um número de mês e mostra qual o nome do mês, se o usuário digitar um número inválido exibir “Mês Inválido”.

2. Solicite que o usuário informe o final da placa do seu veículo, e de acordo com o final informar o dia de rodizio, utilize a tabela abaixo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Dia | segunda | terça | quarta | quinta | sexta |
| Final da placa | 1 e 2 | 3 e 4 | 5 e 6 | 7 e 8 | 9 e 0 |

3. Faça um programa que baseado no final da placa de um veículo informe seu mês de licenciamento.

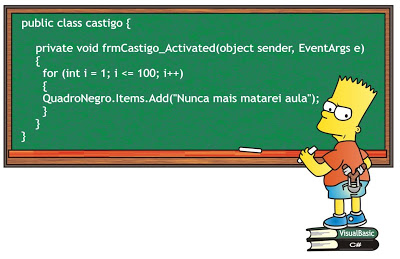
|  |  |
| --- | --- |
| **LICENCIAMENTO DE VEÍCULOS - 2017** | |
| **Final da Placa** | **Mês do Vencimento** |
| 1 | MARÇO |
| 2 | ABRIL |
| 3 | MAIO |
| 4 | JUNHO |
| 5 | JULHO |
| 6 | AGOSTO |
| 7 | SETEMBRO |
| 8 | OUTUBRO |
| 9 | NOVEMBRO |
| 0 | DEZEMBRO |

# Desafio II – Desafio dos Dados

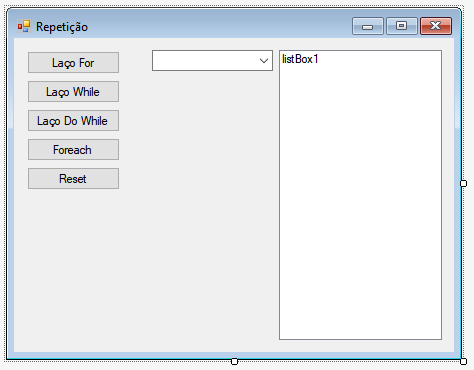
Você foi contratado por uma grande empresa de desenvolvimento de jogos eletrônicos, seu primeiro desafio é fazer um jogo que solicita o palpite do usuário, lança um dado e sorteia um dos lados do dado. O jogo deve informar se o usuário acertou ou errou o palpite.

OBS: Lembre-se que o dado possui 6 lados, isso significa que, o número a ser sorteado deve ficar entre 1 e 6.

# Estruturas de Repetição



As estruturas de repetição permitem que seja possível realizar ações repetitivas sem a necessidade de escrever o mesmo trecho de código várias e várias vezes.

Para demonstrar os laços de repetição existentes vamos utilizar o seguinte projeto:

btnFor

btnWhile

btnDoWhile

btnForeach

comboBox1

listBox1

btnReset

## 

## Laço de repetição for

O laço de repetição for possui a seguinte sintaxe:

**Teste lógico**

**Inicialização**

for (int i = 0; i < 11; i++) {

**Incremento**

}

Na inicialização declaramos uma variável do tipo inteiro e informamos a partir de qual número vamos começar a contagem de repetições.

No teste lógico, informamos o número máximo de repetições.

E no incremento informamos a quantidade de vezes que a variável de contador será incrementada.

Exemplos:

No exemplo a seguir o laço for repete por 10 vezes (de 0 até 10) aumentado o valor da variável i de 1 em 1 (i++) .

Dentro do laço for para cada repetição vamos adicionar aos items do listBox1 o valor atual da variável i.

private void btnFor\_Click(object sender, EventArgs e)

{

for (int i = 0; i < 11; i++)

{

listBox1.Items.Add(i);

}

}

O código a seguir serve para limpar todos os items do listBox1

private void btnReset\_Click(object sender, EventArgs e)

{

listBox1.Items.Clear();

}

No exemplo a seguir o laço for começa a contagem em 10, e vai repetir enquanto a variável i for maior ou igual a zero, subtraindo 1 do valor da variável a cada repetição.

Isso significa que vamos adicionar ao listBox1 uma sequência de 10 até 0.

private void btnFor\_Click(object sender, EventArgs e)

{

for (int i = 10; i >= 0; i--)

{

listBox1.Items.Add(i);

}

}

No exemplo abaixo será escrito a frase Linguagem C# 5 vezes no listBox.

private void btnFor\_Click(object sender, EventArgs e)

{

for (int i = 0; i < 6; i++)

{

listBox1.Items.Add("Linguagem C#");

}

}

## Laço de repetição while

O laço de repetição while possui a seguinte sintaxe:

**Inicialização**

int i = 0;

while (i <= 10) {

i++;

**Teste lógico**

**Incremento**

}

O laço while será executado enquanto a condição do teste lógico for verdadeira.

No exemplo abaixo o contador inicia em zero e a condição para as repetições pararem é que o contador seja maior que dez.

A cada nova repetição adicionamos o valor atual do contador nos itens da listaBox1 e aumentamos o valor do contador em mais 1.

private void btnWhile\_Click(object sender, EventArgs e)

{

int i = 0;

while (i <= 10)

{

listBox1.Items.Add(i);

i++;

}

}

Neste segundo exemplo o contador inicia em dez e as repetições vão parar quando o contador chegar a um número menor que zero. A cada nova repetição, adicionamos o valor atual do contador aos itens do listBox1 e subtraímos um do contador.

private void btnWhile\_Click(object sender, EventArgs e)

{

int x = 10;

while (x >= 0)

{

listBox1.Items.Add(x);

x--;

}

}

Nesse exemplo será escrito Programação C# no listBox1 cinco vezes.

private void btnWhile\_Click(object sender, EventArgs e)

{

int y = 0;

while (y < 6)

{

listBox1.Items.Add("Programação C#");

i++;

}

}

## Laço de repetição do... while

**Inicialização**

int i = 0;

do

**Incremento**

{

i++;

**Teste lógico**

} while (i <= 10);

No exemplo a seguir o contador inicia em zero e as repetições ocorrerão enquanto o contador seja menor que 10.

private void btnDoWhile\_Click(object sender, EventArgs e)

{

int i = 0;

do

{

listBox1.Items.Add(i);

i++;

} while (i <= 10);

}

Já neste exemplo o contador inicia em dez e a cada nova repetição adicionamos o valor atual do contato aos itens da listBox, subtraindo um do valor do contador.

private void btnDoWhile\_Click(object sender, EventArgs e)

{

int x = 10;

do

{

listBox1.Items.Add(x);

x--;

} while (x >= 0);

}

## Diferença entre while e do...while

Para observamos a diferença entre os laços while e do...while, vamos considerar os seguintes trechos de código:

int i = 10;

listBox1.Items.Add("while");

while (i < 5)

{

listBox1.Items.Add(i);

}

int y = 10;

listBox1.Items.Add("do...while");

do

{

listBox1.Items.Add(y);

} while (i < 5);

Veja que no primeiro exemplo a condição é falsa e as instruções do laço while não serão executadas.

Já no segundo exemplo primeiro é executada a instrução de adicionar o valor atual de y ao listBox1 e depois se verifica se a condição é falsa, sendo assim o laço do...while executa as instruções pelo menos uma vez.

# Laço de repetição foreach

O laço de repetição foreach pode percorrer uma lista recuperando os seus dados.

## Obtendo valores de um ArrayList

ArrayList modelos = new ArrayList();

modelos.Add("Amarok");

modelos.Add("Passat");

modelos.Add("CrossFox");

modelos.Add("Fox");

modelos.Add("Fusca");

modelos.Add("Gol");

modelos.Add("Golf");

modelos.Add("Jetta");

foreach (var modelo in modelos)

{

cboModelo.Items.Add(modelo);

}

## Exercícios

1. Elaborar um programa que solicite que que usuário digite um número inteiro e apresente todos os números pares entre 0 e o número que o usuário digitou. Lembre-se que um número é par se o resto da divisão desse número por dois for igual a zero.

2. Elaborar um programa que leia um número inteiro e apresente o valor de sua tabuada. Por exemplo para exibir a tabuada do 2 vc deve fazer algo parecido com o que segue abaixo:

"2 X " + i + " = " + 2 \* i

# Exercícios Complementares

* 1. Observe o código a seguir e indique os valores da variável number em cada uma das linhas:

// number vale 15  
int number = 15;

// number vale \_\_\_\_\_\_\_\_\_

number = number + 10;

// number vale \_\_\_\_\_\_\_\_\_

number++;

// number vale \_\_\_\_\_\_\_\_\_

number += 10;

// number vale \_\_\_\_\_\_\_\_\_

number = 10 \* 2;

// number vale \_\_\_\_\_\_\_\_\_

number = 12 - (42 / 7);

// number vale \_\_\_\_\_\_\_\_\_

number += 10;

// number vale \_\_\_\_\_\_\_\_\_

number \*= 3;

// number vale \_\_\_\_\_\_\_\_\_

number = 71 / 3;

// count vale 0

int count = 0;

// count vale\_\_\_\_\_\_\_\_\_\_\_\_\_

count++;

// count vale\_\_\_\_\_\_\_\_\_\_\_\_\_

count += 2;

// count vale\_\_\_\_\_\_\_\_\_\_\_\_\_

count--;

// result vale\_\_\_\_\_\_\_\_\_\_\_\_\_

string result = "ola";

// result vale\_\_\_\_\_\_\_\_\_\_\_\_\_

result += " de novo " + result;

MessageBox.Show(result);

// yesNo vale\_\_\_\_\_\_\_\_\_\_\_\_\_

bool yesNo = false;

// yesNo vale\_\_\_\_\_\_\_\_\_\_\_\_\_

bool outroBool = true;

// yesNo vale\_\_\_\_\_\_\_\_\_\_\_\_\_

yesNo = !outroBool;

* 1. Eis alguns laços. Escreva se cada um deles vai se repetir para sempre, ou se, eventualmente, ele terminará. Se o laço encerrar, quantas vezes ele se repetirá?

Laço #1

int count = 5;

while (count > 0)

{

count = count \* 3;  
count = count \* -1;

}

Laço #2

int i = 0;

while (i == 0)

{

i = i \* 3;

i = i \* -1;

}

Laço #3

int j = 2;

for (int i = 0; i < 100; i = i \* 2)

{

i = i \* 2;

while (j < 25)

{

j = j + 5;

}

}

Laço #4

while (true)

{

int i = 1;

}

Laço #5

int p = 2;

for (int q = 0; q < 32; q = q \* 2)

{

while (p < q)

{

p = p \* 2;

}

q = p - q;

}

Laço #6

for (int i = 0; true; i++)

{

i = i + 2;

}

# Orientação a objetos

O paradigma de Programação Orientada a Objetos foi criado para aproximar o mundo da programação do mundo real.

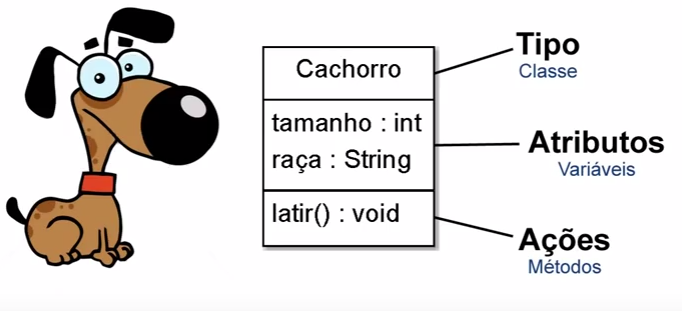
Hoje em dia a maior parte das linguagens de programação de alto nível utiliza o paradigma de Programação Orientada a Objetos.

## Abstração

Abstrato é tudo que não é concreto ou resulta de abstração. É o que só existe na ideia, no conceito, é o que possui alto grau de generalização, que opera unicamente com noções.

**Abstração** **é a ação ou efeito de abstrair**, ou seja, de isolar mentalmente um elemento ou de um todo, para considerar individualmente. Do latim “abstracione”, que significa “separação”.

Quando pensamos em objetos do mundo real, notamos que esses objetos possuem características e comportamentos.

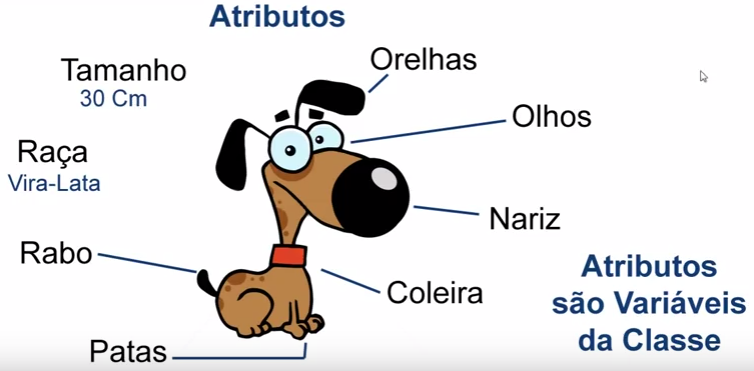


A exemplo disso temos a ilustração acima onde um objeto Cachorro, possui as características tamanho e raça e tem como um de seus comportamentos latir.

Tente por exemplo determinar características e comportamentos de outros objetos do mundo real como um Carro, um Computador ou uma Conta Bancária.

## Atributos

Os atributos são a forma de representação das características de um objeto, no exemplo de um cachorro podemos elencar alguns de seus atributos conforme podemos ver na seguinte ilustração:

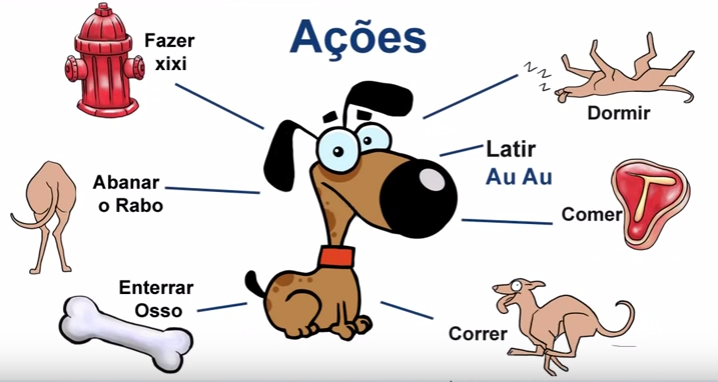


Conforme o contexto em que esse objeto está inserido alguns atributos abstraídos serão necessários e outros não, pode exemplo, vamos imaginar a abstração de um carro, um carro, possui placa, cor, marca, modelo, número de chassis, quantidade de portas, acessórios entre outras coisas, para a fábrica de carros faz sentido o atributo número de chassis, mas para um estacionamento, por exemplo esse atributo já não faz sentido.

Outra coisa importante é saber definir qual o tipo de dados, que poderão definir cada um dos atributos abstraídos.

## Métodos

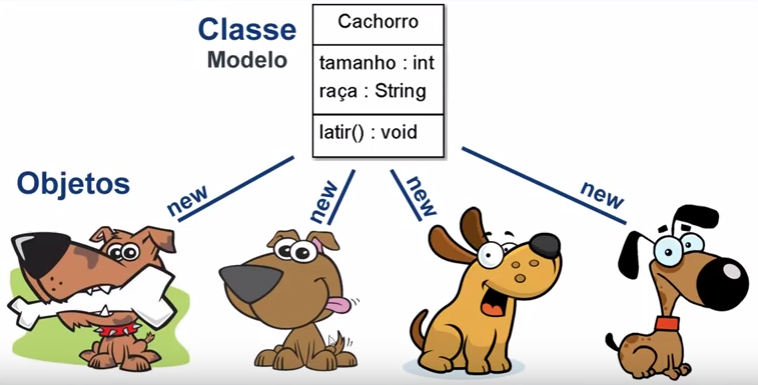
Os métodos são os comportamentos que os objetos possuem, coisas que os objetos podem fazer.



A exemplo disso conforme a ilustração acima, o cachorro possui diversos comportamentos. Cada um desses comportamentos é alguma coisa que o cachorro sabe fazer, e os comportamentos, podem eventualmente alterar os atributos, por exemplo, vamos imaginar uma abstração de Pessoa, sabendo que uma pessoa tem um nome e que uma pessoa tem uma idade, o comportamento Fazer Aniversário aumenta a idade em +1 ano.

## Classe

Uma classe é a forma de representarmos uma abstração, uma classe deve ser genérica, ou seja, é a partir das classes que criamos os objetos, assim sendo, os atributos não são inicializados com valores nas classes.



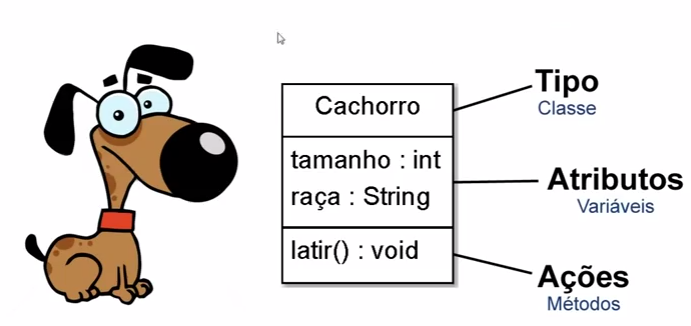
Pense em uma classe como uma pasta de um arquivo onde podemos preencher alguns dados:

Apenas quando criamos um objeto a partir de uma classe é que podemos preencher os dados desse objeto.



Ou seja, uma classe é uma abstração, por meio da qual podemos criar objetos.

Para exemplificar vamos criar a classe Cachorro:



Para adicionar uma nova classe ao Projeto:

1. Clique com o botão direito sobre o projeto
2. Vá até o sub menu Adicionar
3. Escolha a opção Classe...

Lembre-se que o nome de sua classe deve refletir o objeto que você deseja representar no mundo real e deve sempre iniciar com letra maiúscula.

Assim sendo a nossa classe Cachorro ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Abstracao

{

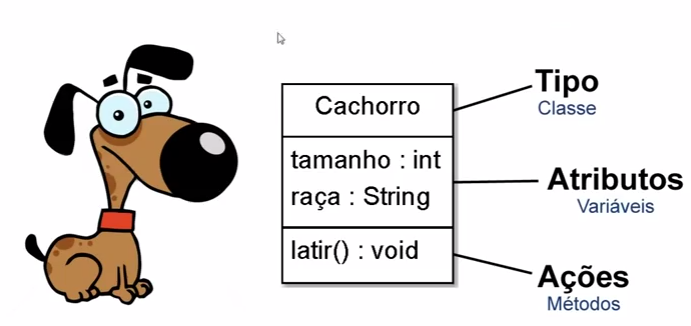
class Cachorro

{

}

}

Vamos adicionar os atributos (características) à nossa classe:



class Cachorro

{

// atributos

public int tamanho;

public string raca;

}

E por fim vamos acrescentar o método (comportamento) latir:

class Cachorro

{

// atributos

public int tamanho;

public string raca;

// métodos

public void Latir()

{

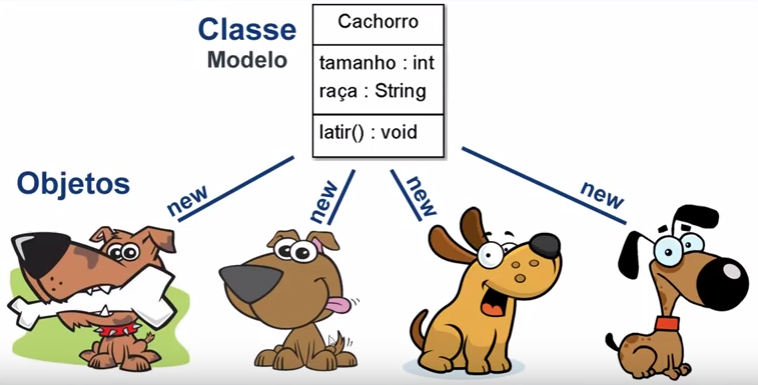
MessageBox.Show("Au au au");

}

}

Adicione um botão ao Form1 e clique nesse botão duas vezes:

Para que seja possível utilizar a classe cachorro devemos criar objetos do tipo Cachorro a partir da classe:



Para que isso seja feito devemos utilizar o operador new da seguinte forma:

Cachorro toto = new Cachorro();

Agora podemos dizer que temos uma variável do tipo Cachorro chamada toto, podemos também dizer que criamos uma instância de Cachorrou ou instanciamos um objeto do tipo Cachorro.

E então a partir daí podemos inicializar os atributos desse objeto chamado toto:

Cachorro toto = new Cachorro();

toto.raca = "Vira-latas";

toto.tamanho = 15;

Podemos também, agora chamar o método latir:

Cachorro toto = new Cachorro();

toto.raca = "Vira-latas";

toto.tamanho = 15;

toto.Latir();

Se precisarmos criar outro cachorro o código ficará da seguinte forma:

Cachorro toto = new Cachorro();

toto.raca = "Vira-latas";

toto.tamanho = 15;

toto.Latir();

Cachorro pitbull = new Cachorro();

pitbull.raca = "Pitbull";

pitbull.tamanho = 25;

pitbull.Latir();

Se houver necessidade de adicionarmos atributos ou métodos ao Cachorro, podemos fazer isso facilmente alterando apenas a classe Cachorro e a alteração vai se refletir para todos os objetos do tipo Cachorro que tivermos.

class Cachorro

{

// atributos

public int tamanho;

public string raca;

public string nome;

...

E após isso basta utilizar o novo atributo

Cachorro toto = new Cachorro();

toto.raca = "Vira-latas";

toto.tamanho = 15;

toto.nome = "Totó";

toto.Latir();

Cachorro pitbull = new Cachorro();

pitbull.raca = "Pitbull";

pitbull.nome = "Sadam";

pitbull.tamanho = 25;

pitbull.Latir();

Agora podemos alterar também o método Latir, fazendo com que seja indicado o nome do cachorro que latiu:

// método

public void Latir()

{

MessageBox.Show("O " + nome + " latiu: Au au au");

}

Vamos também acrescentar um atributo sexo, para que quando o sexo for fêmea apareça “A nome latiu...” e quando o cachorro for macho apareça “O nome latiu...”

class Cachorro

{

// atributos

public int tamanho;

public string raca;

public string nome;

public char sexo;

// método

public void Latir()

{

if (sexo == 'M')

{

MessageBox.Show("O " + nome + " latiu: Au au au");

}

else

{

MessageBox.Show("A " + nome + " latiu: Au au au");

}

}

}

`

Para utilizar o novo atributo e ver o novo método em funcionamento:

Cachorro pitbull = new Cachorro();

pitbull.raca = "Pitbull";

pitbull.nome = "Sadam";

pitbull.tamanho = 25;

pitbull.sexo = 'M';

pitbull.Latir();

Cachorro lessie = new Cachorro();

lessie.raca = "";

lessie.nome = "Lessie";

lessie.tamanho = 30;

lessie.sexo = 'F';

lessie.Latir();

## Exercícios

1. Você está desenvolvendo um software para um estacionamento, crie uma classe

Carro, somente com os atributos que fazem sentido para o estacionamento.  
Na classe carro implemente os métodos Andar (), Parar (), Abastecer (), Ligar () e Desligar ().  
Crie alguns Carros, inicialize seus atributos e execute seus métodos.

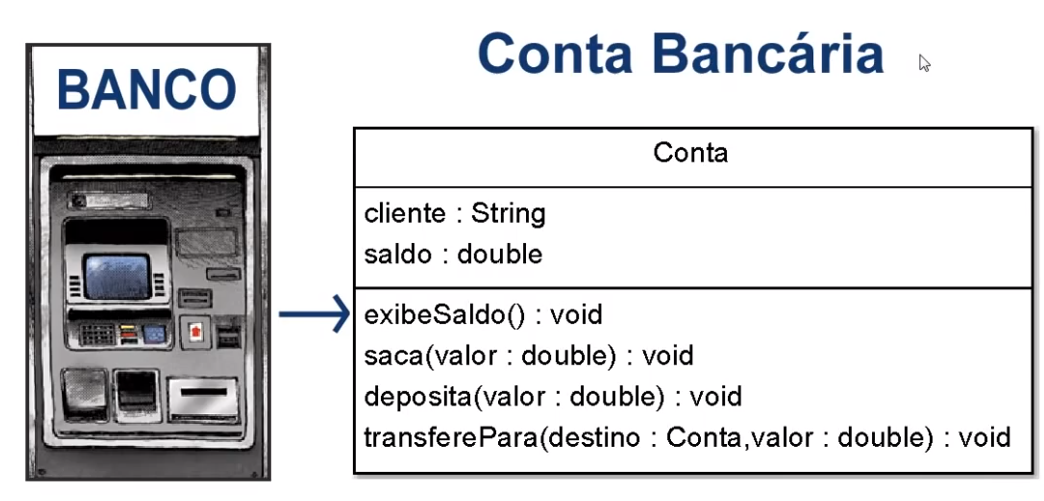
1. Crie uma classe Pessoa, os atributos de Pessoa são nome e idade. Crie um método FazerAniversário () que ao ser chamado aumenta a idade da pessoa em +1.  
   Crie alguns objetos do tipo Pessoa e faça com que façam alguns aniversários, mostre a idade das pessoas.
2. Mostre a diferença da idade das pessoas que você criou.
3. Crie uma classe Produto, um produto possui uma descrição, uma marca, um tipo e se é perecível ou não, além disso cada produto tem um valor.  
   Crie alguns Produtos, inicie seus atributos e mostre o total da compra.

# Métodos

Como já vimos anteriormente os métodos definem os comportamentos dos objetos.

Para exemplificarmos, vamos utilizar uma conta bancária, que possui um cliente titular e um saldo, com a conta bancária é possível sacar um valor, exibir o sando da conta, depositar um valor e transferir um valor para outra conta

## Métodos sem retorno



Todo método deve possuir as seguintes partes:

Modificadores

Tipo de Retorno

Nome do Método

Parâmetros

public void Saca(double valor)

{

// corpo do método

}

Os modificadores veremos adiante, a princípio, todo método deve possui o modificador public, o que permite que qualquer classe tenha acesso a esse método.

O tipo de retorno, informa o tipo de resultado que o método produz, nesse tópico o tipo de retorno será sempre void, pois estamos falando de métodos sem retorno.

O nome do método deve ser composto apenas de letras que compõe o alfabeto, e por convenção deve-se utilizar o PascalCasing.

Os parâmetros devem utilizar camelCasing, somente letras do alfabeto e o nome do(s) parâmetro(s) deve ser precedido de seu tipo de dados, se houver mais de um parâmetro separar por vírgula.

No corpo do método declaramos as instruções que o método deve executar, o corpo do método também pode ter suas próprias variáveis locais, que são válidas apenas em seu escopo (corpo do método).

Para exemplificar vamos criar a classe Conta, esta classe deve ser escrita da seguinte forma:

namespace MetodosSemRetorno

{

class Conta

{

// atributos

public string cliente;

public double saldo;

// métodos

}

}

Vamos implementar um primeiro método, na classe Conta, para exibir o saldo da conta:

public void ExibeSaldo()

{

MessageBox.Show("Seu saldo é de R$ " + saldo.ToString("N2"));

}

Para que tudo funcione a conta deve ser acessada por todos os métodos por isso vamos criar a conta na classe Form1:

namespace MetodosSemRetorno

{

public partial class Form1 : Form

{

Conta conta = new Conta();

...

E vamos inicializar os atributos da conta no evento de load do form:

private void Form1\_Load(object sender, EventArgs e)

{

conta.cliente = "Jorge";

conta.saldo = 10\_750.00;

}

Para testar o método ExibeSaldo vamos programar um evento em um botão:

private void button1\_Click(object sender, EventArgs e)

{

conta.ExibeSaldo();

}

Podemos melhorar o método exibindo o nome do cliente:

public void ExibeSaldo()

{

MessageBox.Show(cliente + " seu saldo é de R$ " + saldo.ToString("N2"));

}

Vamos também criar um método para depositar valores da conta:

public void Deposita(double valor)

{

this.saldo += valor;

}

O operador this faz referência ao objeto que chamou o método deposita, ou seja estamos dizendo que o valor deve ser acrescentado ao saldo do objeto que chamou o método Deposita.

Vamos criar um botão para testar o método Deposita:

private void button2\_Click(object sender, EventArgs e)

{

double valor;

if (!double.TryParse(txtValor.Text, out valor))

{

MessageBox.Show("Valor inválido para realizar a operação !");

}

else

{

conta.Deposita(valor);

MessageBox.Show("Depósito efetuado com sucesso !");

conta.ExibeSaldo();

}

}

Vamos permitir que seja possível também sacar valores da conta, no mundo real, sacar valores da conta implicaria em uma série de processos e procedimentos:

public void Saca(double valor)

{

// ver se tem autorização

// se o limite do dia não foi excedido

// se tem saldo na conta

// deduzir o saldo do valor do saque

// atualizar os registros do banco

// entregar o dinheiro

}

Como estamos implementando um exemplo didático vamos simplificar o método sacar da seguinte forma:

public void Saca(double valor)

{

this.saldo -= valor;

}

E para testar o método vamos programar um próximo botão:

private void button3\_Click(object sender, EventArgs e)

{

double valor;

if (!double.TryParse(txtValor.Text, out valor))

{

MessageBox.Show("Valor inválido para realizar a operação !");

}

else

{

conta.Saca(valor);

MessageBox.Show("Saque efetuado com sucesso !");

conta.ExibeSaldo();

}

}

Por fim vamos criar um método para transferir dinheiro entre duas contas.

O método TrasnferePara deve ser criado na classe conta e deve receber de parâmetro uma conta de destino e um valor a ser transferido:

public void TransferePara(Conta destino, double valor)

{

this.Saca(valor);

destino.Deposita(valor);

}

Teremos de criar uma segunta conta para podermos transferir dinheiro:

namespace MetodosSemRetorno

{

public partial class Form1 : Form

{

Conta conta = new Conta();

Conta conta2 = new Conta();  
 ...

Também vamos precisar inicializar os atributos da segunda conta:

private void Form1\_Load(object sender, EventArgs e)

{

conta.cliente = "Jorge";

conta.saldo = 10\_750.00;

conta2.cliente = "Ricardo";

conta2.saldo = 25\_000.00;

}

E para podermos testar o método vamos programar mais um botão:

private void button4\_Click(object sender, EventArgs e)

{

double valor;

if (!double.TryParse(txtValor.Text, out valor))

{

MessageBox.Show("Valor inválido para realizar a operação !");

}

else

{

conta.TransferePara(conta2, valor);

MessageBox.Show("Transferência concluída com sucesso !");

conta.ExibeSaldo();

conta2.ExibeSaldo();

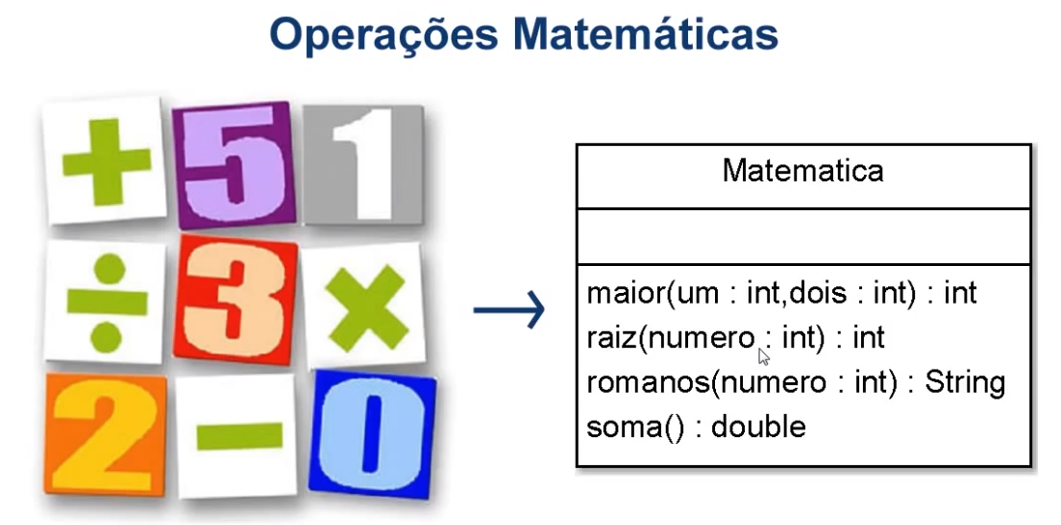
}

}

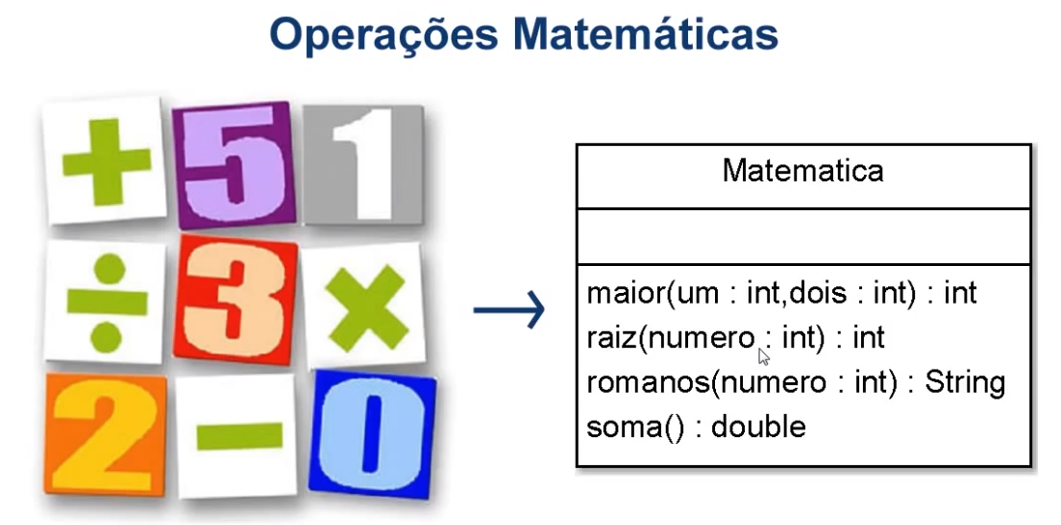
## Métodos com retorno

Ao contrário dos métodos sem retorno (void), os métodos com retorno devolvem para quem os chama um resultado, a esse resultado é dado o nome de retorno do método.

Para exemplificar vamos utilizar uma classe chamada Matematica, que deve possuir métodos para realizar operações matemáticas.



Vamos começar criando a nossa classe Matematica



namespace MetodosComRetorno

{

class Matematica

{

}

}

O primeiro método que vamos implementar é um método que recebe dois números como parâmetro e deve retornar o maior dos dois números:

public int Maior(int um, int dois)

{

if (um > dois)

{

return um;

}

else

{

return dois;

}

}

Para testar vamos fazer com que a classe Matematica seja visível para toda a classe Form1

namespace MetodosComRetorno

{

public partial class Form1 : Form

{

Matematica m = new Matematica();

...

Agora vamos testar o método Maior

private void button1\_Click(object sender, EventArgs e)

{

int x, y;

if (int.TryParse(textBox1.Text, out x)

&& int.TryParse(textBox2.Text, out y))

{

int maior = m.Maior(x, y);

MessageBox.Show("Maior: " + maior);

}

}

Vamos criar na classe Matematica mais um método, este deve somar dois números e retornar o resultado da soma

public int soma(int um, int dois)

{

return um + dois;

}

Testando o método Soma

private void button2\_Click(object sender, EventArgs e)

{

int x, y;

if (int.TryParse(textBox1.Text, out x) &&

int.TryParse(textBox2.Text, out y))

{

int soma = m.Soma(x, y);

MessageBox.Show("Soma: " + soma);

}

}

Exemplo de uso de métodos com retorno:

Imagine que temos o seguinte problema:

O programa deve receber quatro números, dois deles serão pares e dois serão ímpares.

Desafio: Obter a soma do maior número par e do maior número ímpar.

private void button3\_Click(object sender, EventArgs e)

{

int par1, par2, impar1, impar2;

int maiorPar = 0, maiorImpar = 0;

if (int.TryParse(textBox1.Text, out par1) && int.TryParse(textBox2.Text, out par2))

{

maiorPar = m.Maior(par1, par2);

}

if (int.TryParse(textBox3.Text, out impar1) && int.TryParse(textBox4.Text, out impar2))

{

maiorImpar = m.Maior(impar1, impar2);

}

int soma = m.Soma(maiorPar, maiorImpar);

string msg = "Maior Par:" + maiorPar + "\nMaior Ímpar: " + maiorImpar +

"\nSoma: " + soma;

MessageBox.Show(msg);

}

Outra forma de resolver o problema seria assim:

private void button3\_Click(object sender, EventArgs e)

{

int par1, par2, impar1, impar2;

if (int.TryParse(textBox1.Text, out par1) && int.TryParse(textBox2.Text, out par2) && int.TryParse(textBox3.Text, out impar1) && int.TryParse(textBox4.Text, out impar2))

{

int soma = m.Soma(m.Maior(par1, par2), m.Maior(impar1, impar2));

string msg = "Maior Par:" + m.Maior(par1, par2) + "\nMaior Ímpar: " + m.Maior(impar1, impar2) + "\nSoma: " + soma;

MessageBox.Show(msg);

}

}

Perceba que na linha abaixo, os retornos do método Maior servem de parâmetros para os método soma:

int soma = m.Soma(m.Maior(par1, par2), m.Maior(impar1, impar2));

## VarArgs – Argumentos variáveis para métodos

Em alguns casos existe a necessidade de passarmos mais de um argumento para os nossos métodos, porém, nem sempre sabemos antecipadamente a quantidade de argumentos que precisaremos receber.

Para solucionar estes casos podemos utilizar a técnica de argumentos variáveis

Para exemplificar, vamos criar um método soma na classe Matemática que pode receber uma quantidade de argumentos variáveis para somar

public int Soma(params int[] valores)

{

int total = 0;

foreach (var valor in valores)

{

total += valor;

}

return total;

}

E para testar:

private void button5\_Click(object sender, EventArgs e)

{

int x, y, z, w;

if (int.TryParse(textBox1.Text, out x) && int.TryParse(textBox2.Text, out y) && int.TryParse(textBox3.Text, out z) && int.TryParse(textBox4.Text, out w))

{

int total = m.Soma(x, y, z);

int total2 = m.Soma(x, y, z, w);

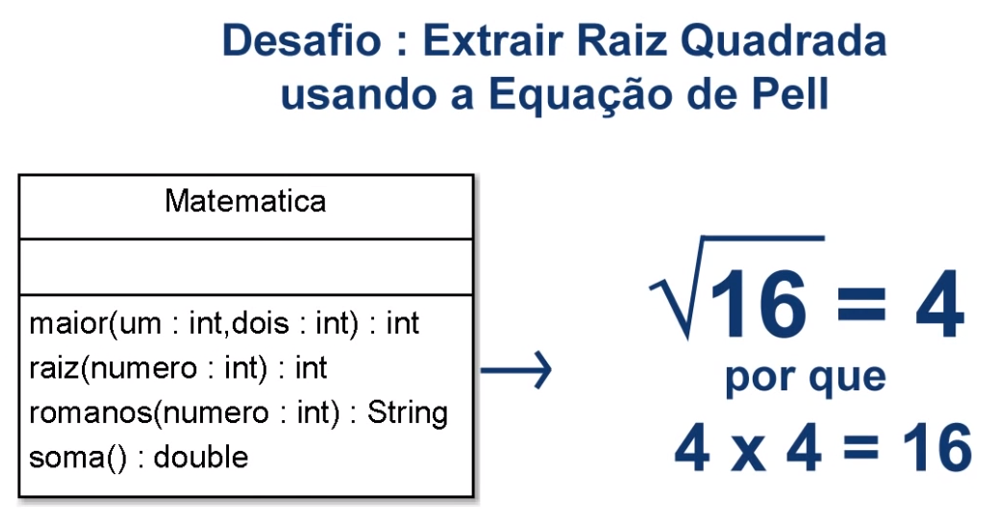
MessageBox.Show("Total 1: " + total);

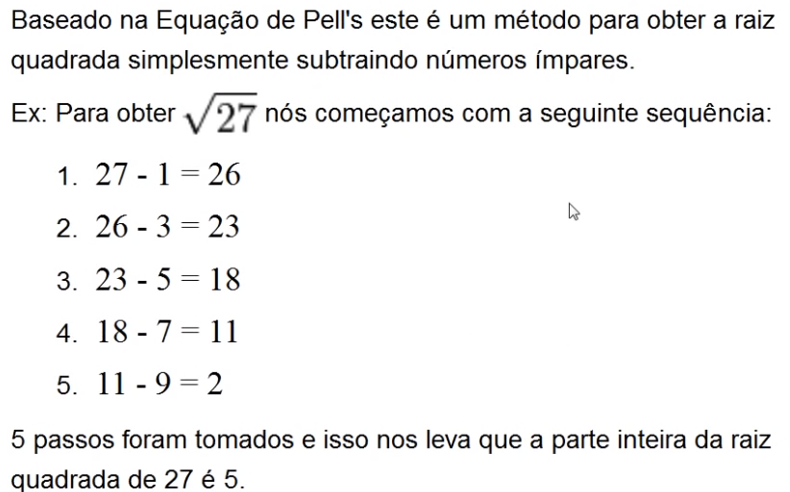
MessageBox.Show("Total 2: " + total2);

}

}

# Desafio: Extrair a raiz quadrada de um número utilizando a Equação de Pell





O método raiz deve ser escrito na classe Matematica e deve ser parecido com algo assim:

public int Raiz(int numero)

{

}

## Métodos sobrecarregados

Os métodos sobrecarregados possuem o mesmo nome mas tem assinaturas diferentes.

Para exemplificar, na classe Matematica vamos criar um método para calcular a média de dois números

public double Media(int x, int y)

{

Console.WriteLine("media(int x, int y)");

return (x + y) / 2;

}

Para testar

private void button6\_Click(object sender, EventArgs e)

{

int x, y;

if (int.TryParse(textBox1.Text, out x) && int.TryParse(textBox2.Text, out y))

{

double media = m.Media(x, y);

MessageBox.Show("Media: " + media);

}

}

Agora vamos criar um método Media, na classe Matematica que recebe duas strings no lugar dos inteiros

public double Media(string x, string y)

{

Console.WriteLine("media(string x, string y)");

int a, b;

double media = 0;

if (int.TryParse(x, out a) && int.TryParse(y, out b))

{

media = (a + b) / 2;

}

return media;

}

E para testar:

private void button6\_Click(object sender, EventArgs e)

{

media = m.Media(textBox1.Text, textBox2.Text);

MessageBox.Show("Media: " + media);

}

Dessa vez o método que será chamado vai ser o que recebe strings de parâmetro, isso acontece por que a CLR é inteligente o suficiente para determinar qual método deve ser chamado.

Podemos ainda fazer o seguinte:

public double Media(params int[] numeros)

{

return this.Soma(numeros) / numeros.Length;

}

Veja que o método public int Soma(params int[] valores) {... está sendo utilizado para somar todos os números e depois estamos dividindo o resultado da soma pela quantidade de números que está sendo passada, calculando assim a média de todos os argumentos.

# Namespaces

O grande problema que surge com os sistemas grandes é a organização de todas as suas classes. Para evitar que o sistema fique caótico, podemos agrupar as classes por características comuns e dar um nome para cada um desses grupos. Isto é, agruparíamos um conjunto de classes em um espaço em comum e lhe daríamos um nome, como por exemplo Senai.Banco.Investimentos. Esse espaço definido por um nome é chamado de **namespace**.

Segundo a convenção de nomes adotada pela Microsoft (<http://msdn.microsoft.com/en-us/library/893ke618.aspx>), os namespaces devem ter a forma: **NomeDaEmpresa.NomeDoProjeto.ModuloDoSistema**.

Para criarmos namespaces no projeto basta criarmos pastas no projeto e criar as classes dentro dessas pastas.

O namespace padrão tem sempre o nome do projeto.

# Modificadores de Acesso e Encapsulamento

Algumas coisas do mundo real, não podem ter valores negativos, como a quantidade de banheiro em uma casa, a velocidade de um avião ou o peso de um alteres.

Os modificadores de acesso são utilizados para que seja possível implementar um dos pilares da Orientação à Objetos, o encapsulamento.

O encapsulamento cria um campo de força em volta das nossas variáveis e nos permite realizar algumas validações antes de inicializarmos as variáveis.

Até o momento todos os atributos de nossas classes podem ser acessados diretamente, para exemplificar vamos criar uma classe Conta

namespace Acessores

{

class Conta

{

public string titular;

public double saldo;

}

}

Podemos inicializar os atributos da classe Conta de duas formas:

private void button1\_Click(object sender, EventArgs e)

{

Conta conta = new Conta();

conta.saldo = 1000;

conta.titular = "Jorge";

}

Ou

private void button1\_Click(object sender, EventArgs e)

{

Conta conta = new Conta() { titular = "Jorge", saldo = 1000 };

}

De qualquer forma nada impede que alguém acesse diretamente o atributo saldo por exemplo e faça o seguinte:

private void button1\_Click(object sender, EventArgs e)

{

Conta conta = new Conta() { titular = "Jorge", saldo = -120 };

}

Supondo que o cliente não tenha um limite o banco estaria em prejuízo.

Agora imagine isso se aplicado à velocidade ou altitude de um avião por exemplo ou até mesmo ao número da conta, que não poderia ser negativo.

Por isso é uma má prática, permitir o acesso direto aos atributos de uma classe, para determinar o nível de acesso de um atributo utilizaremos os assessores ou modificadores de acesso:

Alguns modificadores de acesso ou assessores são:

public o método ou atributo pode ser acessado diretamente por qualquer classe

private. o acesso é limitado apenas à própria classe

Então para resolver o problema dos atributos da conta vamos torna-los privados, assim ninguém mais terá acesso direto à esses atributos:

namespace Acessores

{

class Conta

{

private string titular;

private double saldo;

}

}

Ao fazer isso o código abaixo para de funcionar, isso por que proibimos o acesso direto aos atributos:

private void button1\_Click(object sender, EventArgs e)

{

Conta conta = new Conta() { titular = "Jorge", saldo = 1000 };

}

Agora devemos encapsular esse acesso, e para fazer isso utilizaremos propriedades, nessas propriedades vamos definir os métodos get e set, sendo que os métodos get ou is, servem para recuperar os valores dos atributos e os métodos set servem para inicializar os valores das variáveis.

Em resumo o encapsulamento serve para garantir que os atributos seja inicializados da forma mais adequada.

Lembre-se:

get serve para dizer ao C# como recuperar o valor de um atributo da classe

set serve para dizer ao C# como inicializar um atributo de uma classe

Assim sendo vamos criar uma propriedade para o titular da conta:

namespace Acessores

{

class Conta

{

// atributos

private string titular;

private double saldo;

public string Titular

{

get

{

return this.titular;

}

set

{

this.titular = value;

}

}

}

}

Também vamos criar uma propriedade para o saldo da conta

namespace Acessores

{

class Conta

{

// atributos

private string titular;

private double saldo;

public string Titular

{

get

{

return this.titular;

}

set

{

this.titular = value;

}

}

public double Saldo

{

get

{

return this.saldo;

}

set

{

this.saldo = value;

}

}

}

}

Agora para criar o nosso objeto conta faremos da seguinte forma:

Vamos supor que a conta tem um número e que esse número não pode ser negativo, vamos criar o atributo número:

class Conta

{

// atributos

private string titular;

private double saldo;

private int numero;

...

Agora vamos criar uma propriedade para o número da conta

public int Numero

{

get

{

return this.numero;

}

set

{

if (value > 0)

{

this.numero = value;

}

else

{

MessageBox.Show("O número da conta precisa ser positivo");

}

}

}

Outra forma de escrever as propriedades seria utilizando as propriedades nomeadas:

namespace Acessores

{

class Conta

{

// atributos

private string titular;

private double saldo;

private int numero;

public string Titular

{

get => titular;

set => titular = value;

}

public double Saldo

{

get => saldo;

set => saldo = value;

}

public int Numero

{

get => numero;

set => numero = value;

}

}

}

Agora podemos instanciar a conta da seguinte forma:

Conta conta = new Conta() { Numero = 1, Titular = "Jorge", Saldo = 1000 };

O que acontece se você tentar fazer o seguinte ?

Conta conta = new Conta() { Numero = -9, Titular = "Jorge", Saldo = 1000 };

Existe uma convenção para os nomes das propriedades, estas devem ser escritas utilizando PascalCasing.

Para saber mais:

Existem outros modificadores de acesso:

protected o acesso é limitado a classe as classes que herdam dessa classe

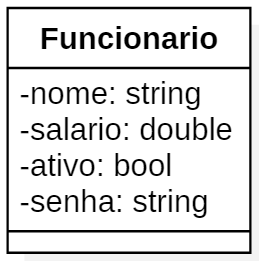
internal o acesso é limitado ao namespace atual

A sua documentação pode ser encontrada em:   
<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/access-modifiers>

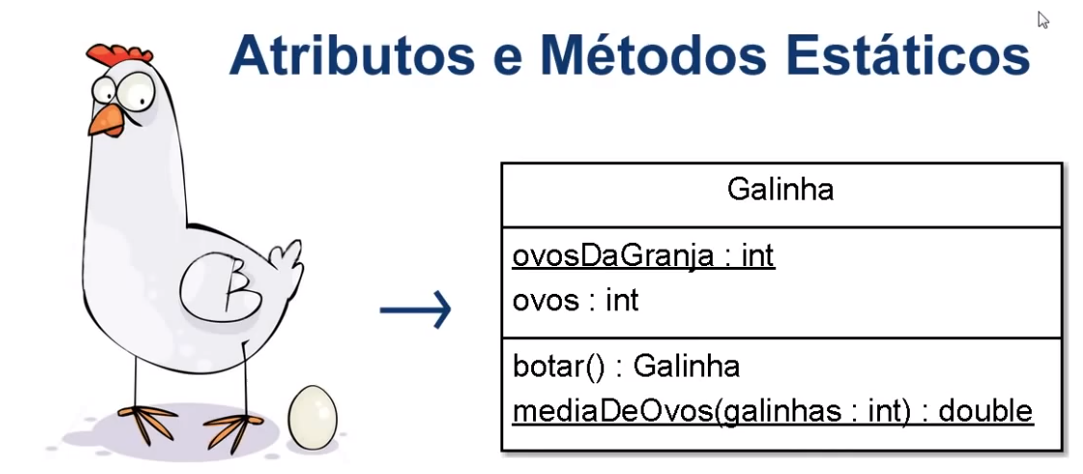
## Exercício

1. Modele a classe Funcionario, utilizando propriedades, depois utilize o evento de load do form para instanciar um Funcionário, inicializar suas propriedades.



1. Mostre todas as propriedades do funcionário em labels no form, também utilizando as propriedades.

# Variáveis e métodos estáticos



Os atributos que trabalhamos até o momento são chamadas de variáveis locais ou de instância, pois alteram apenas a instancia de objeto a qual pertencem.

Imagine um carro que tenha um atributo combustível. Cada carro sabe apenas informações sobre si próprio, por exemplo, quanto combustível ainda há no seu tanque. Porém a fábrica dos carros sabe quantos carros foram fabricados e sabe o chassi de cada carro.

Para tornar esse exemplo factível, precisaremos de um tipo de variável que pode ser acessada globalmente, ou seja que pertence a classe e não a instância de objeto.

Para nosso exemplo, vamos imaginar o controle de uma granja, e queremos saber quantos ovos cada galinha botou e o total de ovos que existem na granja.

Crie um projeto novo chamado Estatico e crie nele uma classe chamada Galinha:

namespace Estatico

{

class Galinha

{

private int ovos;

public int Ovos

{

get => ovos;

set => ovos = value;

}

public void Botar()

{

this.ovos++;

}

}

}

Cada vez que uma galinha bota um ovo a quantidade de ovos pertencente a galinha aumenta em mais um.

Para testar:

private void button1\_Click(object sender, EventArgs e)

{

Galinha carijo = new Galinha() { Ovos = 1 };

MessageBox.Show("Ovos: " + carijo.Ovos);

carijo.Botar();

carijo.Botar();

carijo.Botar();

MessageBox.Show("Ovos: " + carijo.Ovos);

}

Podemos fazer uma alteração no método botar:

public Galinha Botar()

{

this.ovos++;

return this;

}

Assim podemos fazer o seguinte:

private void button1\_Click(object sender, EventArgs e)

{

Galinha carijo = new Galinha() { Ovos = 1 };

MessageBox.Show("Ovos: " + carijo.Ovos);

carijo.Botar().Botar().Botar();

MessageBox.Show("Ovos: " + carijo.Ovos);

}

Sabemos quantos ovos cada galinha botou, mas e se, precisássemos saber quantos ovos há na granja? Para isso precisaremos de uma variável global, que pertença a classe galinha e não ao objeto galinha.

Para isso vamos criar um atributo na classe galinha e utilizar o modificador static

class Galinha

{

public static int OvosDaGranja;

...

O modificador final faz com que a variável OvosDaGranja seja global ou seja pertença a classe Galinha e não a um objeto galinha.

Assim sendo podemos modificar o método Botar, fazendo com que quando qualquer galinha botar um ovo a quantidade de ovos da granja seja aumentada.

public Galinha Botar()

{

this.ovos++;

Galinha.OvosDaGranja++;

return this;

}

Podemos testar fazendo o seguinte

private void button1\_Click(object sender, EventArgs e)

{

Galinha carijo = new Galinha() { Ovos = 0 };

MessageBox.Show("Ovos Carijo: " + carijo.Ovos);

carijo.Botar().Botar().Botar();

MessageBox.Show("Ovos Carijo: " + carijo.Ovos);

Galinha penosa = new Galinha() { Ovos = 0 };

MessageBox.Show("Ovos Penosa: " + penosa.Ovos);

penosa.Botar().Botar();

MessageBox.Show("Ovos Penosa: " + penosa.Ovos);

MessageBox.Show("Ovos da Granja: " + Galinha.OvosDaGranja);

}

Os métodos também podem ser estáticos, pertencendo assim a classe, para exemplificar, vamos criar um método que calcule a média de ovos da granja

public static double MediaDeOvos(int galinhas)

{

return Galinha.OvosDaGranja / galinhas;

}

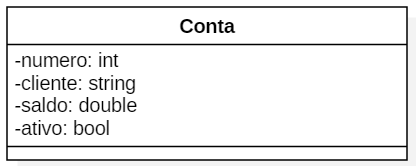
Para testar

double mediaDeOvos = Galinha.MediaDeOvos(2);

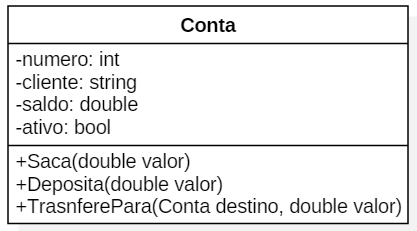
MessageBox.Show("Média: " + mediaDeOvos);

## Exercícios

1. Você foi contratado para criar um protótipo de uma aplicação bancária, sua tarefa será criar uma classe para representar uma conta bancária, a conta bancária possui um titular, um número e um saldo, também deve ser possível saber se a conta está ativa ou não. Você deverá criar a classe conta e implementar suas propriedades, a princípio não serão necessárias validações.



1. Agora que a conta está pronta, implemente os métodos Saca, Deposita e TransferePara



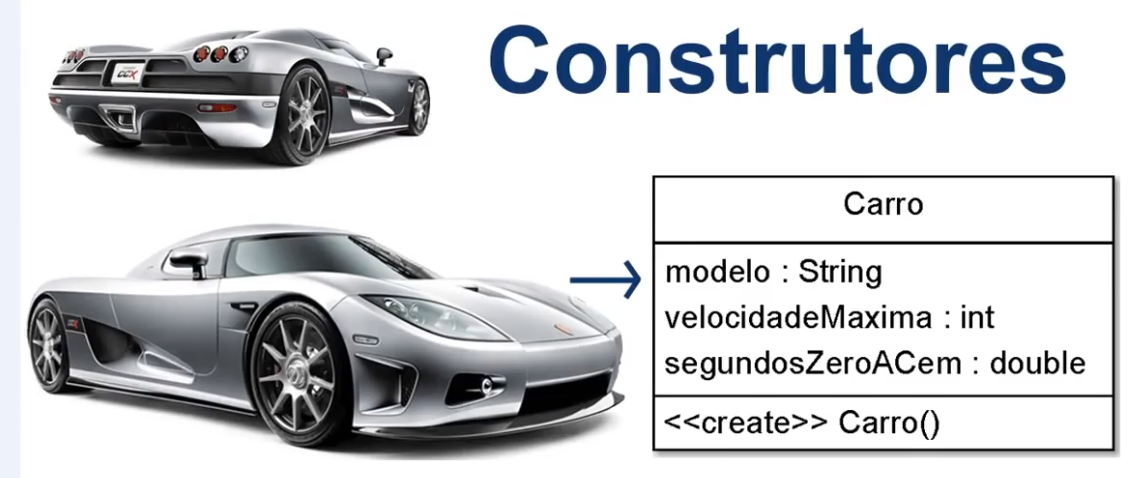
1. É preciso manter um registro de todo a dinheiro existente nos cofres do banco, sendo assim você deve implementar uma variável global, que, registre quanto há de dinheiro no cofre do banco, esse valor inicialmente será de R$ 1000.00, e cada vez que alguém sacar dinheiro de uma conta o valor deve ser deduzido do cofre do banco, e toda vez que alguém depositar algum valor esse valor deve ser acrescentado ao cofre do banco.

# Construtores

Os construtores, como o próprio nome já diz, nos ajudam a construir os nossos objetos.

Os construtores são blocos de comandos que você pode utilizar para inicializar os atributos de seus objetos.

Todas as classes possuem construtores, mesmo se você não declarar um.



Apesar de se parecer com um método o construtor não é um método.

Todo construtor sempre terá o mesmo nome da classe.

Vamos criar um projeto chamado Construtores e nesse projeto vamos criar uma classe chamada Carro.

namespace Construtores

{

class Carro

{

private string modelo;

private int velocidadeMaxima;

private double segundoZeroACem;

public string Modelo

{

get => modelo;

set => modelo = value;

}

public int VelocidadeMaxima

{

get => velocidadeMaxima;

set => velocidadeMaxima = value;

}

public double SegundosZeroACem

{

get => segundosZeroACem;

set => segundosZeroACem = value;

}

}

}

Agora podemos instanciar um carro:

private void Form1\_Load(object sender, EventArgs e)

{

**Construtor**

Carro ferrari = new Carro()

{

Modelo = "Ferrari Enzo",

SegundosZeroACem = 3.2,

VelocidadeMaxima = 349

};

}

Embora não tenhamos declarado um construtor explicitamente a CLR é capaz de criar um construtor para a classe Carro

class Carro

{

private string modelo;

private int velocidadeMaxima;

private double segundosZeroACem;

**Construtor**

public Carro()

{

}

...

Porém podemos escrever um construtor para inicializar os atributos do nosso carro

public Carro(string modelo, int velocidadeMaxima, double segundosZeroACem)

{

this.modelo = modelo;

this.velocidadeMaxima = velocidadeMaxima;

this.segundosZeroACem = segundosZeroACem;

}

Ao realizar essa alteração o trecho a seguir acusará um erro

private void Form1\_Load(object sender, EventArgs e)

{

Carro ferrari = new Carro()

...

Isso ocorre por que agora somos obrigados a informar os dados do carro ao instanciarmos e o construtor padrão não é mais válido, para corrigir o problema podemos manter dois construtores na classe Carro

class Carro

{

private string modelo;

private int velocidadeMaxima;

private double segundosZeroACem;

public Carro() { }

public Carro(string modelo, int velocidadeMaxima, double segundosZeroACem)

{

this.modelo = modelo;

this.velocidadeMaxima = velocidadeMaxima;

this.segundosZeroACem = segundosZeroACem;

}

...

E para utilizar o novo construtor podemos instanciar um carro da seguinte forma

Carro koenigsegg = new Carro("Koenigsegg CCXR", 430, 2.3);

Perceba que se retirarmos o construtor padrão será obrigatório a todos que instanciarem um carro dizer qual seu modelo, sua velocidade máxima e quantos segundo o carro leva para atingir a velocidade de 100 Quilômetros por hora.

## Exercícios

1. Crie um construtor padrão para a classe Conta
2. Crie também um construtor para a classe Conta do seu protótipo de banco com todos os atributos.
3. Utilize o novo construtor para instanciar as Contas.

# Composição

A maioria dos objetos do mundo real são compostos de outros objetos, por exemplo um computador é composto de Teclado, Mouse, Monitor entre outros, um pedido é composto por itens, um classe é composta por alunos, um time é composto por jogadores e um carro, entre muitas outras coisas é composto também por um motor.



A composição é um relacionamento do tipo **tem um**, o carro tem um motor.

Para exemplificar, vamos criar uma classe Motor para o carro:

class Motor

{

private string tipo;

private int potencia;

public Motor() { }

public Motor(string tipo, int potencia)

{

this.tipo = tipo;

this.potencia = potencia;

}

public string Tipo

{

get => tipo;

set => tipo = value;

}

public int Potencia

{

get => potencia;

set => potencia = value;

}

}

Agora vamos alterar o carro para que ele receba um motor

class Carro

{

private string modelo;

private int velocidadeMaxima;

private double segundosZeroACem;

private Motor motor;

...

E vamos criar a propriedade para o motor

public Motor Motor

{

get => motor;

set => motor = value;

}

Agora podemos instanciar um motor e atribuir a instancia a um motor de um carro

Motor v12 = new Motor("V12", 660);

Carro ferrari = new Carro()

{

Modelo = "Ferrari Enzo",

SegundosZeroACem = 3.2,

VelocidadeMaxima = 349,

Motor = v12

};

Também podemos fazer o mesmo da seguinte forma

Motor v8 = new Motor("V8", 1018);

Carro koenigsegg = new Carro("Koenigsegg CCXR", 430, 2.3);

koenigsegg.Motor = v8;

Vamos alterar o construtor do carro para que receba uma instancia de motor

public Carro(string modelo, int velocidadeMaxima, double segundosZeroACem, Motor motor)

{

this.modelo = modelo;

this.velocidadeMaxima = velocidadeMaxima;

this.segundosZeroACem = segundosZeroACem;

this.motor = motor;

}

E para utilizar o novo construtor vamos fazer da seguinte forma

Motor w16 = new Motor("W16", 1200);

Carro bugatti = new Carro("Bugatti Veyron", 430, 2.2, w16);

Ou podemos instanciar diretamente o motor

Carro bugatti = new Carro("Bugatti Veyron", 430, 2.2, new Motor("W16", 1200));

## Acessando os atributos da composição

lblModelo.Text = bugatti.Modelo;

lblVelocidadeMaxima.Text = bugatti.VelocidadeMaxima.ToString();

lblSegundosZeroACem.Text = bugatti.SegundosZeroACem.ToString();

lblMotor.Text = bugatti.Motor.Tipo;

lblPotencia.Text = bugatti.Motor.Potencia.ToString();

lblModelo.Text = "Modelo: " + bugatti.Modelo;

lblVelocidadeMaxima.Text = "Velocidade Máxima: " + bugatti.VelocidadeMaxima.ToString();

lblSegundosZeroACem.Text = "Segundos de Zero a Cem: " + bugatti.SegundosZeroACem.ToString();

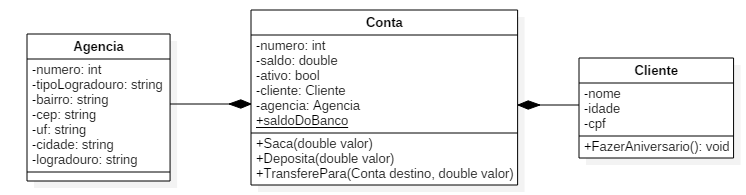
lblMotor.Text = "Motor: " + bugatti.Motor.Tipo;

lblPotencia.Text = "Potência: " + bugatti.Motor.Potencia.ToString();

## Exercícios

1. Na aplicação bancária, crie a classe Cliente
2. Crie também a classe Agencia
3. Altere a classe Conta para que seja composta de Cliente e Agencia, crie propriedades para Cliente e Agencia e altere o construtor da classe Conta para receber um cliente e uma agencia.
4. Instancie uma conta bancária, que tenha um cliente e uma agência associados e mostre todos os dados da conta em labels no load do formulário

OBS: Segue abaixo um diagrama de classes contendo os atributos de cada classe que você deverá criar.



# Enumerações

As enumerações são coleções de dados.

Uma enumeração de dados é útil quando precisamos criar estruturas que serão pouco alteradas ao longo do desenvolvimento do projeto.

Podemos declarar uma enum da seguinte forma:

public enum Sexo

{

Masculino, Feminino, Outro

}

Para utilizar podemos criar um propriedade com o tipo da enum

private Sexo sexo;

public Sexo Sexo

{

get { return sexo; }

set { sexo = value; }

}

Neste caso podemos também utilizar a enum nos construtores

public Cliente(string nome, int idade, string cpf, Sexo sexo)

{

this.nome = nome;

this.idade = idade;

this.cpf = cpf;

this.sexo = sexo;

}

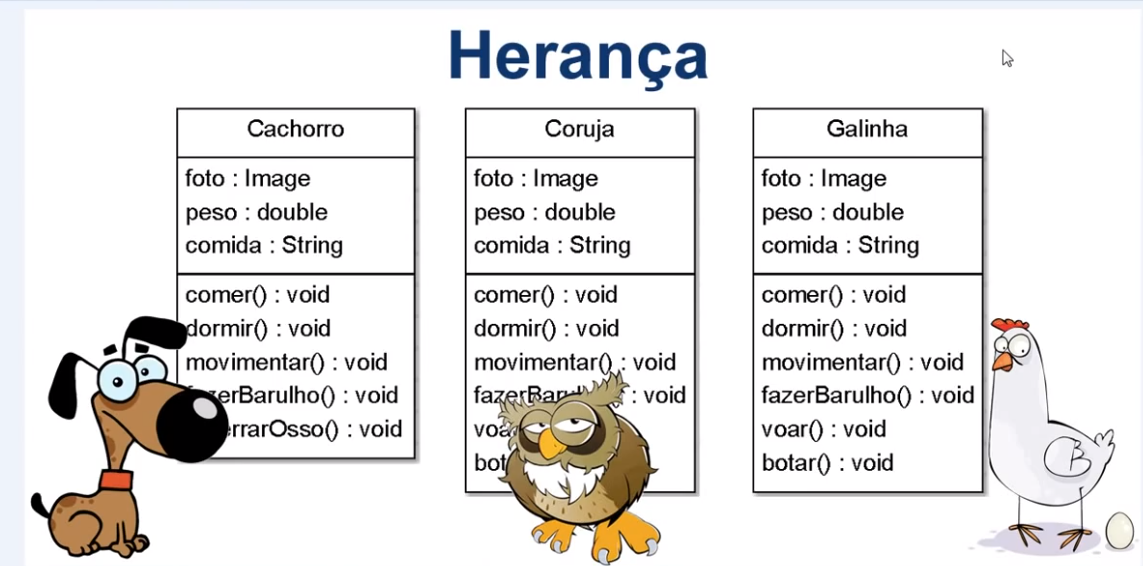
Na instancia do objeto podemos fazer assim

cliente = new Cliente("Jorge", 29, "36421147844", Sexo.Masculino);

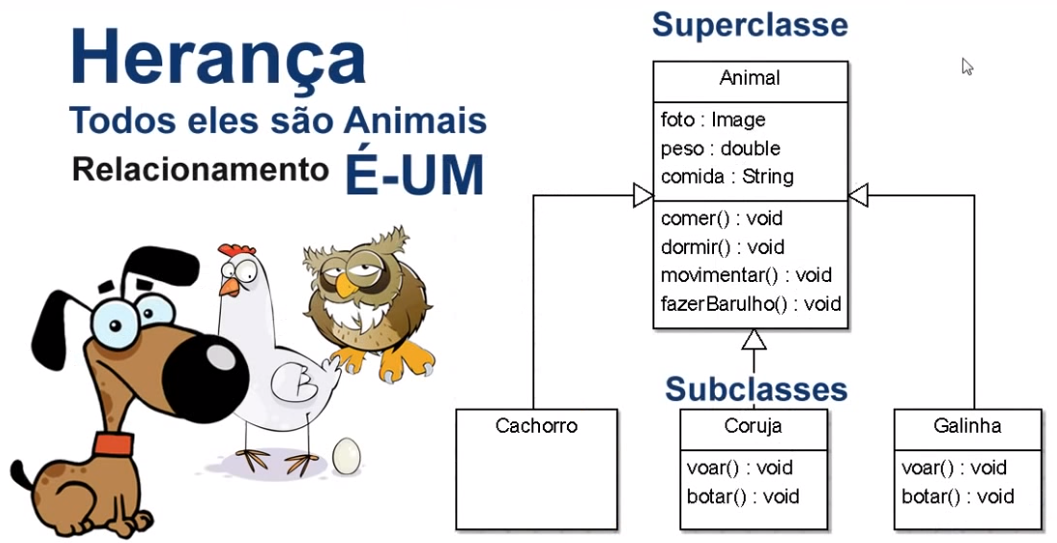
Se quisermos acessar a enum podemos fazer assim

string sexo = cliente.Sexo.ToString();

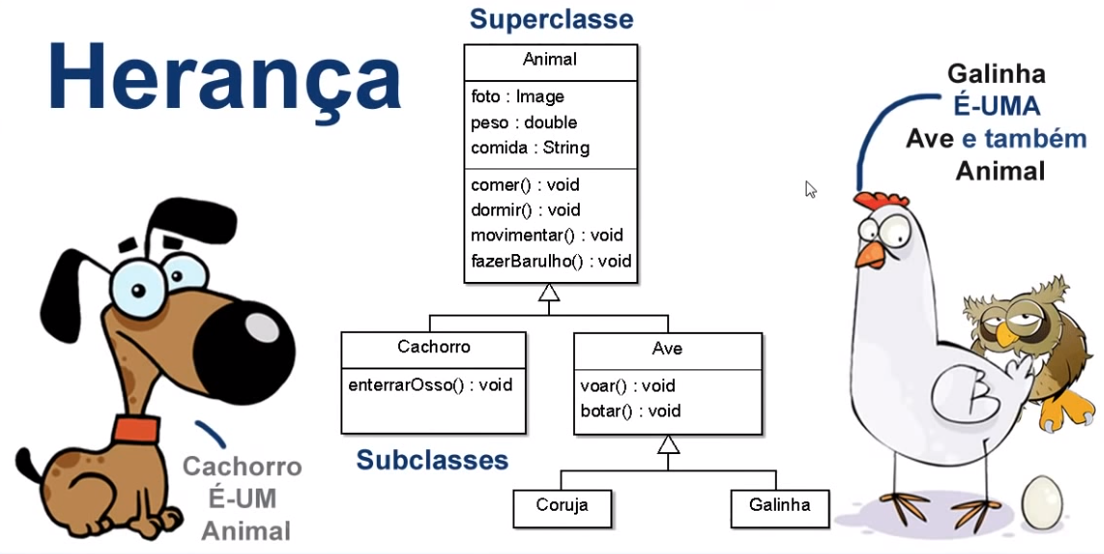
# Herança



A herança é utilizada quando temos abstrações que compartilham atributos e/ou métodos.



A herança pode ocorrer em várias classes também



Podemos criar a classe Animal e fazer com que as classes de tipo de animal herdem dela

class Animal

{

private double peso;

public double Peso

{

get { return peso; }

set { peso = value; }

}

private string comida;

public string Comida

{

get { return comida; }

set { comida = value; }

}

public void Dormir()

{

MessageBox.Show("Dormiu");

}

public void FazerBarulho()

{

MessageBox.Show("Fez barulho !");

}

}

Para dizer que uma classe herda de outra utilizamos o operador :

class Cachorro : Animal

{

}

class Galinha : Animal

{

}

Agora o cachorro e a galinha podem fazer uso das propriedades e dos métodos de Animal

Cachorro toto = new Cachorro() { Peso = 20, Comida = "Carne" };

toto.FazerBarulho();

toto.Dormir();

Galinha penosa = new Galinha() { Peso = 5, Comida = "Milho" };

penosa.FazerBarulho();

toto.Dormir();

A herança é um relacionamento do tipo é um, para saber se um tipo pode herdar de outro tipo podemos fazer o “teste do é um”:

1. Quadrado é uma Forma ?
2. Bebida é uma Cerveja ?
3. Forno é uma Cozinha ?
4. Funcionário é uma Pessoa ?
5. Suco é uma Bebida ?
6. Metal é um Titânio ?
7. OvoFrito é um Alimento ?
8. Ferrari é um Motor ?

## Herança e construtores

Podemos utilizar construtores nas classes pai, por exemplo:

public Animal(double peso, string comida)

{

this.peso = peso;

this.comida = comida;

}

Porém ao fazer isso, você terá de acrescentar um construtor nas classes que herdam de Animal

public Cachorro(double peso, string comida) : base(peso, comida)

{

}

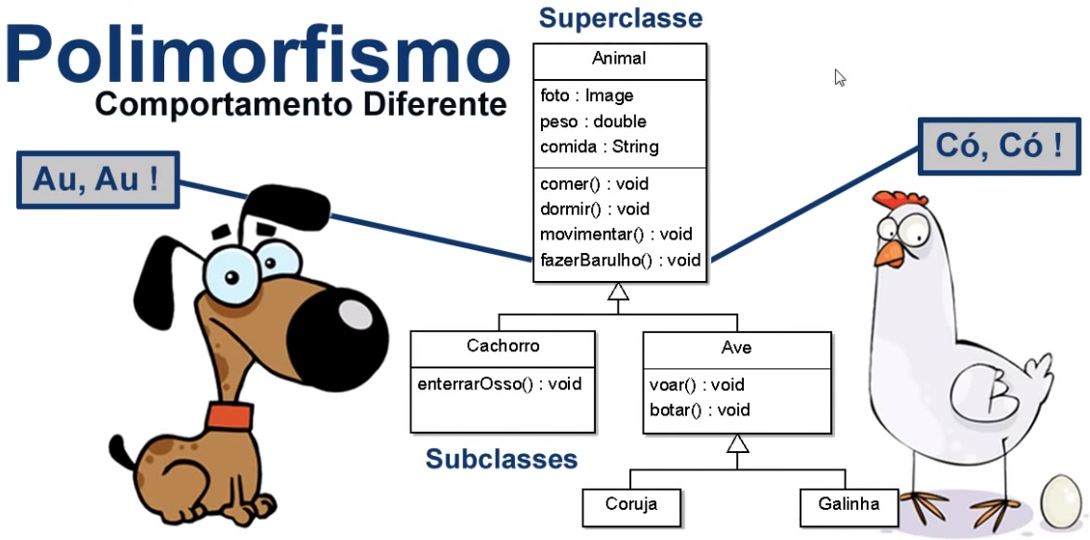
O trecho base(peso, comida) chama o construtor da classe pai (classe Animal) passando para ele os valores de peso e comida.

Isso deve ser feito para que a classe Animal consiga inicializar os atributos peso e comida.

# Polimorfismo

Polimorfismo é a capacidade dos objetos de assumirem múltiplas formas.

## Virtual e Override



O primeiro conceito de polimorfismo que vamos ver se chama override (sobrescrita). Em alguns casos a herança não é suficiente para atender todas as necessidades.

Por exemplo, observe o método FazerBarulho( ) da classe Animal

public void FazerBarulho()

{

MessageBox.Show("Fez barulho !");

}

Na forma como esse método está todo animal faz barulho do mesmo jeito ! O que não é necessariamente uma verdade no mundo real.

Então o correto seria sobrescrever o método FazerBarulho( ) nas classes que herdam de Animal.

Para fazer isso a primeira coisa que teremos de fazer é marcar o método fazer barulho na classe Animal com o modificador virtual

public virtual void FazerBarulho()

{

MessageBox.Show("Fez barulho !");

}

Em segundo lugar teremos de criar um método na classe cachorro com o modificador override que indica que esse método pertence a outra classe e está sendo modificado nesta.

public override void FazerBarulho()

{

MessageBox.Show("Au au au ");

}

Podemos fazer o mesmo na classe Galinha

public override void FazerBarulho()

{

MessageBox.Show("Cocóricó !");

}

Agora o cachorro tem seu próprio jeito de fazer barulho e a galinha também.

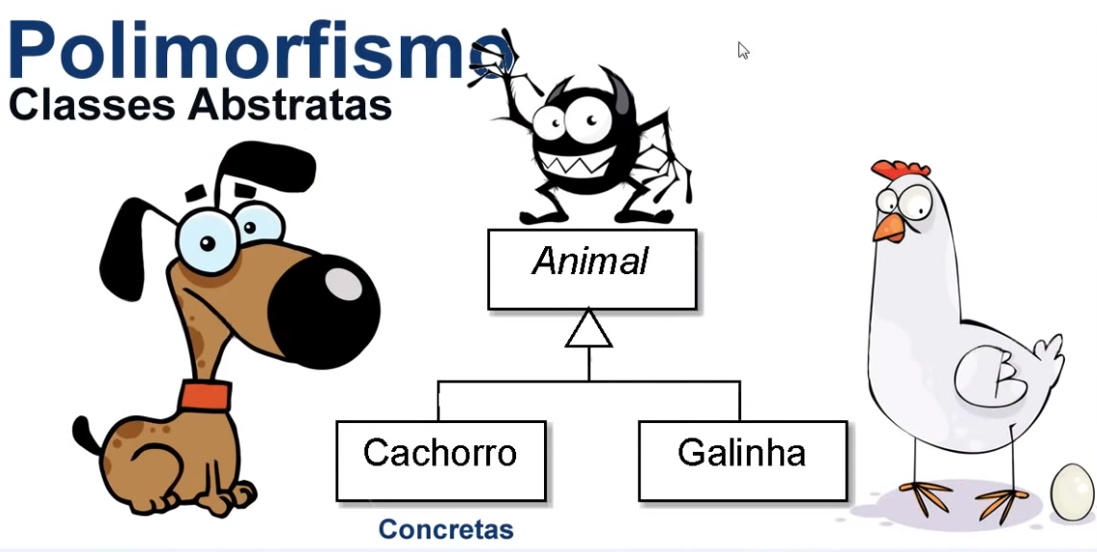
No entanto se criarmos um Animal genérico

Animal animal = new Animal(0, null);

animal.FazerBarulho();

Ele utilizará o método FazerBarulho( ) da classe Animal, o mesmo ocorre se não sobrescrevermos o método FazerBarulho( ).

## Abstract



Um método quando marcado com o modificador abstract não pode ter corpo, pois não tem uma forma definida.

O modificador abstract é utilizado nos métodos quando se deseja que a classe herdeira implemente o comportamento do método.

O método FazerBarulho( ) poderia ser abstrato, uma vez que nem todo animal faz barulho do mesmo jeito

public abstract void FazerBarulho();

Ao fazer isso a classe Animal também terá de receber o modificador abstract, ficando assim

abstract class Animal

{

...

}

Uma classe quando marcada como abstrata não pode ser instanciada, isso significa que o código abaixo não vai mais funcionar.

Animal animal = new Animal();

Animal animal = new Animal(0, null);

Quando um método é abstrato, ele não tem um comportamento definido, sendo assim, sempre que alguma classe herdar de Animal, será obrigatório, implementar o método FazerBarulho, para que seja possível definir como aquele Animal faz barulho.

## Métodos e classes sealed

O modificador sealed quando aplicado a um método significa que este método não pode ser sobrescrito em uma classe herdeira.

Por exemplo, vamos assumir que todo cachorro vai FazerBarulho( ) do mesmo jeito, ou seja, que o latido dos cachorros são sempre iguais

public sealed override void FazerBarulho()

{

MessageBox.Show("Au au au ");

}

Dessa forma se eu tiver uma classe que herde de cachorro, ela não poderá sobrescrever (alterar o comportamento) do método FazerBarulho( ).

Se quisermos que não seja possível que uma classe herde de Cachorro, podemos utilizar o modificador sealed na classe Cachorro

sealed class Cachorro : Animal

{

...

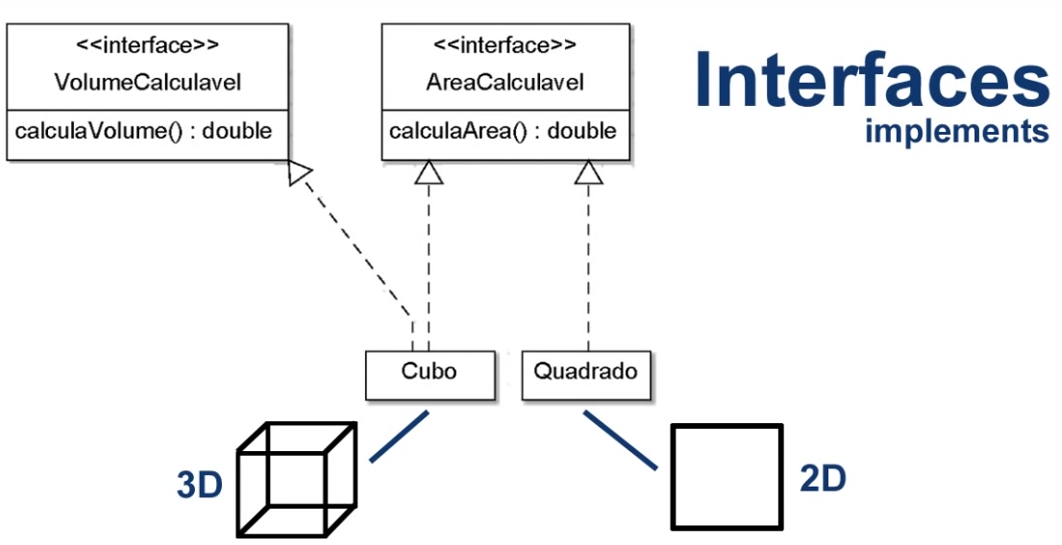
}

Dessa forma nenhuma classe poderá herdar de Cachorro.

## Interfaces

As interfaces têm por objetivo padronizar as interações com a aplicação.

Todos os métodos de uma interface são abstratos, diferente da classe abstrata que pode ter métodos abstratos (a implementar) e não abstratos (implementados).



Vamos começar criando uma interface chamada IAreaCalculavel

interface IAreaCalculavel

{

double CalculaArea();

}

Essa interface, terá um método chamado CalculaArea( ), que deve calcular a área de uma forma geométrica.

Vamos criar então uma classe chamada Quadrado

class Quadrado

{

private double lado;

public double Lado

{

get { return lado; }

set { lado = value; }

}

public Quadrado(double lado)

{

this.lado = lado;

}

}

E vamos fazer com que essa classe implemente a interface IAreaCalculavel

class Quadrado : IAreaCalculavel

{

private double lado;

public double Lado

{

get { return lado; }

set { lado = value; }

}

public Quadrado(double lado)

{

this.lado = lado;

}

public double CalculaArea()

{

return lado \* lado;

}

}

Ao utilizar o operador : para implementar a interface, você será obrigado a implementar, todos os métodos presentes na interface, uma vez que eles são abstratos.

Vamos criar uma segunda interface chamada IVolumeCalculavel

interface IVolumeCalculavel

{

double CalculaVolume();

}

Para utilizarmos essa interface vamos criar uma classe chamada Cubo

class Cubo

{

private double lado;

public double Lado

{

get { return lado; }

set { lado = value; }

}

public Cubo(double lado)

{

this.lado = lado;

}

}

E vamos implementar nessa classe a interface IAreaCalculavel, ao fazer isso seremos obrigados a implementar o método CalculaArea( )

class Cubo : IAreaCalculavel

{

private double lado;

public double Lado

{

get { return lado; }

set { lado = value; }

}

public Cubo(double lado)

{

this.lado = lado;

}

public double CalculaArea()

{

return 6 \* Math.Pow(lado, 2);

}

}

A grande vantagem, é que podemos implementar também a interface IVolumeCalculavel

class Cubo : IAreaCalculavel, IVolumeCalculavel

{

private double lado;

public double Lado

{

get { return lado; }

set { lado = value; }

}

public Cubo(double lado)

{

this.lado = lado;

}

public double CalculaArea()

{

return 6 \* Math.Pow(lado, 2);

}

public double CalculaVolume()

{

return Math.Pow(lado, 3);

}

}

Sendo assim, é possível utilizar o mesmo método, inclusive com a mesma assinatura, e quem define o comportamento do método é o objeto que o chama.

Quadrado quadrado = new Quadrado(2);

double area = quadrado.CalculaArea();

Cubo cubo = new Cubo(4);

double areaCubo = cubo.CalculaArea();

double volumeCubo = cubo.CalculaVolume();

Note que a assinatura do método CalculaArea( ) é a mesma tanto pro Quadrado quanto pro Cubo. O que altera o comportamento desse método é o objeto que chama ele, então quando você faz

double area = quadrado.CalculaArea();

A área vai ser calculada conforme definido na classe Quadrado

public double CalculaArea()

{

return lado \* lado;

}

Já quando você faz

double areaCubo = cubo.CalculaArea();

A área será calculada conforme definido na classe Cubo

public double CalculaArea()

{

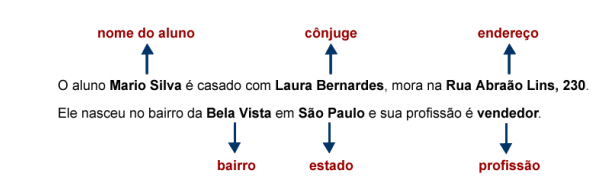
return 6 \* Math.Pow(lado, 2);

}

# Banco de Dados

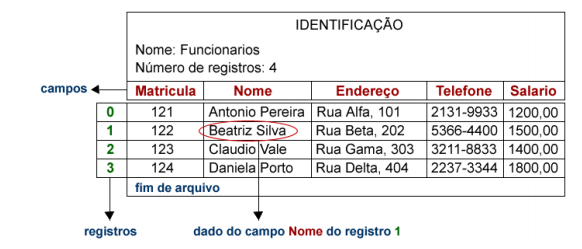
## 24.2. Diferença entre dados e informação

Observando a imagem abaixo podemos obter uma informação completa, porém essa informação é composta de dados menores. Isso significa que os dados, se separados, e fora de um contexto, não fazem sentido, para que os dados se tornem informação é preciso que haja uma ligação entre esses dados.



## 24.2. Formas de Armazenamento de Dados

Muito antigamente, a forma de armazenamento de dados era em arquivos sequenciais, onde cada registro de dados era realizado em linhas do arquivo.



Esse formato apresentava uma infinidade de problemas:

* **Inconsistência e redundância de dados:**

Uma vez que os arquivos são mantidos por aplicações feitas por vários programadores, ao longo do tempo, com as mudanças que as empresas enfrentam, é comum haver alteração dos formatos e criação de arquivos para novas implementações. Assim, os dados se repetem, pois cada aplicação possui a sua forma de armazenamento.

* **Dificuldade de acesso aos dados:**

Toda vez que o usuário necessita de outra forma de acesso aos dados, diferente da que o sistema oferece, o programador precisa mudar o programa ou, muitas vezes, desenvolver outro para atender ao usuário. Isso tem um custo elevado de desenvolvimento para as empresas, além do tempo gasto para elaborá-lo, na maioria das vezes, há urgência em receber os dados, retardando a tomada de decisões.

* **Isolamento dos dados:**

Como os dados estão armazenados em arquivos independentes, é possível que, ao tentar acessá-los de outro arquivo com formato diferente, não seja possível ou haja necessidade de criar novas aplicações para compatibilizá-los.

* **Falta de integridade:**

Uma aplicação de usuário geralmente é feita por vários arquivos e existe sempre uma dependência e uma relação entre os dados de um arquivo para outro. Se o programador não se preocupar, ou não conhecer essa dependência, correrá o risco de criar programas que, ao buscar dados, resultem em uma pesquisa errada, ocasionando danos à empresa.

* **Falta de atomicidade:**

Transação atômica – em computação – significa que, em uma interrupção qualquer no sistema por falha técnica, essa transação deve ser completada ou abortada, ou seja, todos os arquivos devem ser atualizados automaticamente ou retornados à situação anterior. No sistema baseado em arquivos, esta preocupação recai na mão dos programadores, que devem criar mecanismos de tratamento a falhas. Se a empresa não possui uma política de padronização das ações a serem tomadas, o programador pode errar no processo, afetando os dados.

* **Falta de segurança:**

Os sistemas baseados em arquivos dependem da proteção do sistema operacional e das permissões de acesso ao dispositivo onde estão armazenados. O programador poderá ter se lembrado de criar senha na sua aplicação, mas se outro programador tentar criar um programa para acessar o mesmo arquivo, sem proteção alguma, este fica vulnerável a modificações indesejadas, que normalmente são feitas pelos usuários.

## Sistema de banco de dados

Um Sistema de Gerenciamento de Banco de Dados (SGBD) consiste em uma coleção de dados inter-relacionados e um conjunto de programas que fazem acesso aos dados. Os SGBDs são concebidos para gerenciar desde um pequeno conjunto de dados até um grande volume.

Oferecem as estruturas para armazenamento e todos os mecanismos para manipulação, ou seja, inclusão, alteração, exclusão, seleção e busca dos dados.

A estrutura é criada pelos analistas de banco de dados ou pelos desenvolvedores de sistemas aplicativos para o usuário final obter fácil acesso.

Os sistemas de gerenciamento de banco de dados possuem as seguintes vantagens:

* Integridade dos dados.
* Segurança de acesso aos dados.
* Atomicidade nas transações.
* Concentração dos dados em um repositório, geralmente em um servidor de dados.
* Independência de linguagem de programação.
* Impõem regras de utilização para toda e qualquer aplicação que se conectar ao banco de dados.

# Linguagem SQL

A linguagem SQL (Sctructured Query Language) ou linguagem estruturada de consulta, é a linguagem utilizada pelo Sistemas de Bancos de Dados.

É a linguagem SQL que nos permite interagir com o servidor de banco de dados.

# 26. Subdivisão de Instruções SQL

A linguagem SQL se divide em grupos de comandos SQL, que veremos a seguir.

## 26.1. DDL (Data Definition Language)

Linguagem de Definição de Dados, define a estrutura dos dados e tabelas.

Exemplos:

* CREATE  
  Cria estrutura de dados
* ALTER

Modifica uma estrutura de dados

* DROP  
  Remove uma estrutura de dados

## 26.2. DML (Data Manipulation Language)

Linguagem de Manipulação de Dados, modifica o conteúdo das tabelas, alterando os dados.

Exemplos:

* INSERT

Insere novos registros na tabela

* UPDATE

Atualiza os registros da tabela

* DELETE

Remove Registros da tabela

* SELECT

Consulta dados da tabela

## 26.3. DCL (Data Control Language)

Linguagem de Controle de Dados alteram as permissões no banco de dados.

Exemplos:

* GRANT

Concede permissão de acesso

* REVOKE

Remove permissão de acesso

## 26.4. DTL (Data Transaction Language)

Faz o controle de Transações, registra, permanentemente, as alterações realizadas.

Exemplos:

* COMMIT

Registra permanentemente alterações

* ROLLBACK

Desfaz alteraçõs

## 26.5 Exercícios DDL

* + - 1. Crie um novo banco de dados chamado Empresa  
         -- Cria um novo banco de dados chamado Empresa  
         CREATE DATABASE Empresa;
      2. Selecione o banco Empresa para uso

-- Seleciona o banco de dados Empresa para uso

USE Empresa;

* + - 1. Cria a tabela TipoDependencia

-- Cria a tabela TipoDependencia

CREATE TABLE TipoDependencia (

IDDependencia BIGINT NOT NULL IDENTITY,

Descriao VARCHAR(20) NOT NULL UNIQUE,

PRIMARY KEY(IDDependencia)

);

* + - 1. Cria a tabela Funcionário

-- Cria a tabela Funcionario

CREATE TABLE Funcionario (

IDFuncionario BIGINT NOT NULL IDENTITY,

Nome VARCHAR(200) NOT NULL,

Cpf VARCHAR(11) NOT NULL UNIQUE,

Rg VARCHAR(10) NOT NULL,

Email VARCHAR(50) NOT NULL UNIQUE,

Telefone VARCHAR(14),

PRIMARY KEY(IDFuncionario)

);

* + - 1. Cria a tabela Dependente

-- Cria a tabela dependente

CREATE TABLE Dependente (

IDDependente BIGINT NOT NULL IDENTITY,

Nome VARCHAR(200) NOT NULL,

Cpf VARCHAR(11) NOT NULL UNIQUE,

DataNascimento DATETIME NOT NULL,

IDDependencia BIGINT NOT NULL,

IDFuncionario BIGINT NOT NULL,

PRIMARY KEY(IDDependente)

);

* + - 1. Relaciona as tabela de Dependente e TipoDependencia

-- Relaciona a tabela de dependente a tabela de tipo dependencia

ALTER TABLE Dependente

ADD CONSTRAINT FK\_Dependente\_TipoDependencia

FOREIGN KEY (IDDependencia)

REFERENCES TipoDependencia(IDDependencia);

* + - 1. Relaciona as tabelas Dependente e Funcionario

-- Relaciona a tabela de dependente a tabela de tipo funcionario

ALTER TABLE Dependente

ADD CONSTRAINT FK\_Dependente\_Funcionario FOREIGN KEY (IDFuncionario)

REFERENCES Funcionario(IDFuncionario);

* + - 1. Para Excluir uma tabela

DROP TABLE TipoDependencia;

**ATENÇÃO: Todos os registros da tabela serão excluídos.**

* + - 1. Para excluir um banco de dados

DROP DATABASE Empresa;

**ATENÇÃO: Todas as tabelas bem como os registros das mesmas serão excluídos.**

## 26.6. Exercícios DML

## 26.6.1. SELECT

Para consultar os dados de uma tabela utilizamos a instrução SELECT

Sintaxe:

SELECT [campo] FROM [tabela];

-- mostra todos os dados de todos os campos  
SELECT \* FROM Funcionario;

-- mostra todos os dados somente do campo Descricao

SELECT Descricao FROM TipoDependencia;

## 26.6.2. INSERT

Para inserir novos dados em uma tabela utilizamos a instrução INSERT

Sintaxe:

INSERT INTO [tabela]   
 (campo1, campo2, campoN) VALUES   
 (valorcampo1, valorcampo2, valorcampoN);

-- Cadastra novos dados na tabela Tipoependencia

INSERT INTO TipoDependencia (Descricao) VALUES ('Filho');

INSERT INTO TipoDependencia (Descricao) VALUES ('Filha');

INSERT INTO TipoDependencia (Descricao) VALUES ('Conjuge');

INSERT INTO TipoDependencia (Descricao) VALUES ('Pai');

INSERT INTO TipoDependencia (Descricao) VALUES ('Mae');

## 26.6.3. UPDATE

Para atualizar os dados de um registro em uma tabela utilizamos a instrução UPDATE

Sintaxe:

UPDATE [tabela]   
 SET campo1 = valorcampo1, campo2 = valorcampo2, campon=valorcampon  
 WHERE chaveprimaria = valorchaveprimaria;

-- Atualiza a descrição do registro de id 6

UPDATE TipoDependencia SET Descricao = 'Tia' WHERE IDDependencia = 6;

## 26.6.4. DELETE

Para excluir um registro em uma tabela utilizamos a instrução DELETE

Sintaxe:

DELETE FROM [tabela] WHERE chaveprimaria = valorchaveprimaria;

-- Exclui o registro de id 6

DELETE FROM TipoDependencia WHERE IDDependencia = 6;

## 26.6.5. Relatórios

Podemos obter diversos tipos de relatórios utilizando filtros na instrução SELECT

-- Ordena os registros da tabela por id em ordem crescente

SELECT \* FROM TipoDependencia ORDER BY IDDependencia;

-- Ordena os registros da tabela por id em ordem decrescente

SELECT \* FROM TipoDependencia ORDER BY IDDependencia DESC;

-- Ordena os registros da tabela por descrição em ordem crescente a-z

SELECT \* FROM TipoDependencia ORDER BY Descricao;

-- Ordena os registros da tabela por descrição em ordem crescente z-a

SELECT \* FROM TipoDependencia ORDER BY Descricao DESC;

-- Pesquisa registros que tenham a sílaba fi na descrição

SELECT \* FROM TipoDependencia WHERE Descricao LIKE '%Fi%';

-- Pesquisa registros que tenham a palavra filho na descrição

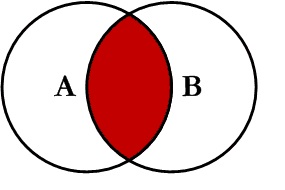
SELECT \* FROM TipoDependencia WHERE Descricao = 'Filho';

-- Pesquisa o registro de id igual a 1

SELECT \* FROM TipoDependencia WHERE IDDependencia = 1;

## 26.6.6. Relatórios com JOIN

A clausula JOIN é utilizada para fazer a junção de tabelas que possuem dados em comum:



-- Mostra as associações entre funcionarários e dependentes

SELECT \* FROM Funcionario AS F

JOIN Dependente AS D ON F.IDFuncionario = D.IDFuncionario;

-- Mostra as associações entre funcionarários e dependentes com seleção de campos

SELECT F.Nome, F.Cpf, F.Rg, D.Nome, D.DataNascimento FROM Funcionario AS F

JOIN Dependente AS D ON F.IDFuncionario = D.IDFuncionario;

-- Mostra quantos dependentes cada funcionário tem

SELECT F.Nome, COUNT(\*) AS Dependentes

FROM Funcionario AS F, Dependente AS D

WHERE F.IDFuncionario = D.IDFuncionario

GROUP BY F.Nome;

Podemos ainda realizar cálculos e devolver resultados utilizando as funções do Microsoft SQL Server

-- A função GETDATE() obtém a data atual

SELECT GETDATE() AS 'Data Atual';

-- A função YEAR() obtém o ano atual para isso precisa

-- receber uma data de parâmetro, isso foi realizado com

-- o uso da função GETDATE()

SELECT YEAR(GETDATE()) AS 'Ano Atual';

-- Mostra a idade dos dependentes

SELECT \*, YEAR(GETDATE()) - YEAR(D.DataNascimento) AS Idade   
FROM Dependente AS D;

-- Mostra a idade dos dependentes e os funcionários responsáveis por eles

SELECT F.Nome AS Funcionario, D.Nome AS Dependente, D.DataNascimento, YEAR(GETDATE()) - YEAR(D.DataNascimento) AS Idade

FROM Dependente AS D

JOIN Funcionario AS F ON D.IDFuncionario = F.IDFuncionario;

-- Mostra a idade dos dependentes e os funcionários responsáveis por eles

-- ordena por nome de funcionário em ordem alfabética crescente (a-z)

SELECT F.Nome AS Funcionario, D.Nome AS Dependente, D.DataNascimento, YEAR(GETDATE()) - YEAR(D.DataNascimento) AS Idade

FROM Dependente AS D

JOIN Funcionario AS F ON D.IDFuncionario = F.IDFuncionario   
ORDER BY Funcionario;

Com o uso da clausula JOIN podemos criar relatórios elaborados, utilizando inclusive mais de uma tabela.

-- Relaciona as tabelas de Dependentes, Funcionários e Tipo de Dependencia

SELECT \* FROM Dependente AS D

JOIN Funcionario AS F

ON D.IDFuncionario = F.IDFuncionario

JOIN TipoDependencia AS T

ON D.IDDependencia = T.IDDependencia

-- Relaciona as tabelas de Dependentes, Funcionários e Tipo de Dependencia

SELECT D.Nome, D.DataNascimento, F.Nome, F.Email, T.Descricao

FROM Dependente AS D

JOIN Funcionario AS F

ON D.IDFuncionario = F.IDFuncionario

JOIN TipoDependencia AS T

ON D.IDDependencia = T.IDDependencia

-- Relaciona as tabelas de Dependentes, Funcionários e Tipo de Dependencia

SELECT D.Nome AS 'Depedente', D.DataNascimento, F.Nome AS 'Funcionario', F.Email, T.Descricao

FROM Dependente AS D

JOIN Funcionario AS F

ON D.IDFuncionario = F.IDFuncionario

JOIN TipoDependencia AS T

ON D.IDDependencia = T.IDDependencia

Para saber mais:

Tipos de dados Transact-SQL (T-SQL)   
<https://msdn.microsoft.com/pt-br/library/ms187752(v=sql.120).aspx>

Outros tipos de JOIN  
<http://www.devmedia.com.br/sql-join-entenda-como-funciona-o-retorno-dos-dados/31006>

# Construindo uma aplicação para cadastrar funcionários

Vamos construir uma aplicação com Visual C# para cadastrar, consultar, alterar e remover funcionários.

## 27.1. Diagrama de Banco de Dados

Em primeiro lugar vamos projetar a aplicação, e a primeira coisa de que precisamos é de um diagrama de banco de dados que nos descreva a estrutura do banco de dados:



## 27.2. Diagrama de Classes

Em segundo lugar vamos desenvolver um diagrama de classes para a nossa aplicação

Uma imagem contendo captura de tela

Descrição gerada com alta confiança

## Mockups das Telas

Agora podemos desenvolver os projetos das telas que vamos construir

Home

Uma imagem contendo captura de tela

Descrição gerada com muito alta confiança

## Cadastro de Funcionários

Uma imagem contendo captura de tela

Descrição gerada com muito alta confiança

## Cadastro de Dependências

Uma imagem contendo captura de tela

Descrição gerada com muito alta confiança

## Cadastro de Dependentes

Uma imagem contendo captura de tela

Descrição gerada com muito alta confiança

# Código Fonte:

## 28.1.1. Senai.RH.Model.Dependente.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RH.Senai.RH.\_Model

{

class Dependente

{

private long id;

public long ID

{

get { return id; }

set { id = value; }

}

private string nome;

public string Nome

{

get { return nome; }

set { nome = value; }

}

private string cpf;

public string CPF

{

get { return cpf; }

set { cpf = value; }

}

private DateTime dataNascimento;

public DateTime DataNascimento

{

get { return dataNascimento; }

set { dataNascimento = value; }

}

// composição - relação tem um

private Dependencia dependencia;

public Dependencia Dependencia

{

get { return dependencia; }

set { dependencia = value; }

}

private Funcionario funcionario;

public Funcionario Funcionario

{

get { return funcionario; }

set { funcionario = value; }

}

// construtores

public Dependente()

{

}

public Dependente(long id, string nome, string cpf, DateTime dataNascimento, Dependencia dependencia, Funcionario funcionario)

{

this.id = id;

this.nome = nome;

this.cpf = cpf;

this.dataNascimento = dataNascimento;

this.dependencia = dependencia;

this.funcionario = funcionario;

}

}

}

## 28.1.2. Senai.RH.Model.Dependencia.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RH.Senai.RH.\_Model

{

class Dependencia

{

private long id;

public long ID

{

get { return id; }

set { id = value; }

}

private string descricao;

public string Descricao

{

get { return descricao; }

set { descricao = value; }

}

public Dependencia()

{

}

public Dependencia(long id, string descricao)

{

this.id = id;

this.descricao = descricao;

}

// sobrescreve o método ToString()

public override string ToString()

{

return this.descricao;

}

}

}

## 28.1.3. Senai.RH.Model.Funcionario.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RH.Senai.RH.\_Model

{

class Funcionario

{

private long id;

public long ID

{

get { return id; }

set { id = value; }

}

private string nome;

public string Nome

{

get { return nome; }

set { nome = value; }

}

private string cpf;

public string CPF

{

get { return cpf; }

set { cpf = value; }

}

private string rg;

public string RG

{

get { return rg; }

set { rg = value; }

}

private string email;

public string Email

{

get { return email; }

set { email = value; }

}

private string telefone;

public string Telefone

{

get { return telefone; }

set { telefone = value; }

}

// construtores

public Funcionario()

{

}

public Funcionario(long id, string nome, string cpf, string rg, string email, string telefone)

{

this.id = id;

this.nome = nome;

this.cpf = cpf;

this.rg = rg;

this.email = email;

this.telefone = telefone;

}

// sobrescreve o método ToString()

public override string ToString()

{

return this.nome;

}

}

}

## 28.1.4. Senai.RH.Dao.ConnectionFactory.cs

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RH.Senai.RH.Dao

{

class ConnectionFactory

{

// método de conexão com o banco de dados

public SqlConnection GetConnection() {

// string de conexão com o servidor

string servidor = @"Data Source=.\SQLEXPRESS; Initial Catalog=Empresa; Integrated Security=true; Pooling=false";

// cria uma instância de SqlConnection

SqlConnection connection = new SqlConnection(servidor);

// retorna um SqlConnection

return connection;

}

}

}

## 28.1.5. Senai.RH.Dao.IDao.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RH.Senai.RH.Dao

{

interface IDao<T>

{

void Salvar(T t);

List<T> Consultar();

void Excluir(T t);

T Consultar(string parametro);

}

}

## 28.1.6. Senai.RH.Dao.FuncionarioDao.cs

using RH.Senai.RH.\_Model;

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Dao

{

class FuncionarioDao : IDao<Funcionario>

{

// atributos

// conexão com o banco de dados

private SqlConnection connection;

// instrução sql

private string sql = null;

// mensagem do messagebox

private string msg = null;

// título do messagebox

private string titulo = null;

// construtor

public FuncionarioDao()

{

// cria uma conexão com o banco de dados

connection = new ConnectionFactory().GetConnection();

}

public Funcionario Consultar(string parametro)

{

// instrução sql

sql = "SELECT \* FROM Funcionario WHERE Cpf = @Cpf";

Funcionario funcionario = null;

try

{

// abre a conexão com o banco de dados

connection.Open();

// comando sql

SqlCommand cmd = new SqlCommand(sql, connection);

// parametro do comando sql

cmd.Parameters.AddWithValue("@Cpf", parametro);

// leitor de dados sql

// recebe os dados do banco de dados

SqlDataReader leitor = cmd.ExecuteReader();

while (leitor.Read())

{

// compara se o cpf dos registros é igual

// ao cpf de parâmetro

// 36421147844.Equals(36421147844)

if (parametro.Equals(leitor["Cpf"].ToString()))

{

// cria o funcionário

funcionario = new Funcionario();

funcionario.ID = (long)leitor["IDFuncionario"];

funcionario.Nome = leitor["Nome"].ToString();

funcionario.CPF = leitor["Cpf"].ToString();

funcionario.RG = leitor["Rg"].ToString();

funcionario.Email = leitor["Email"].ToString();

funcionario.Telefone = leitor["Telefone"].ToString();

} // fim do if

} // fim do while

} // fim do try

catch (SqlException ex)

{

msg = "Erro ao consultar o funcionário !";

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

// retorna o funcionário

return funcionario;

}

// métodos

public List<Funcionario> Consultar()

{

// comando sql de consulta

sql = "SELECT \* FROM Funcionario";

// lista de funcionários cadastrados

List<Funcionario> funcionarios = new List<Funcionario>();

try

{

// abre a conexão com o banco de dados

connection.Open();

// comando sql

SqlCommand cmd = new SqlCommand(sql, connection);

// cria um leitor de dados

SqlDataReader leitor = cmd.ExecuteReader();

// enquanto o leitor tiver dados para ler

while (leitor.Read())

{

Funcionario funcionario = new Funcionario();

funcionario.ID = (long)leitor["IDFuncionario"];

funcionario.Nome = leitor["Nome"].ToString();

funcionario.CPF = leitor["Cpf"].ToString();

funcionario.RG = leitor["Rg"].ToString();

funcionario.Email = leitor["Email"].ToString();

funcionario.Telefone = leitor["Telefone"].ToString();

// adiciona o funcionário na lista de funcionários

funcionarios.Add(funcionario);

} // fim do while

}

catch (SqlException ex)

{

msg = "Erro ao consultar os funcionários cadastrados: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

return funcionarios;

}

public void Excluir(Funcionario funcionario)

{

// instrução sql

sql = "DELETE FROM Funcionario WHERE IDFuncionario = @IDFuncionario";

try

{

// abre a connexão com o banco de dados

connection.Open();

// cria um comando sql

SqlCommand cmd = new SqlCommand(sql, connection);

// adiciona valor ao parâmetro @IDFuncionario

cmd.Parameters.AddWithValue("@IDFuncionario", funcionario.ID);

// executa o comando sql no banco de dados

cmd.ExecuteNonQuery();

// mensagem de feedback

msg = "Funcionário " + funcionario.Nome + " excluído com sucesso !";

// titulo da mensagem de feedback

titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

// mensagem de feedback

msg = "Erro ao excluir o funcionário !" + ex.Message;

// titulo da mensagem de erro

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

// fecha a conexão com o banco de dados

connection.Close();

}

}

public void Salvar(Funcionario funcionario)

{

// verifica SE o id do funcionário é DIFERENTE de 0

if (funcionario.ID != 0)

{

// update

sql = "UPDATE Funcionario SET Nome=@Nome, Cpf=@Cpf, Rg=@Rg, Email=@Email, Telefone=@Telefone WHERE IDFuncionario = @IDFuncionario";

}

else

{

// insert

sql = "INSERT INTO Funcionario(Nome, Cpf, Rg, Email, Telefone) VALUES (@Nome, @Cpf, @Rg, @Email, @Telefone)";

}

try

{

// abre uma conexão com o banco de dados

connection.Open();

// comando sql

SqlCommand cmd = new SqlCommand(sql, connection);

cmd.Parameters.AddWithValue("@IDFuncionario", funcionario.ID);

cmd.Parameters.AddWithValue("@Nome", funcionario.Nome);

cmd.Parameters.AddWithValue("@Cpf", funcionario.CPF);

cmd.Parameters.AddWithValue("@Rg", funcionario.RG);

cmd.Parameters.AddWithValue("@Email", funcionario.Email);

cmd.Parameters.AddWithValue("@Telefone", funcionario.Telefone);

// executa o INSERT

cmd.ExecuteNonQuery();

msg = "Funcionario " + funcionario.Nome + " salvo com sucesso !";

titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

msg = "Erro o salvar o funcionário: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

}

}

}

## 28.1.7. Senai.RH.Dao.DependenciaDao.cs

using RH.Senai.RH.\_Model;

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Dao

{

class DependenciaDao : IDao<Dependencia>

{

private SqlConnection connection;

private string sql = null;

private string msg = null;

private string titulo = null;

public DependenciaDao()

{

connection = new ConnectionFactory().GetConnection();

}

public List<Dependencia> Consultar()

{

sql = "SELECT \* FROM Dependencia";

List<Dependencia> dependencias = new List<Dependencia>();

try

{

connection.Open();

SqlCommand cmd = new SqlCommand(sql, connection);

SqlDataReader leitor = cmd.ExecuteReader();

while (leitor.Read())

{

Dependencia dependencia = new Dependencia();

dependencia.ID = (long)leitor["IDDependencia"];

dependencia.Descricao = leitor["Descricao"].ToString();

dependencias.Add(dependencia);

}

}

catch (SqlException ex)

{

msg = "Erro ao consultar o cadastro de dependências\nErro: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

return dependencias;

}

public Dependencia Consultar(string parametro)

{

throw new NotImplementedException();

}

public void Excluir(Dependencia dependencia)

{

sql = "DELETE FROM Dependencia WHERE IDDependencia = @IDDependencia";

try

{

connection.Open();

SqlCommand cmd = new SqlCommand(sql, connection);

cmd.Parameters.AddWithValue("@IDDependencia", dependencia.ID);

cmd.ExecuteNonQuery();

string msg = "Dependência excluída com sucesso !";

string titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

string msg = "Erro ao excluir a dependência\nErro: " + ex.Message;

string titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

}

public void Salvar(Dependencia dependencia)

{

if (dependencia.ID != 0)

{

sql = "UPDATE Dependencia SET Descricao = @Descricao WHERE IDDependencia = @IDDependencia";

}

else

{

sql = "INSERT INTO Dependencia (Descricao) VALUES (@Descricao)";

}

try

{

connection.Open();

SqlCommand cmd = new SqlCommand(sql, connection);

cmd.Parameters.AddWithValue("@Descricao", dependencia.Descricao);

cmd.Parameters.AddWithValue("@IDDependencia", dependencia.ID);

cmd.ExecuteNonQuery();

msg = "Dependência salva com sucesso !";

titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

msg = "Erro ao salvar a dependência\nErro: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

}

}

}

## 28.1.8. Senai.RH.Dao.DependenteDao.cs

using RH.Senai.RH.\_Model;

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Dao

{

class DependenteDao : IDao<Dependente>

{

// variáveis do dao

// conexão com o banco de dados

private SqlConnection connection;

// instrução sql

private string sql = null;

// mensagem e título do message box

string msg = null;

string titulo = null;

// construtor

public DependenteDao()

{

// conecta a aplicação ao banco de dados

connection = new ConnectionFactory().GetConnection();

}

public List<Dependente> Consultar()

{

// intrução sql

sql = "SELECT \* FROM DependenteView";

// lista de dependentes

List<Dependente> dependentes = new List<Dependente>();

// vdm

try

{

// abre uma conexão com o banco de dados

connection.Open();

// cria um comando sql

SqlCommand cmd = new SqlCommand(sql, connection);

// cria um leitor para receber os dados da tabela

SqlDataReader leitor = cmd.ExecuteReader();

// enquanto o leitor estiver lendo

while (leitor.Read())

{

// cria uma instancia de dependencia

Dependencia dependencia = new Dependencia();

// define os dados da dependencia

dependencia.ID = (long)leitor["IDDependencia"];

dependencia.Descricao = leitor["Descricao"].ToString();

// cria uma instância de funcionário

Funcionario funcionario = new Funcionario();

// define os dados do funcionario

funcionario.ID = (long)leitor["IDFuncionario"];

funcionario.Nome = leitor["NomeFuncionario"].ToString();

funcionario.CPF = leitor["CPFFuncionario"].ToString();

funcionario.RG = leitor["Rg"].ToString();

funcionario.Email = leitor["Email"].ToString();

funcionario.Telefone = leitor["Telefone"].ToString();

// cria uma instância de dependente

Dependente dependente = new Dependente();

// define os dados do dependente

dependente.ID = (long)leitor["IDDependente"];

dependente.Nome = leitor["NomeDependente"].ToString();

dependente.CPF = leitor["CPFDependente"].ToString();

dependente.DataNascimento = (DateTime)leitor["DataNascimento"];

dependente.Dependencia = dependencia;

dependente.Funcionario = funcionario;

// adiciona o dependente na lista

dependentes.Add(dependente);

}

}

catch (SqlException ex)

{

msg = "Erro ao consultar os dependentes\nErro: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

// retorna a lista de dependentes

return dependentes;

}

public Dependente Consultar(string parametro)

{

throw new NotImplementedException();

}

public void Excluir(Dependente dependente)

{

sql = "DELETE FROM Dependente WHERE IDDependente = @IDDependente";

try

{

connection.Open();

SqlCommand cmd = new SqlCommand(sql, connection);

cmd.Parameters.AddWithValue("@IDDependente", dependente.ID);

cmd.ExecuteNonQuery();

cmd.ExecuteNonQuery();

string msg = "Dependente excluído com sucesso !";

string titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

string msg = "Erro ao excluir o dependente\nErro: " + ex.Message;

string titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

}

public void Salvar(Dependente dependente)

{

if (dependente.ID != 0)

{

sql = "UPDATE Dependente SET Nome = @Nome, Cpf = @Cpf, DataNascimento = @DataNascimento, IDFuncionario = @IDFuncionario, IDDependencia = @IDDependencia";

}

else

{

sql = "INSERT INTO Dependente (Nome, Cpf, DataNascimento, IDFuncionario, IDDependencia) VALUES (@Nome, @Cpf, @DataNascimento, @IDFuncionario, @IDDependencia)";

}

// vdm

try

{

connection.Open();

SqlCommand cmd = new SqlCommand(sql, connection);

cmd.Parameters.AddWithValue("@Nome", dependente.Nome);

cmd.Parameters.AddWithValue("@Cpf", dependente.CPF);

cmd.Parameters.AddWithValue("@DataNascimento", dependente.DataNascimento);

cmd.Parameters.AddWithValue("@IDFuncionario", dependente.Funcionario.ID);

cmd.Parameters.AddWithValue("@IDDependencia", dependente.Dependencia.ID);

cmd.Parameters.AddWithValue("@IDDependente", dependente.ID);

cmd.ExecuteNonQuery();

msg = "Dependente salvo com sucesso";

titulo = "Sucesso...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (SqlException ex)

{

msg = "Erro ao salvar o dependente.\nErro: " + ex.Message;

titulo = "Erro...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

finally

{

connection.Close();

}

}

}

}

## 28.1.9. Senai.RH.Forms.HomeForm.cs

using RH.Senai.RH.Forms;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH

{

public partial class HomeForm : Form

{

public HomeForm()

{

InitializeComponent();

}

private void pictureBox1\_Click(object sender, EventArgs e)

{

string mensagem = "Sistema de RH v1.0\nDesenvolvido por: Jorge Rabello";

string titulo = "Sobre...";

MessageBox.Show(mensagem, titulo, MessageBoxButtons.OK, MessageBoxIcon.Information);

}

private void funcionárioToolStripMenuItem\_Click(object sender, EventArgs e)

{

// abre o form de cadastro de funcionários

// cria uma instância do form

FuncionarioForm form = new FuncionarioForm();

// chama o método ShowDialog()

form.ShowDialog();

}

private void dependênciaToolStripMenuItem\_Click(object sender, EventArgs e)

{

DependenciaForm form = new DependenciaForm();

form.ShowDialog();

}

private void dependenteToolStripMenuItem\_Click(object sender, EventArgs e)

{

//Cria uma instância de DependenteForm

DependenteForm form = new DependenteForm();

// exibe o form

form.ShowDialog();

}

}

}

## 28.1.10. Senai.RH.Forms.FuncionarioForm.cs

using RH.Senai.RH.\_Model;

using RH.Senai.RH.Dao;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Forms

{

public partial class FuncionarioForm : Form

{

public FuncionarioForm()

{

InitializeComponent();

}

private void btnSalvarFuncionario\_Click(object sender, EventArgs e)

{

// validação

// verifica se os campos obrigatórios foram

// preenchidos

if (string.IsNullOrEmpty(txtNomeFuncionario.Text)

&& string.IsNullOrEmpty(txtCpfFuncionario.Text)

&& string.IsNullOrEmpty(txtRgFuncionario.Text)

&& string.IsNullOrEmpty(txtEmailFuncionario.Text)

&& string.IsNullOrEmpty(txtTelefoneFuncionario.Text))

{

MessageBox.Show("Preencha todos os campos !");

}

else

{

// instancia um funcionário

Funcionario funcionario = new Funcionario();

// obtém o id do funcionário

// SE o TextBox de id de funcionário NÃO estiver vazio

// se foi preenchido o TextBox de IDFuncionário

if (!string.IsNullOrEmpty(txtIDFuncionario.Text))

{

// cria um id

long id = 0;

// converte o texto do TextBox para long

// armazenando na variável id

if (long.TryParse(txtIDFuncionario.Text, out id))

{

// atribui o id ao objeto funcionário

funcionario.ID = id;

}

}

// atribui dados ao funcionário

funcionario.Nome = txtNomeFuncionario.Text;

funcionario.CPF = txtCpfFuncionario.Text;

funcionario.RG = txtRgFuncionario.Text;

funcionario.Email = txtEmailFuncionario.Text;

funcionario.Telefone = txtTelefoneFuncionario.Text;

// instancia o dao de funcionários

FuncionarioDao dao = new FuncionarioDao();

// salva o funcionário no banco de dados

dao.Salvar(funcionario);

// atualiza o data grid view

PreencheDados();

LimparFormulario();

}

} // fim do evento de click

// métodos do programador

private void LimparFormulario()

{

txtIDFuncionario.Clear();

txtNomeFuncionario.Clear();

txtCpfFuncionario.Clear();

txtRgFuncionario.Clear();

txtEmailFuncionario.Clear();

txtTelefoneFuncionario.Clear();

txtCpfFuncionario.Focus();

}

private void FuncionarioForm\_Load(object sender, EventArgs e)

{

PreencheDados();

}

// preenche os dados do data grid view

private void PreencheDados()

{

// instancia uma dao

// FuncionarioDao dao = new FuncionarioDao();

// preenche o data grid view

dgvFuncionarios.DataSource = new FuncionarioDao().Consultar();

// oculta algumas colunas

dgvFuncionarios.Columns["ID"].Visible = false;

dgvFuncionarios.Columns["Rg"].Visible = false;

// limpa a seleção do data grid view

dgvFuncionarios.ClearSelection();

// limpa os campos do formulário

LimparFormulario();

}

private void dgvFuncionarios\_SelectionChanged(object sender, EventArgs e)

{

// se alguma linha for selecionada

if (dgvFuncionarios.CurrentRow != null)

{

// pega o id e coloca no textbox de id

txtIDFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[0].Value.ToString();

txtNomeFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[1].Value.ToString();

txtCpfFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[2].Value.ToString();

txtRgFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[3].Value.ToString();

txtEmailFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[4].Value.ToString();

txtTelefoneFuncionario.Text = dgvFuncionarios.CurrentRow.Cells[5].Value.ToString();

}

}

private void btnExcluirFuncionario\_Click(object sender, EventArgs e)

{

// verifica se o TextBox de ID de Funcionário é nulo ou

// vazio.

// se sim isso significa que nenhum funcionário foi

// selecionado na lista

if (string.IsNullOrEmpty(txtIDFuncionario.Text))

{

string msg = "Selecione um funcionário na lista abaixo !";

string titulo = "Operação não realizada...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Warning);

}

else

{

// instancia um funcionario

Funcionario funcionario = new Funcionario();

// cria um id para o funcionário

long id = 0;

// converte a string do text box de id para long

if (long.TryParse(txtIDFuncionario.Text, out id))

{

// atribui o id ao id do funcionário

funcionario.ID = id;

}

// instancia uma dao

FuncionarioDao dao = new FuncionarioDao();

// resposta do usuário

DialogResult resposta = MessageBox.Show("Deseja realmente excluir este funcionário ?", "Mensagem...", MessageBoxButtons.YesNo, MessageBoxIcon.Question);

// se o usuário respondeu sim

if (resposta.Equals(DialogResult.Yes))

{

// exclui o funcionário

dao.Excluir(funcionario);

// atualiza p data grid view

PreencheDados();

}

}

}

private void txtCpfFuncionario\_Leave(object sender, EventArgs e)

{

// pega o texto do textbox de cpf

string cpf = txtCpfFuncionario.Text;

// cria um funcionário

Funcionario funcionario = new FuncionarioDao().Consultar(cpf);

// verifica SE funcionário não é nulo

if (funcionario != null)

{

// preenche os campos do form

txtIDFuncionario.Text = funcionario.ID.ToString();

txtCpfFuncionario.Text = funcionario.CPF;

txtRgFuncionario.Text = funcionario.RG;

txtNomeFuncionario.Text = funcionario.Nome;

txtEmailFuncionario.Text = funcionario.Email;

txtTelefoneFuncionario.Text = funcionario.Telefone;

}

}

} // fim da classe

}

## 28.1..11 Senai.RH.Forms.DependenteForm.cs

using RH.Senai.RH.\_Model;

using RH.Senai.RH.Dao;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Forms

{

public partial class DependenteForm : Form

{

public DependenteForm()

{

InitializeComponent();

}

private void DependenteForm\_Load(object sender, EventArgs e)

{

// preenche as combobox

PreencheComboBox();

// Preenche o data grid view

PreencheDados();

}

// preenche os dados da combobox

private void PreencheComboBox()

{

// preenche a combox de dependencia

// define a fonte de dados da combobox

cboDependencia.DataSource = new DependenciaDao().Consultar();

// define o que mostrar na combobox

cboDependencia.DisplayMember = "Descricao";

cboFuncionario.DataSource = new FuncionarioDao().Consultar();

cboFuncionario.DisplayMember = "Nome";

}

// preenche os dados do data grid view

private void PreencheDados()

{

// define a fonte de dados do data grid view

dgvDependentes.DataSource = new DependenteDao().Consultar();

// adicionado depois

dgvDependentes.Columns["ID"].Visible = false;

dgvDependentes.ClearSelection();

LimpaFormulario();

}

private void btnSalvar\_Click(object sender, EventArgs e)

{

// cria uma nova instância de dependente

Dependente dependente = new Dependente();

if (!string.IsNullOrEmpty(txtIDDependente.Text))

{

long id = 0;

if (long.TryParse(txtIDDependente.Text, out id))

{

dependente.ID = id;

}

}

// preenche os dados do dependente

dependente.Nome = txtNomeDependente.Text;

dependente.CPF = txtCPFDependente.Text;

// pega a data do date time picker

dependente.DataNascimento = dtpDataNascimento.Value;

// obtém a dependência do combobox

Dependencia dependencia = (Dependencia)cboDependencia.SelectedItem;

Funcionario funcionario = (Funcionario)cboFuncionario.SelectedItem;

// associa o dependente a dependencia

dependente.Dependencia = dependencia;

// associa o dependente ao funcionario

dependente.Funcionario = funcionario;

// cria uma instância de dao de dependente

DependenteDao dao = new DependenteDao();

// salva o dependente

dao.Salvar(dependente);

// atualiza o datagridview

PreencheDados();

// limpa o formulário

LimpaFormulario();

}

private void LimpaFormulario()

{

txtIDDependente.Clear();

txtCPFDependente.Clear();

dtpDataNascimento.Value = DateTime.Now;

txtNomeDependente.Clear();

PreencheComboBox();

txtCPFDependente.Focus();

}

private void dgvDependentes\_SelectionChanged(object sender, EventArgs e)

{

// preenche os campos do form

// com os dados do item selecionado

if (dgvDependentes.CurrentRow != null)

{

txtIDDependente.Text = dgvDependentes.CurrentRow.Cells[0].Value.ToString();

txtNomeDependente.Text = dgvDependentes.CurrentRow.Cells[1].Value.ToString();

txtCPFDependente.Text = dgvDependentes.CurrentRow.Cells[2].Value.ToString();

dtpDataNascimento.Value = (DateTime)dgvDependentes.CurrentRow.Cells[3].Value;

// define o item da combobox

cboDependencia.Text = dgvDependentes.CurrentRow.Cells[4].Value.ToString();

cboFuncionario.Text = dgvDependentes.CurrentRow.Cells[5].Value.ToString();

}

}

private void btnExcluir\_Click(object sender, EventArgs e)

{

if (string.IsNullOrEmpty(txtIDDependente.Text))

{

string msg = "Selecione um funcionário na lista abaixo";

string titulo = "Operação não realizada...";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Warning);

}

else

{

Dependente dependente = new Dependente();

long id = 0;

if (long.TryParse(txtIDDependente.Text, out id))

{

dependente.ID = id;

}

DialogResult dr = MessageBox.Show("Deseja realmente excluir esse dependente ?", "Mensagem...", MessageBoxButtons.YesNo, MessageBoxIcon.Question);

if (dr.Equals(DialogResult.Yes))

{

DependenteDao dao = new DependenteDao();

dao.Excluir(dependente);

PreencheDados();

LimpaFormulario();

}

}

}

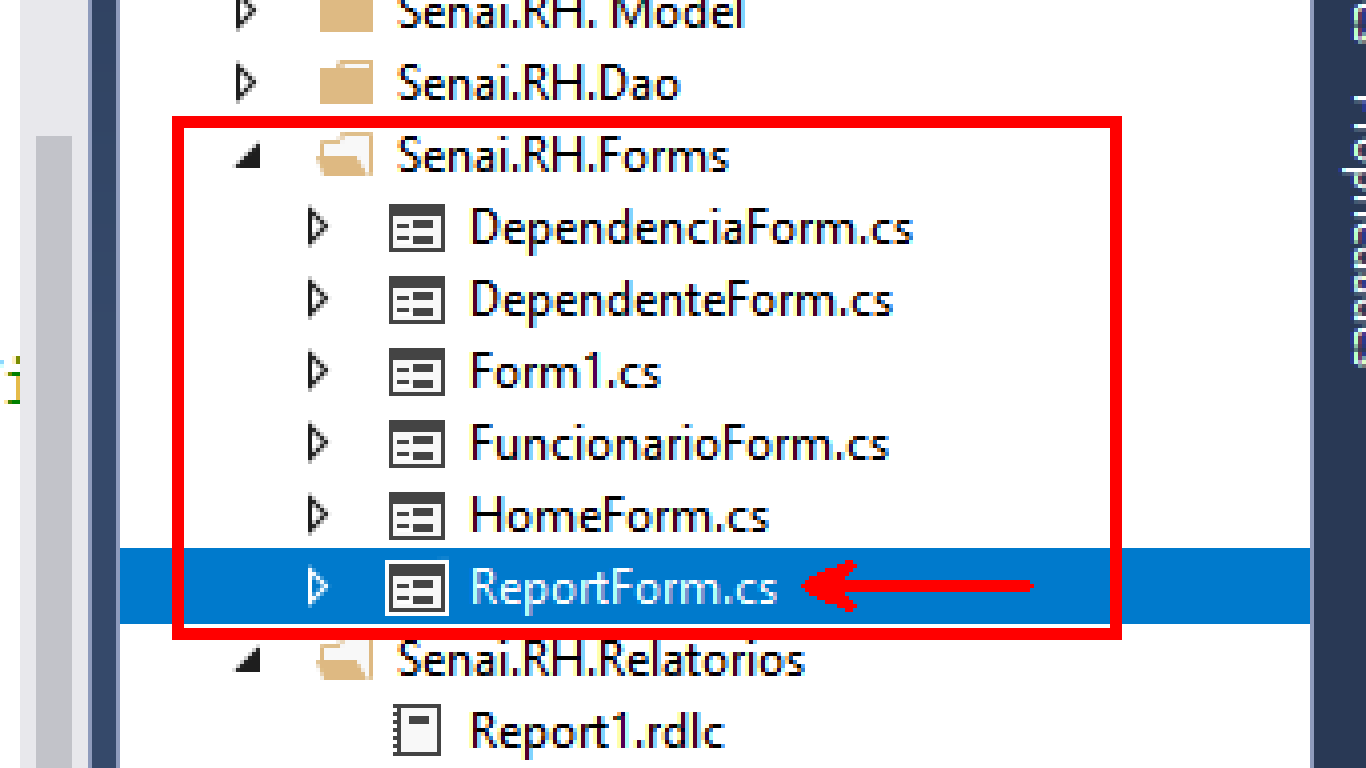
}

}

## Gerando relatórios

O primeiro para para se gerar relatórios é instalar a o ReportViewer, você pode encontrar um passo a passo dessa instalação em <http://www.macoratti.net/17/02/rpv_vs17.htm>.

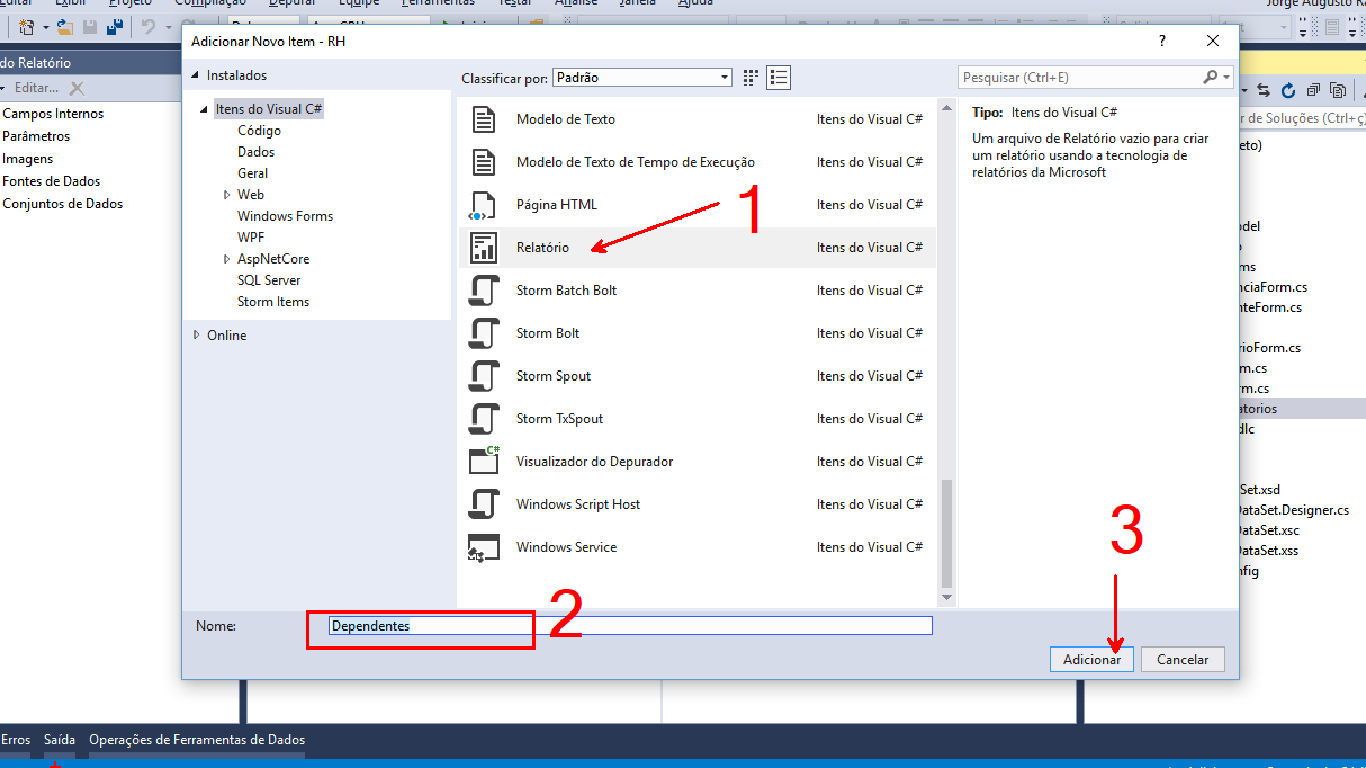
Para gerar o relatório em primeiro lugar vamos criar um formulário que deverá abrigar o relatório, crie um formulário chamado ReportForm



A seguir vamos criar o relatório em si, crie em novo namespace chamado Senai.RH.Util

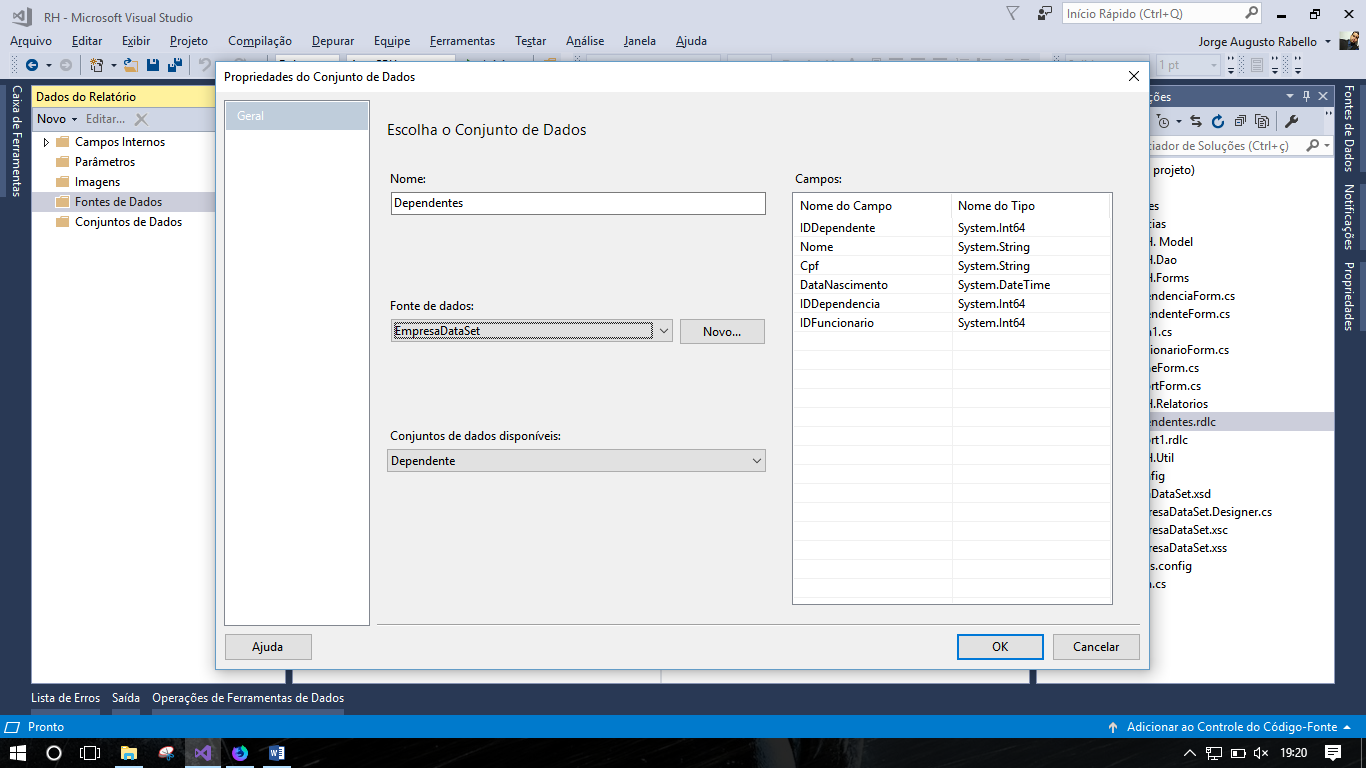
Clique com o botão direito no novo namespace e selecione Adicionar > Novo Item...

Escolha a opção Relatório, nomeie como Dependentes e clique em Adicionar.

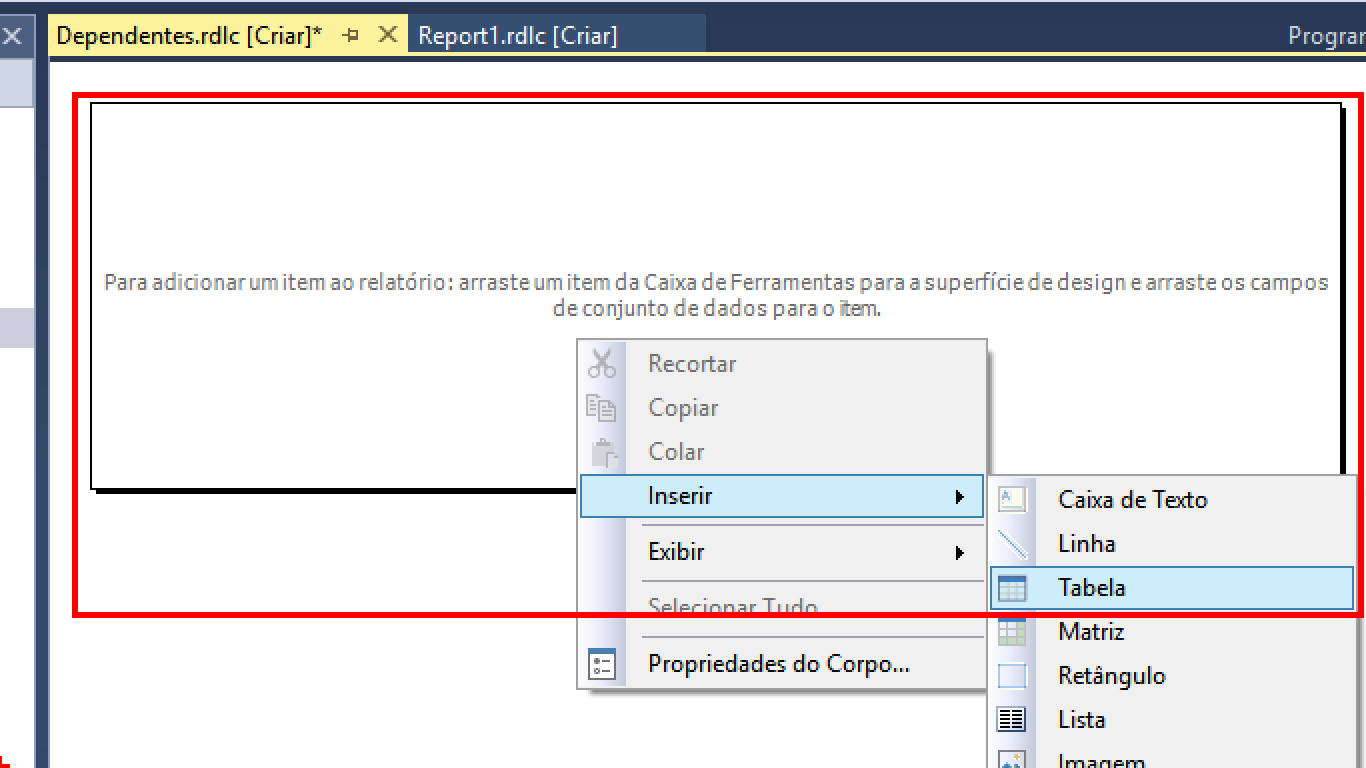


Com o novo relatório criado será aberta a janela Dados do Relatório clique em Novo > Conjunto de Dados

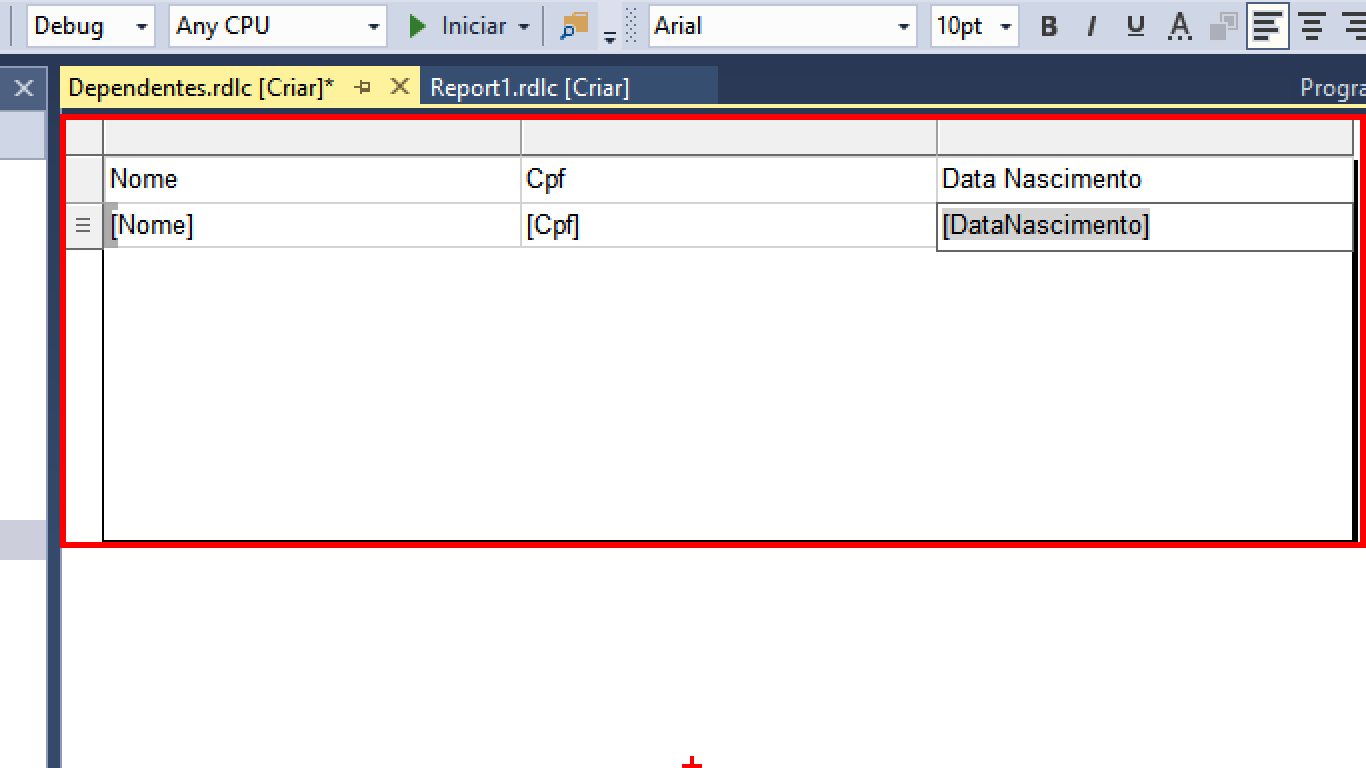
Configure a Conjunto de Dados (DataSet) e a fonte de dados (Data Source)



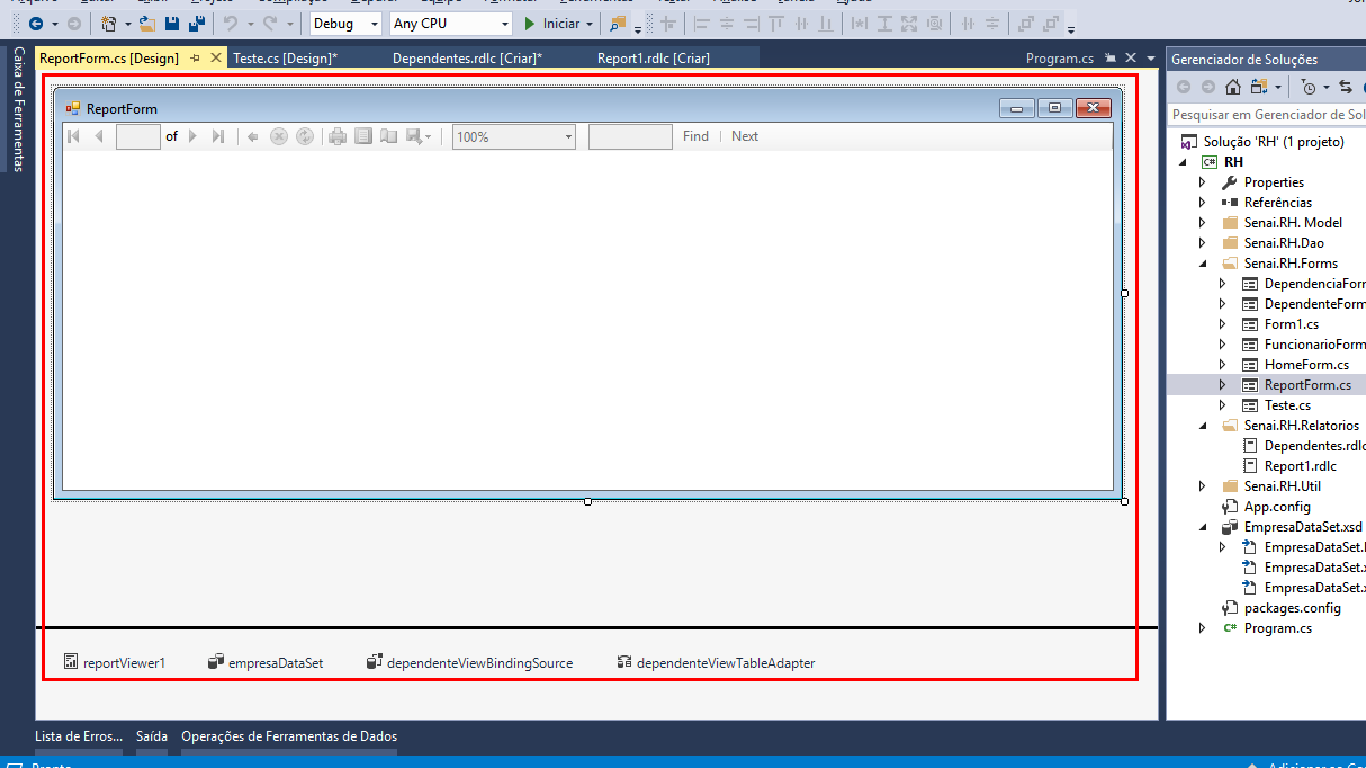
No relatório clique com o botão direito Inserir > Tabela



Acrescente os campos que deseja exibir opcionalmente você pode acrescentar um cabeçalho e um rodapé ao relatório.



Crie um novo Formulário e adicione a esse formulário o componente ReportViewer, seu formulário ficará com a seguinte aparência



Pressione a tela F7 e edite o código para que ao carregar o form o ReportViwer seja preenchido

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace RH.Senai.RH.Forms

{

public partial class ReportForm : Form

{

public ReportForm()

{

InitializeComponent();

}

private void ReportForm\_Load(object sender, EventArgs e)

{

this.dependenteViewTableAdapter.Fill(this.empresaDataSet.DependenteView);

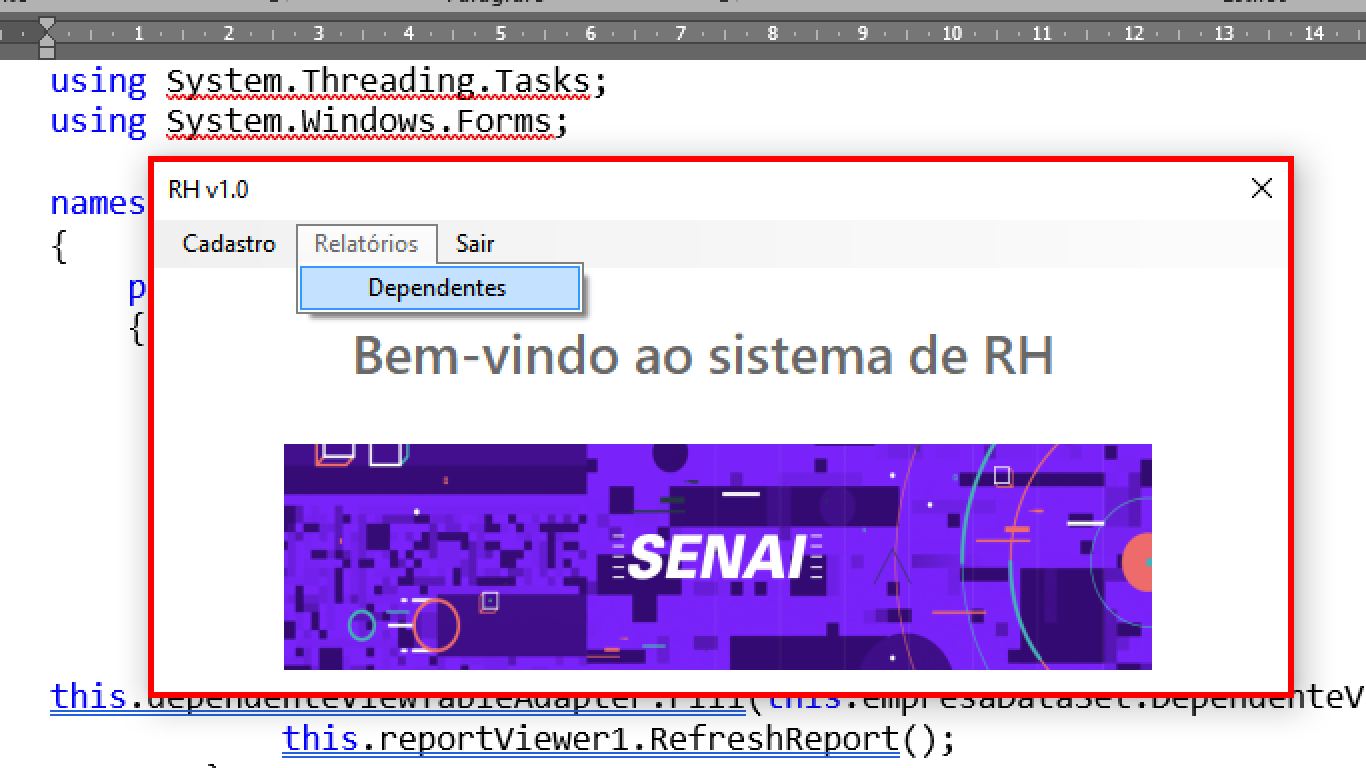
this.reportViewer1.RefreshReport();

}

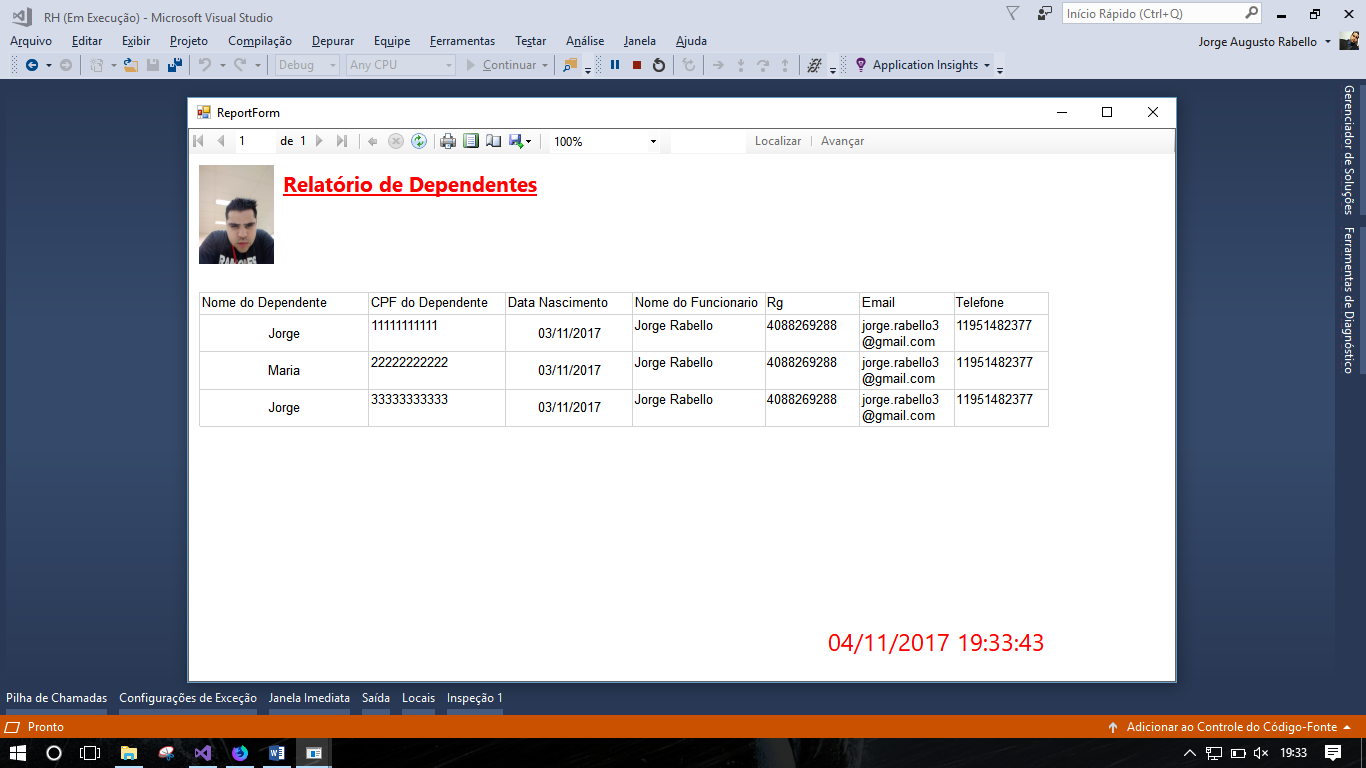
}

}

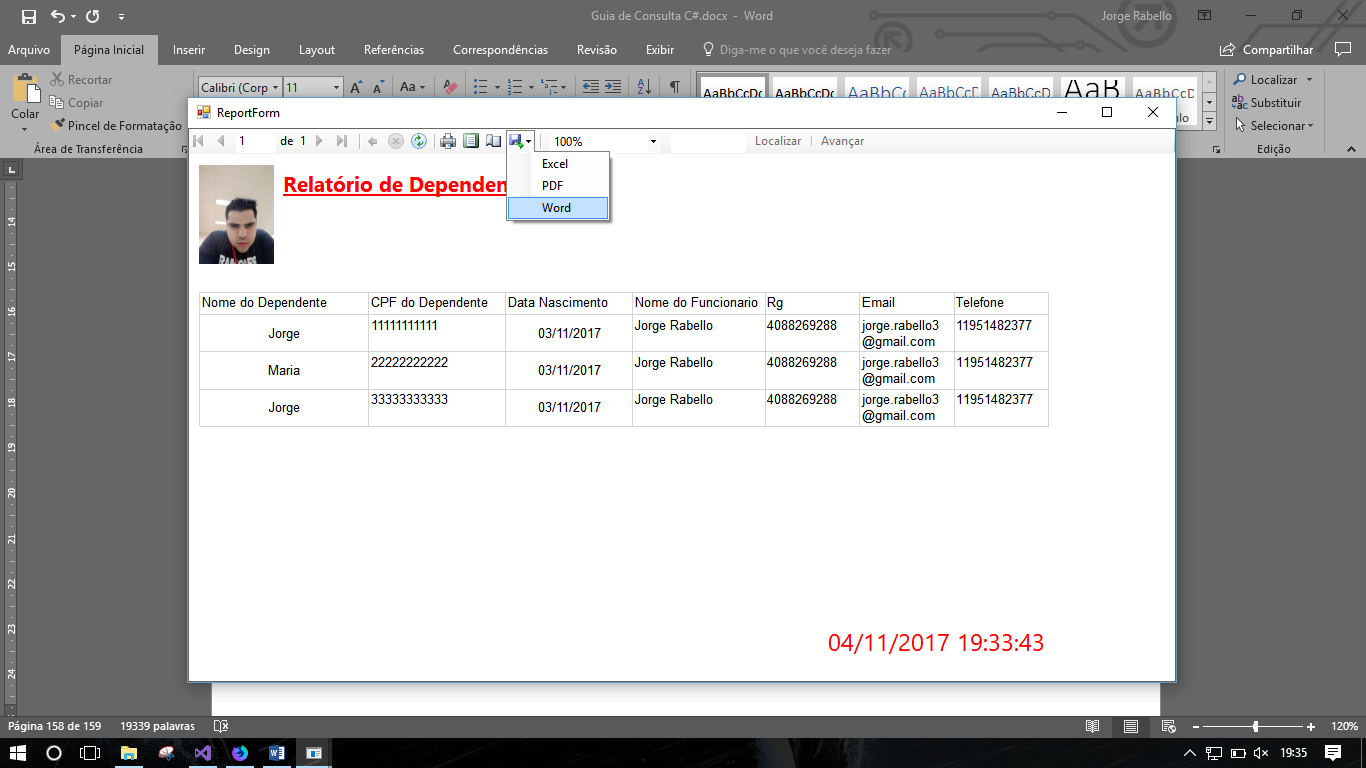
Execute o projeto clicando em Relatórios > Dependentes



O resultado será algo parecido com a imagem abaixo



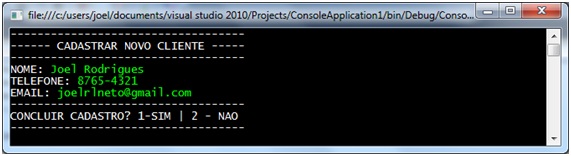
Sendo que nessa mesma tela você poderá exportar ou imprimir o relatório



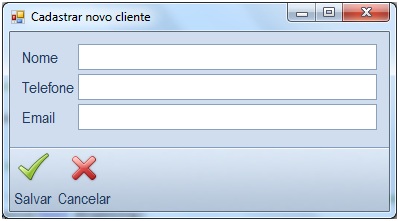
# Delegates e Eventos

## 30.1. Eventos

Quando os primeiros programas de computadores surgiram, não haviam interfaces gráficas como hoje e sendo assim o código de um software era executado de forma sequencial, linha após linha.



Com o advento das interfaces mais completas, contendo botões, caixas de texto, entre outros elementos gráficos diversos, o código das aplicações passou a ser dividido blocos que são executados de acordo com a interação do usuário (clicando em um botão, por exemplo).



Nesse caso, o processamento dos dados e a gravação ou não destes, será feito quando o usuário clicar em um dos botões. O desenvolvedor não pode prever quando e se isso ocorrerá, mas um bloco de código deverá ser executado quando o botão for pressionado.

De fato, quando o usuário clica em um botão, um **evento** chamado Click é disparado e é neste evento que o programador trabalhará para executar suas rotinas específicas.

Um evento é “algo” que ocorre em consequência de uma ação, no exemplo citado, a ação é o pressionamento do botão.

## 30.1. Delegates na prática

Crie um novo projeto do tipo Windows Forms, adicione um botão e dê dois clicks nele

private void button1\_Click(object sender, EventArgs e)

{

}

Foi criado um método void no código do form, como se pode ver acima, e nesse método vamos definir o que deve acontecer ao clicar nesse botão.

Abrindo o arquivo Form1.Designer.cs, vamos nos deparar com o seguinte código:

//

// button1

//

this.button1.Location = new System.Drawing.Point(42, 47);

this.button1.Name = "button1";

this.button1.Size = new System.Drawing.Size(75, 23);

this.button1.TabIndex = 0;

this.button1.Text = "button1";

this.button1.UseVisualStyleBackColor = true;

this.button1.Click += new System.EventHandler(this.button1\_Click);

Observe a seguinte linha

this.button1.Click += new System.EventHandler(this.button1\_Click);

Veja que está sendo atribuído ao evento de click um objeto do tipo **EventHandler** que recebe como parâmetro o método criado anteriormente. Dessa forma o método button1\_Click fica associado ao botão denominado button1.

Para testar vamos adicionar o evento de click um MessageBox

private void button1\_Click(object sender, EventArgs e)

{

MessageBox.Show("BOTÃO 01");

}

Agora crie mais um botão com evento de click com o seguinte código

private void button2\_Click(object sender, EventArgs e)

{

MessageBox.Show("BOTÃO 02");

}

Clique duas vezes no form, gerando assim um evento de Load e faça o seguinte

private void Form1\_Load(object sender, EventArgs e)

{

button1.Click += new EventHandler(button2\_Click);

}

Agora clique no botão button1, você vai notar que o botão disparou seu evento e também dispatou o evento do botão dois, isso por que nós somamos ao evento do button1 o evento de click do button2.

Agora crie os seguintes eventos

private void metodo\_click1(object sender, EventArgs e)

{

MessageBox.Show("MÉTODO 01");

}

private void metodo\_click2(object sender, EventArgs e)

{

MessageBox.Show("MÉTODO 02");

}

E altere o evento de load

private void Form1\_Load(object sender, EventArgs e)

{

button1.Click += new EventHandler(metodo\_click1);

button2.Click += new EventHandler(metodo\_click2);

}

Execute e note que agora foi acrescentado aos eventos dos botões a execução dos métodos que criamos.

Em fim vamos criar o nosso próprio delegate

public delegate void MeuDelegate(Object sender, EventArgs e);

E vamos alterar o load do form para utilizar o nosso delegate private void

Form1\_Load(object sender, EventArgs e)

{

button1.Click += delegate (Object sd, EventArgs ea)

{

MessageBox.Show("Meu Delegate ");

};

}

Em resumo os Delegates são utilizados quando se deseja passar um bloco de código para que um evento execute esse bloco de código.

No nosso exemplo associamos aos eventos de click blocos de código que exibiam MessageBox.

# Consumindo WebServices

Web Service é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes.

Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis.

Os Web Services são componentes que permitem às aplicações enviar e receber dados. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, um formato intermediário como XML', Json, CSV, etc.

Para exemplificar vamos fazer o consumo de um web service que retorna os dados de um endereço com base em um cep.

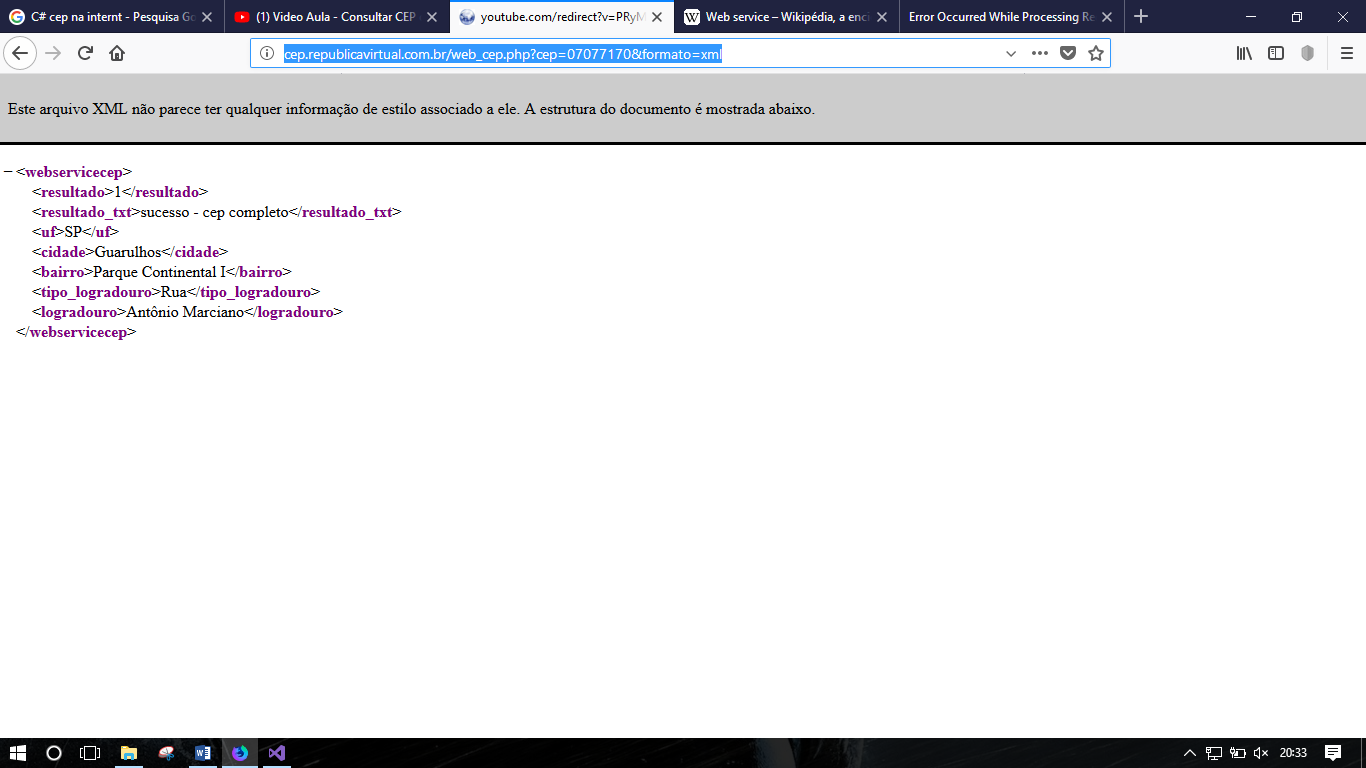
Para testar a plataforma vamos acessar o seguinte endereço:

<http://cep.republicavirtual.com.br/web_cep.php?cep=@cep&formato=xml>

Substituindo o caractere @ pro um cep válido, então deve ficar algo parecido com

<http://cep.republicavirtual.com.br/web_cep.php?cep=07077170&formato=xml>

O resultado será um arquivo xml



Contendo os dados do endereço.

Agora vamos fazer com que esses dados sejam enviados para nossa aplicação

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace BuscaCep

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

try

{

// cria um conjunto de dados vazio

DataSet ds = new DataSet();

// obtém o xml do web service com os dados consultados

// obtém o resultado da consulta

string xml = "http://cep.republicavirtual.com.br/web\_cep.php?cep=@cep&formato=xml".Replace("@cep", txtCep.Text);

// faz a leitura do xml preenchendo o conjunto de dados

ds.ReadXml(xml);

// preenche os campos do form com os dados do conjunto de dados

txtTipoLogradouro.Text = ds.Tables[0].Rows[0]["tipo\_logradouro"] as string;

txtLogradouro.Text = ds.Tables[0].Rows[0]["logradouro"] as string;

txtCidade.Text = ds.Tables[0].Rows[0]["cidade"] as string;

txtBairro.Text = ds.Tables[0].Rows[0]["bairro"] as string;

txtUF.Text = ds.Tables[0].Rows[0]["uf"] as string;

}

catch (Exception ex)

{

// trata as possível exceções

string msg = "Erro ao consultar o cep " + ex.Message;

string titulo = "Operação não realizada";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

}

Podemos também preencher o formulário automaticamente quando perder o foco no campo do cep, utilizando o evento Leave

private void txtCep\_Leave(object sender, EventArgs e)

{

try

{

// cria um conjunto de dados vazio

DataSet ds = new DataSet();

// obtém o xml do web service com os dados consultados

// obtém o resultado da consulta

string xml = "http://cep.republicavirtual.com.br/web\_cep.php?cep=@cep&formato=xml".Replace("@cep", txtCep.Text);

// faz a leitura do xml preenchendo o conjunto de dados

ds.ReadXml(xml);

// preenche os campos do form com os dados do conjunto de dados

txtTipoLogradouro.Text = ds.Tables[0].Rows[0]["tipo\_logradouro"] as string;

txtLogradouro.Text = ds.Tables[0].Rows[0]["logradouro"] as string;

txtCidade.Text = ds.Tables[0].Rows[0]["cidade"] as string;

txtBairro.Text = ds.Tables[0].Rows[0]["bairro"] as string;

txtUF.Text = ds.Tables[0].Rows[0]["uf"] as string;

}

catch (Exception ex)

{

// trata as possível exceções

string msg = "Erro ao consultar o cep " + ex.Message;

string titulo = "Operação não realizada";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

E na nossa versão final vamos fazer duas alterações:

Primeira: Criar um método que recebe o cep como parâmetro e dispara a busca, assim podemos reutilizar o nosso código

public void ConsultarCep(string cep)

{

try

{

// cria um conjunto de dados vazio

DataSet ds = new DataSet();

// obtém o xml do web service com os dados consultados

// obtém o resultado da consulta

string xml = "http://cep.republicavirtual.com.br/web\_cep.php?cep=@cep&formato=xml".Replace("@cep", txtCep.Text);

// faz a leitura do xml preenchendo o conjunto de dados

ds.ReadXml(xml);

// preenche os campos do form com os dados do conjunto de dados

txtTipoLogradouro.Text = ds.Tables[0].Rows[0]["tipo\_logradouro"] as string;

txtLogradouro.Text = ds.Tables[0].Rows[0]["logradouro"] as string;

txtCidade.Text = ds.Tables[0].Rows[0]["cidade"] as string;

txtBairro.Text = ds.Tables[0].Rows[0]["bairro"] as string;

txtUF.Text = ds.Tables[0].Rows[0]["uf"] as string;

}

catch (Exception ex)

{

// trata as possível exceções

string msg = "Erro ao consultar o cep " + ex.Message;

string titulo = "Operação não realizada";

MessageBox.Show(msg, titulo, MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

Segunda: Vamos chamar o nosso método da seguinte forma:

private void button1\_Click(object sender, EventArgs e)

{

ConsultarCep(txtCep.Text);

}

private void txtCep\_Leave(object sender, EventArgs e)

{

ConsultarCep(txtCep.Text);

}

# Bibliografia

* MANZANO, J. A. N. G. Microsoft Visual C# 2010 Express. São Paulo: Ed. Érica, 2010.
* DEITEL, H. M. *et al*. C#  Como Programar. São Paulo: Pearson Makron Books, 2003
* STELLMAN, A. e GREENE, J.  Use a Cabeça C#. Rio de Janeiro: Ed. Alta Books, 2011.
* Microsoft: C# Reference   
  <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/>
* Microsoft: C# Programing Guide  
  <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/>

1. Lista de palavras reservadas da linguagem C#  
   <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/> [↑](#footnote-ref-1)
2. Uma classe é um trecho de código fonte. [↑](#footnote-ref-2)